Kapitel IV:
Deduktion als Berechnung

(Hans-Jürgen Bürckert)

1 Einführung: Logische Programme

In den letzten Jahren haben sich die logischen Programmiersprachen, wie PROLOG und andere, zu einer der wichtigsten Anwendungen der automatischen Deduktion entwickelt. Der Ursprung ihrer Entwicklung liegt in der Erkenntnis, daß einerseits die formale Beschreibung (Spezifikation) einer Programmieraufgabe üblicherweise in Logik formuliert wird, daß andererseits eine Deduktion als Berechnung betrachtet werden kann [Hay 73]. Dabei wird ein Logikkalkül dazu verwendet, gewisse Existenzaussagen konstruktiv zu beweisen; d.h. Belegungen oder *Zeugen* für die existenzquantifizierten Variablen der Aussage anzugeben, für die die Aussage gültig ist. Im allgemeinen wird diese Aufgabe so formuliert, daß eine Existenzaussage (*Anfrage*) aus einer Menge von anderen Aussagen (*logisches Programm*) konstruktiv abgeleitet werden soll. Die dabei benötigten Belegungen der Existenzvariablen der Anfrage stellen eine Ausgabe (*Antwort*) des Programmes auf die Anfrage dar.

Beispiel:

Programm:	Leibniz ist ein Mensch.
	Sokrates ist ein Mensch.
	Sokrates ist ein Grieche.
	Jeder Mensch ist fehlbar.
Anfrage:	Gibt es fehlbare Griechen?
Antwort:	Ja, Sokrates.

Man beachte, daß die Antwort konstruktiv ist in dem Sinne, daß ein fehlbarer Grieche explizit benannt wird und nicht einfach nur „Ja“ geantwortet wird. ■

Mit dem Resolutionsprinzip und der Unifikation [Rob 65] wurde eine einfache und unter gewissen Einschränkungen auch effiziente automatische Abarbeitung einer Spezifikation in Prädikatenlogik erster Stufe möglich. Der breite Abgrund zwischen der formalen Spezifikation und dem ausführbaren Programm wäre also geschlossen. Allerdings treten dabei gewisse Probleme auf: Zum einen eignet sich die volle Prädikatenlogik erster Stufe aus mehreren Gründen nicht als Programmiersprache, zum anderen kann eigentlich erst Prädikatenlogik mit Gleichheit als Spezifikationssprache angesehen werden. Letztere eignet sich jedoch noch weniger für eine *effiziente* Programmiersprache.

Warum ist die volle Prädikatenlogik (mit Gleichheit) nicht als Programmiersprache geeignet?

Zunächst einmal sind die Suchräume enorm und ohne entsprechende Beschränkung ist daher keine einigermaßen effiziente Programmierung in dieser Logik möglich. Beschränkung der Suchräume hat aber häufig sehr komplizierte Kontrollstrukturen zur Folge. Da Implementierungen von Logikkalkülen normalerweise unvollständig sind (Implementierungsunvoll-

ständigkeit) und natürlich auch um Programmierfehler zu vermeiden, ist der Programmierer auf möglichst einfache Kontrollstrukturen angewiesen, um den Ablauf seines Programmes zu überschauen und so wenigstens im Prinzip dessen Fehlerfreiheit und (logische) Vollständigkeit zu garantieren. Entsprechendes gilt wegen der Semi-Entscheidbarkeit der Prädikatenlogik auch für die Terminierung der Programme.

Darüberhinaus benötigt man bei einer Programmiersprache eine Determiniertheit des Programmablaufs, sowie eine klare und eindeutige Form der von einem Programm berechneten Ausgabe. Letzteres ist bei der vollen, klassischen Prädikatenlogik nicht gegeben, wie wir an einem Beispiel sehen werden.

Dazu müssen wir zuerst den Begriff der „berechneten Ausgabe“ einer Deduktion (eines Beweises oder einer Widerlegung einer logischen Formel) präzisieren.

Sei P eine prädikatenlogische Formel - ein *logisches Programm* - und sei Q eine Formel - die *Anfrage* (engl. *query*) an P -, die die freien Variablen $x_1,\ldots,x_n$ enthält. Das Resultat der Anfrage zu berechnen, ist dann die Aufgabe, festzustellen, ob die Formel

$$P \Rightarrow \exists x_1 \ldots \exists x_n .\ Q$$

(konstruktiv) allgemeingültig ist, d.h. Terme $t_1,\ldots,t_n$ zu konstruieren, so daß die Formel

$$P \Rightarrow \ \forall.\sigma Q \ ^{1}$$

(mit der Substitution $\sigma := \{x_1 \leftarrow t_1,\ldots,x_n \leftarrow t_n\}$) allgemeingültig ist. Die Substitution σ wird als (ein) Resultat – eine *Antwort(substitution)* auf die Anfrage Q an das Programm P – betrachtet. Solche Substitutionen σ, mit denen die Formel $P \Rightarrow \forall.\sigma Q$ allgemeingültig ist, heißen *korrekte* Antwortsubstitutionen auf die Anfrage Q an P. Man kann die Formel mit der Antwortsubstitution auch mithilfe von Gleichungen äquivalent umschreiben als

$$P \Rightarrow \ \forall.(x_1 = t_1 \wedge \ldots \wedge x_n = t_n \ \Rightarrow \ Q)$$

oder kurz

$$P \Rightarrow \ \forall.([\sigma] \ \Rightarrow \ Q).$$

Hierbei ist $[\sigma]$ die *Gleichungsdarstellung*, d.h. die Formel $x_1 = t_1 \wedge \ldots \wedge x_n = t_n$, der Substitution $\sigma := \{x_1 \leftarrow t_1,\ldots,x_n \leftarrow t_n\}$. (Der Leser mache sich klar, daß $P \Rightarrow \forall.\sigma Q$ äquivalent ist zur Formel $P \Rightarrow \forall([\sigma] \Rightarrow Q)$. Letztere Formel besagt gerade, daß jede Lösung von $[\sigma]$ die Anfrage $\exists.Q$ wahr macht in den Modellen von P. Wir werden diese Sicht in einem späteren Abschnitt als Ausgangspunkt für eine Verallgemeinerung des Begriffs *Antwort* verwenden.)

Im allgemeinen müssen solche Terme $t_1,\ldots,t_n$ jedoch nicht existieren, wie das folgende Beispiel zeigt:

Beispiel: Sei P die Formel $P(a) \vee P(b)$ und Q die Anfrage $P(x)$. Dann ist die zugehörige Formel

---

[1]Wir kürzen gelegentlich Formeln $\forall z_1 \ldots \forall z_n .\ F$ oder $\exists z_1 \ldots \exists z_n .\ F$ wie hier mit $\forall.F$ (die All-Hülle von F) bzw. $\exists.F$ (die Existenz-Hülle von F) ab, wenn $z_1,\ldots,z_n$ alle freien Variablen in F sind.

$$P(a) \vee P(b) \Rightarrow \exists x.\ P(x)$$

allgemeingültig, aber es gibt natürlich keinen Term t, so daß die Formel

$$P(a) \vee P(b) \Rightarrow P(t)$$

allgemeingültig ist. Hier gibt es lediglich eine *indefinite* Antwort: x kann a oder b sein. Dies ist streng zu unterscheiden von der Möglichkeit, daß eine Anfrage mehrere *definite* Antworten haben kann:

Das Programm $P(a) \wedge P(b)$ hat auf dieselbe Anfrage sowohl die Antwort $\{x \leftarrow a\}$ als auch $\{x \leftarrow b\}$. ■

Um definite Antworten zu erhalten, schränkt man die Programme auf *Horn-Formeln* und die Anfragen auf Konjunktionen *atomarer* Formeln ein. Atomare Formeln (oder Atome) sind Formeln der Form $P(t_1,\ldots,t_n)$, wobei P ein Prädikatensymbol und $t_1,\ldots,t_n$ Terme sind. Horn-Formeln sind Konjunktionen von *Horn-Klauseln*, d.h. Formeln der Form

(i) H

(ii) $H \vee \neg B_1 \vee \ldots \vee \neg B_n$

(iii) $\neg B_1 \vee \ldots \vee \neg B_n$

wobei die H, $B_1,\ldots, B_n$ atomare Formeln sind. Die einzelnen Horn-Klauseln sind als allquantifiziert zu betrachten. Wir werden im folgenden die im Logischen Programmieren übliche - äquivalente - Implikationsform mit umgekehrter Reihenfolge (Dann-Wenn-Form) benutzen:

(i)	H oder auch $H \Leftarrow$.	(Fakten)
(ii)	$H \Leftarrow B_1,\ldots,B_n$.	(Regeln)
(iii)	$\Leftarrow B_1,\ldots,B_n$.	(Ziele)

Das Atom H heißt *Kopf* (engl. *head*) und $B_1,\ldots,B_n$ ist der *Rumpf* (engl. *body*) der Klausel; die Kommata sind als Konjunktion zu lesen. Fakten sind also gerade Regeln ohne Rumpf, während Ziele Regeln ohne Kopf sind. (Man kann eine Zielklausel lesen als 'false $\Leftarrow B_1,\ldots,B_n$'; dies macht den Zusammenhang zur normalen Klauselschreibweise deutlicher.)

Ein Horn-Programm ist dann eine Menge von Fakten und Regeln, die als Konjunktion der Klauseln zu interpretieren ist. Die Regeln und Fakten heißen auch definite Klauseln oder Programmklauseln. Eine *Horn-Anfrage* (engl. *query*) ist einfach eine Zielklausel (engl. *goal clause*). Die Menge aller Klauseln eines Programmes, deren Kopf das gleiche Prädikatensymbol P haben, heißt oft die *Prozedur* P oder die P definierende Klauselmenge. Eine Zielklausel kann dann als Folge von Prozeduraufrufen betrachtet werden.

Beispiel: Wir betrachten eine formale Version unseres Eingangsbeispiels.

Programm:	Mensch(Leibniz).
	Mensch(Sokrates).
	Grieche(Sokrates).
	Fehlbar(x) $\Leftarrow$ Mensch(x).
Anfrage:	$\Leftarrow$ Fehlbar(y), Grieche(y).

Antwort: y = Sokrates.

Die Antwort ist korrekt, denn die beiden nachfolgenden Formeln sind offenbar allgemeingültig:

(1) $\forall x.\text{Mensch}(\text{Leibniz}) \wedge \text{Mensch}(\text{Sokrates}) \wedge \text{Grieche}(\text{Sokrates}) \wedge (\text{Mensch}(x) \Rightarrow \text{Fehlbar}(x))$

$$\Rightarrow \exists y.\ \text{Fehlbar}(y) \wedge \text{Grieche}(y)$$

(2) $\forall x.\ \text{Mensch}(\text{Leibniz}) \wedge \text{Mensch}(\text{Sokrates}) \wedge \text{Grieche}(\text{Sokrates}) \wedge (\text{Mensch}(x) \Rightarrow \text{Fehlbar}(x))$

$$\Rightarrow \text{Fehlbar}(\text{Sokrates}) \wedge \text{Grieche}(\text{Sokrates})$$ ■

Das nachfolgende Theorem zeigt, daß dieser Zusammenhang bei Horn-Programmen immer gilt.

1.1 Theorem: Sei P ein Horn-Programm und $\Leftarrow B_1,\ldots,B_n$ eine Horn-Anfrage mit den Variablen $x_1,\ldots,x_n$. Dann ist die Formel

$$P \Rightarrow \exists x_1 \ldots \exists x_n.\ (B_1 \wedge \ldots \wedge B_n)$$

genau dann allgemeingültig, wenn Terme $t_1,\ldots,t_n$ existieren, so daß

$$P \Rightarrow \forall.\{x_1 \leftarrow t_1,\ldots,x_n \leftarrow t_n\}\ (B_1 \wedge \ldots \wedge B_n)$$

allgemeingültig ist. ■

Man beachte, daß die Formel

$$P \Rightarrow \exists x_1 \ldots \exists x_n.\ (B_1 \wedge \ldots \wedge B_n)$$

genau dann allgemeingültig ist, wenn

$$P \wedge \forall x_1 \ldots \forall x_n.\ (\neg B_1 \vee \ldots \vee \neg B_n)$$

unerfüllbar ist. Wir können diese Horn-Formel also mit dem Resolutionsprinzip widerlegen und erhalten unsere Antwortsubstitution, indem wir die dabei benötigten Unifikatoren „aufsammeln“. Wie das geschehen soll, wollen wir uns im nächsten Abschnitt näher ansehen.

2 Resolution für Horn-Formeln

Bei logischen Programmiersprachen wählt man als Kontrollstruktur des Programmablaufs – der Deduktion – ein Top-Down-Verfahren (backward-chaining), d.h. man beginnt mit der Anfrage und sucht eine Programmklausel, die mit ihr resolviert werden kann. Dadurch entsteht als Resolvente eine neue Zielklausel, die wiederum mit einer geeigneten Programmklausel resolviert wird und so weiter, bis man die leere Zielklausel erhält. Dies hat den Vorteil, daß man, wie wir sehen werden, die Antwort auf die Anfrage einfach durch Aufsammeln der benötigten Unifikatoren erhält. Man beachte, daß die Atome in den Zielklauseln negativen und die Klauselköpfe der Programmklauseln positiven Literalen entsprechen.

Wir erhalten daher folgende Resolutionsregel, die mithilfe des Programms P Zielklauseln in neue Zielklauseln transformiert (Zielreduktion).

(R) $P(t_1,\ldots,t_n) \wedge G \ \rightarrow\ \sigma B \wedge \sigma G$ (SLD-Resolution)

falls '$P(s_1,\ldots,s_n) \Leftarrow B$' eine Variante einer Klausel aus P ist,

und σ der allgemeinste Unifikator der s_i und t_i ist.

Die Regel ist so zu lesen: Unter Berücksichtigung der angegebenen Bedingung entsteht aus der alten Zielklausel bestehend aus dem fokussierten Zielliteral $P(t_1,\ldots,t_n)$ und dem Rest G aus Zielliteralen eine neue Zielklausel mit den beiden Klauselteilen σB und σG; sie ist gerade die Resolvente der alten Zielklausel mit der Variante der gewählten Programmklausel, wenn man als Resolutionsliterale das fokussierte Zielliteral und den Kopf der Klausel wählt. Dabei erhält man eine Variante einer Klausel, indem die Variablen der Klausel durch neue, bisher noch nicht verwendete Variablen ersetzt werden. Das '$\wedge$' bezeichnet hier sowohl die Konkatenation von Klauselteilen mit Klauselteilen als auch von Klauselteilen mit einzelnen Atomen. Es handelt, da die Rümpfe der Zielklauseln Konjunktionen sind (wir haben das führende '$\Leftarrow$' weggelassen), tatsächlich um eine logische Konjunktion und ist von daher gerechtfertigt.

Diese Form der Resolutionsregel enthält zwei Indeterminismen, zum einen bei der Fokussierung des Zielliterals, zum anderen bei der Auswahl der Programmklausel, mit der resolviert werden soll. Der erste Indeterminismus ist unkritisch (engl. *don't care nondeterminism*), sofern man beim zweiten stets die „richtige“ Programmklausel wählt (engl. *don't know nondeterminism*). Bei Implementierungen wird üblicherweise jeweils die erste Programmklausel bei einer fest vorgegebenen Anordnung des Programmes ausgewählt und setzt nötigenfalls zurück und versucht die nächste Klausel (Backtracking). Dieses Verfahren ist allerdings im allgemeinen nicht vollständig, wie man etwa an den Symmetrie- und den Transitivitätsklauseln im Abschnitt über Theorieunifikation (s.u.) sieht.

Der allgemeinste Unifikator σ einer Menge $\Gamma := \{s_1 = t_1,\ldots,s_n = t_n\}$ von Termpaaren ist eine Substitution σ, die die s_i und t_i syntaktisch gleich macht, der Art, daß man jeden anderen Unifikator der s_i und t_i durch weitere Spezialisierung dieses allgemeinsten Unifikators erhält. Man berechnet ihn mit dem bekannten Unifikationsalgorithmus von J.A. Robinson. Man kann den Algorithmus durch ein Regelsystem beschreiben, das Termpaarmengen Γ' in Termpaarmengen Γ'' transformiert. Dabei schreiben wir die Termpaare als Gleichungen, da die Terme durch die Unifikation gleichgemacht werden sollen. Wie in einem anderen Beitrag dieses Buches dargestellt wird, ist Unifikation ja tatsächlich ein Gleichungslöseprozeß. (Wir benutzen wieder '$\wedge$' für die Konkatenation von Gleichungssystemen; auch dies ist damit gerechtfertigt, daß Gleichungssystem logische Konjunktionen der Gleichungen darstellen.)

(T) $x = x \wedge \Gamma \rightarrow \Gamma$ (Tautologie)

(B) $x = t \wedge \Gamma \rightarrow x = t \wedge \{x \leftarrow t\}\Gamma$ (Bindung)

falls x eine Variable ist, die noch in Γ aber nicht in t vorkommt

(D) $f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n) \wedge \Gamma \rightarrow s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge \Gamma$ (Dekomposition)

(O) $t = x \wedge \Gamma \rightarrow x = t \wedge \Gamma$ (Orientierung)

Die Tautologieregel besagt, daß alle Termpaare der Form x = x weggelassen werden können, da sie nichts zur Lösung beitragen. Bei der Bindungsregel werden alle Vorkommen von x im nicht-fokussierten Rest Γ der Termpaarmenge durch t ersetzt. Bei den meisten PROLOG-Implementierungen wird hier der Test, ob die Variable im Term vorkommt (‚occur check‘), aus

Effizienzgründen weggelassen. Die Dekompositionsregel besagt, daß man die Unifikatoren zweier Terme mit demselben Top-Symbol durch Unifikation der Argumente erhält. Man beachte, daß mit dieser Regel auch Konstantenpaare c = c aus der Termpaarmenge entfernt werden, da Konstanten wie nullstellige Funktionen behandelt werden. Die Orientierungsregel ist notwendig, da wir auch für Termpaare t = x die Bindungsregel anwenden wollen.

Den Unifikator einer Termpaarmenge, mit der die Regeltransformation gestartet wurde, erhält man, wenn keine der Regeln mehr anwendbar ist - das Regelsystem terminiert stets - und die verbleibende Termpaarmenge in *gelöster* Form vorliegt. Eine Menge von Termpaaren ist gelöst, falls sie die Form $\{x_1 = t_1,\ldots,x_n = t_n\}$ hat, wobei alle x_i verschieden sind und in keinem der t_j vorkommen. Man beachte, daß dies gerade die Gleichungsdarstellung $x_1 = t_1 \wedge \ldots \wedge x_n = t_n$ einer Substitutions entspricht. Der Unifikator ist daher die Substitution $\{x_1 \leftarrow t_1,\ldots,x_n \leftarrow t_n\}$, und wir sagen, die Regeln terminieren mit der Substitution σ. Wenn die Regelanwendung stoppt, ohne daß die resultierende Termpaarmenge gelöst ist, dann enthält sie Termpaare der Form x = t, wobei x eine Variable aus t ist (‚cycle'), oder Termpaare der Form $f(s_1,\ldots,s_n) = g(t_1,\ldots,t_m)$, wobei f und g verschiedene Funktionssymbole sind (‚clash'). Die Ausgangsmenge ist dann nicht unifizierbar, die Regeln terminieren mit einem ‚failure'. Falls man die Unifikation mit einem Interpretierer für diese Regeln implementieren will, kann man natürlich noch Regeln hinzufügen, die diese beiden Failure-Fälle bereits früher abtesten, sodaß die Regelanwendung demgemäß im Falle der Nicht-Unifizierbarkeit bereits früher terminieren kann.

Beispiel: Die Termpaarmenge {f(x, y) = f(a, b), g(x) = g(a)} kann durch folgende Regelanwendungen unifiziert werden:

$$
\begin{aligned}
& f(x, y) = f(a, b) \wedge g(x) = g(a) \\
\rightarrow_D\ & x = a \wedge y = b \wedge g(x) = g(a) \\
\rightarrow_B\ & x = a \wedge y = b \wedge g(a) = g(a) \\
\rightarrow_D\ & x = a \wedge y = b \wedge a = a \\
\rightarrow_D\ & x = a \wedge y = b
\end{aligned}
$$

Der Unifikator ist $\{x \leftarrow a, y \leftarrow b\}$. Wenn die zweite Gleichung dagegen durch g(x) = g(b) ersetzt wird, terminiert die Regelanwendung nach dem dritten Schritt mit $x = a \wedge y = b \wedge a = b$, also mit einem clash-failure. ■

Die Kombination der Resolutionsregel (R) mit den Unifikationsregeln (T) - (O) ist unter dem Namen *SLD-Resolution* (oder auch LUSH-Resolution) bekannt und definiert einen vollständigen und korrekten Widerlegungskalkül für Horn-Klauselmengen [Hil 74]: Ausgehend von einer Anfrage Q an das Programm P gibt es genau dann eine endliche Folge von Zielreduktionen, die mit der leeren Zielklausel endet, wenn die Formel $P \Rightarrow \exists.Q$ allgemeingültig ist. Üblicherweise wird dabei noch eine Auswahlfunktion für das fokussierte Literal $P(t_1,\ldots,t_n)$ in der Regel (R) verwendet - daher der Name SLD-Resolution (selected *l*iteral with *d*efinite clauses). Da die Vollständigkeit unabhängig von der Wahl dieser Funktion ist, ist auch die indeterministische Regel (R) vollständig. K.L. Clark beweist noch eine stärkere Version der Vollständigkeit dieses Kalküls, wie man sie für die Verwendung desselben als Programmiersprache benötigt [Cla 79]: Zu jeder möglichen korrekten Antwort auf eine Anfrage Q an P berechnet der Kalkül eine allgemeinere Antwort, d.h. man erhält jede korrekte Antwort durch

Spezialisierung einer berechneten Antwort. Die Menge der vom Kalkül berechneten Antworten repräsentiert in diesem Sinne alle korrekten Antworten. Bevor wir das entsprechende Theorem formulieren können, müssen wir noch klarstellen, was wir unter einer „durch die Regeln berechneten Antwort σ“ verstehen wollen. Dazu sei $Q = G_0 \to \ldots \to G_n = \emptyset$ irgendeine mit der leeren Zielklausel abbrechende Folge von Regelanwendungen (R) mit den Unifikatoren $\sigma_1,\ldots,\sigma_n$. Dann ist die Komposition dieser Unifikatoren $\sigma := \sigma_n \ldots \sigma_1$ eine *berechnete Antwort* auf die Anfrage Q an das Programm P. Hierbei entsprich die Komposition $\sigma\tau$ zweier Substitutionen $\sigma = \{x_1 \leftarrow s_1,\ldots, x_m \leftarrow s_m\}$ und $\tau = \{y_1 \leftarrow t_1,\ldots, y_n \leftarrow t_n\}$ dem „Nacheinandereinsetzen“ der Substitutionen:

$$\sigma\tau = \{y_j \leftarrow \sigma t_j\colon 1 \le j \le n\} \cup \{x_i \leftarrow s_i\colon x_i \ne y_j,\ 1 \le i \le m\}.$$

Beispiel: Mit $\sigma = \{x \leftarrow a, u \leftarrow a\}$ und $\tau = \{y \leftarrow f(x, z), u \leftarrow b\}$ erhält man als Komposition die Substitution $\sigma\tau = \{x \leftarrow a, y \leftarrow f(a, z), u \leftarrow b\}$. Dies entspricht gerade dem Nacheinandereinsetzen der Substitution für Terme, die Variablen aus den Substitutionen enthalten:

$$\sigma\tau f(x, g(y, z, u)) = f(x, g(f(a, z), z, b) = \sigma(\tau f(x, g(y, z, u))).$$ ■

2.1 Theorem: Die Regel (R) mit der Unifikation (T) – (O) definiert einen korrekten und *stark* vollständigen Kalkül für Horn-Programme P und Horn-Anfragen Q:

a) Jede berechnete Antwort σ ist korrekt: Terminiert (R) mit σ, dann sind $P \Rightarrow \exists.Q$ und $P \Rightarrow \forall.\sigma Q$ allgemeingültig.

b) Zu jeder korrekten Antwort δ gibt es eine berechnete Antwort σ, die allgemeiner ist: Es gibt eine Substitution λ mit $\delta x = \lambda\sigma x$ für alle Variablen x der Anfrage Q. ■

Nun tauchen in PROLOG-Programmen manchmal auch Literale der Form s = t im Rumpf von Regelklauseln auf. Das Prädikatensymbol '=' ist dann ein ‚eingebautes‘ Prädikat, d.h. es gibt eine spezielle Form der Abarbeitung von Zielliteralen mit dem Gleichheitssymbol. Die zugehörige Prozedur kann aber auch durch die Faktklausel x = x definiert werden, so daß die Resolution eines Zielliterals s = t mit der Klausel x = x gerade der Unifikation von s und t entspricht. Mit diesem eingebauten Gleichheitsprädikat kann man daher eine „Lazy“-Version der Resolutionsregel mit den Unifikationsregeln zu einem gemeinsamen Regelsystem zusammenfassen, das einen ebenfalls korrekten und stark vollständigen Kalkül definiert. Dadurch wird eine einheitliche Darstellung der Resolution und der Unifikation möglich, die darüberhinaus den Vorteil hat, sowohl die deduktive Sichtweise des Logikkalküls als auch die operationale Sichtweise der damit beschriebenen Programmiersprache widerzuspiegeln.

(R') $P(t_1,\ldots,t_n) \wedge G \ \to\ s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge B \wedge G$ (Lazy-Resolution)
falls '$P(s_1,\ldots,s_n) \Leftarrow B$' eine Variante einer Klausel aus P ist

(T) $x = x \wedge G \ \to\ G$ (Tautologie)

(B) $x = t \wedge G \ \to\ x = t \wedge \{x \leftarrow t\}G$ (Bindung)
falls x eine Variable ist, die noch in G aber nicht in t vorkommt

(D) $f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n) \wedge G \ \to\ s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge G$ (Dekomposition)

(O) $t = x \wedge G \ \to\ x = t \wedge G$

(Orientierung)

Die in PROLOG übliche Abarbeitung der Zielklauseln von links nach rechts entspricht gerade der Unifikation der Termpaare $s_1 = t_1,\ldots,s_n = t_n$ (Unifikationsbedingungen) aus der Regel (R') und der Anwendung ihres Unifikatoren auf B und G, d.h. die Kombination entspricht tatsächlich der Regel (R). Mit diesem Regelsystem wird allerdings keine leere Zielklausel abgeleitet, sondern direkt die Antwortsubstitution (bzw. ihre Gleichungsdarstellung) produziert. Es beschreibt also tatsächlich das Verhalten der Horn-Logik als Programmiersprache (Eingabe Q und Ausgabe σ) wesentlich besser als die ursprüngliche Regel.

2.2 Theorem: Das Regelsystem (R'), (T) - (O) definiert einen korrekten und stark vollständigen Kalkül für Programme P und Anfragen Q:

a) Terminieren die Regeln mit einer Substitution σ, so sind $P \Rightarrow \exists.Q$ und $P \Rightarrow \forall.\sigma Q$ allgemeingültig.

b) Für jede mögliche korrekte Antwort terminieren die Regeln mit einer allgemeineren Antwort. ■

Wir wollen dies am Sokrates-Beispiel von oben demonstrieren.

Beispiel: Für die Anfrage '⇐ Fehlbar(y), Grieche(y).' erhalten wir folgende Zielreduktionssequenz:

$$\begin{aligned}
&\text{Fehlbar(y)} \wedge \text{Grieche(y)} \\
&\rightarrow_{R'} x = y \wedge \text{Mensch(x)} \wedge \text{Grieche(y)} \\
&\rightarrow_{B} \text{Mensch(y)} \wedge \text{Grieche(y)} \\
&\rightarrow_{R'} y = \text{Sokrates} \wedge \text{Grieche(y)} \\
&\rightarrow_{B} y = \text{Sokrates} \wedge \text{Grieche(Sokrates)} \\
&\rightarrow_{R',D} y = \text{Sokrates}
\end{aligned}$$

Im zweiten Schritt haben wir gleichzeitig die Hilfsvariable x eliminiert. Allgemein kann man Gleichungen der Form x = t, wenn x eine neue Variable war und nirgends mehr sonst in der Zielklausel vorkommt, weglassen, da sie nichts zur Antwort beitragen. Beim letzten Schritt haben wir die Regeln (R') und (D) zusammen angewendet. ■

Mehr über die Theorie des Hornklausel-Beweisens und über logisches Programmieren findet man etwa bei [Kow 79], [Llo 84], [Gal 86] und in zahlreichen PROLOG-Büchern.

3 Kompilation von logischen Programmen

Wir können nun für die logischen Programme einen einfachen Regelinterpretierer implementieren:

1 fokussiere ein Zielliteral G in der Zielklausel G

2 suche eine Klausel im Programm P, deren Kopf mit G unifiziert

- ersetze G in G durch den Rumpf der Programmklausel

- wende den Unifikator auf die entstandene neue Zielklausel an

- speichere den Unifikator
- mache weiter bei 1 mit der neuen Zielklausel

3 falls keine der Klauseln aus P mit G unifiziert,

- restauriere den Vorgänger von G (backtracking)
- mache weiter bei 1 mit der restaurierten Zielklausel

Wenn G leer geworden ist - falls man im Schritt 2 mit einer Faktklausel unifizieren kann, wird die fokussierte Zielklausel durch einen leeren Rumpf ersetzt -, gibt man die aufgesammelten Unifikatoren aus. Die meisten Implementierungen fokussieren die Zielliterale von links nach rechts und testen die Programmklauseln von oben nach unten in einer festen Reihenfolge. Diese Tiefensuche ist zwar grundsätzlich unvollständig, hat sich aber als sehr effizient erwiesen - im Gegensatz zu einer (vollständigen) Breitensuche. Außerdem können die Programme für die Tiefensuche relativ einfach kompiliert werden.

Die wesentliche Idee der Kompilationsmethoden für logische Programme liegt darin, daß eigentlich kein allgemeiner Unifikationsalgorithmus für beliebige Termpaare nötig ist. Stattdessen kann jeder Klauselkopf in einen speziellen Unifikationsalgorithmus übersetzt werden, der dessen Argumente mit den Argumenten jedes beliebigen Zielliterals unifizieren kann. Darüberhinaus wird die Abarbeitung des Programmes bei fester Programmanordnung und Reihenfolge der Ziel- bzw. Rumpfliterale (Fokussierung der Zielliterale von links nach rechts, Suche der passenden Klausel im Programm von oben nach unten und entsprechendes Rücksetzen) kompiliert.

Die bekannteste Methode zur Kompilation logischer Programme ist die Übersetzung von definiten Horn-Klauseln in eine Abstrakte PROLOG Maschine (WAM), wie sie von D.H.D. Warren [War 77 + 83] beschrieben wird. Eine WAM besteht aus mehreren „Stacks“ für die berechneten Unifikatoren und für die Backtrack-Informationen, aus Registern A_i für die Argumente der fokussierten Zielklausel, sowie aus einem Instruktionensatz für die Unifikation, das Backtracking und die Klauselauswahl. Sie kann in einer beliebigen Programmiersprache, aber auch direkt in Hardware implementiert werden. Die k-te Programmklausel $H \Leftarrow B_1,\ldots,B_n$ einer Prozedur P wird dann folgendermaßen in WAM-Code übersetzt:

Pk Speicherbereitstellung
Unifikation der Argumente von H mit den Argumentregistern
Initialisierung der Argumentregister mit den Argumenten von B_1
Aufruf der Prozedur von B_1 (Sprung an die entsprechende Stelle im Code)
…
Initialisierung der Argumentregister mit den Argumenten von B_n
Aufruf der Prozedur von B_n (Sprung an die entsprechende Stelle im Code)
Freigabe des Speichers
Rückkehr zum Aufruf von H

Ein einfaches Beispiel soll die Funktionsweise demonstrieren.

Beispiel: Betrachten wir ein Programm für die Konkatenation von Listen.

Append(nil, x, x).

Append(cons(v, xlist), ylist, cons(v, zlist) ⇐ Append(xlist, ylist, zlist).

Der Code für diese Prozedur lautet dann etwa folgendermaßen (er wird für ein aktuelles Zielliteral, dessen Prädikatensymbol Append ist und dessen Argumente in den Registern A1, A2, A3 gespeichert sind, aufgerufen und abgearbeitet):

```
Append1   if A1 ≠ nil then goto append2
          elseif A3 is a variable then push {A3 ← A2} onto unifierstack
          elseif unify(A2, A3) fails then goto append2
          else push the result of unify(A2, A3) onto unifierstack
          return with success
Append2   if A1 is a variable then push {A1 ← cons(v, xlist)} onto unifierstack and goto  a1
          elseif top-symbol(A1) ≠ cons then  return with failure
          else push {v ← arg(A1, 1)} onto unifierstack          *** 1. Argument von A1 ***
          push {xlist ← arg(A1, 2)} onto unifierstack           *** 2. Argument von A1 ***
a1        push {ylist ← A2} onto unifierstack
          if A3 is a variable then push {A3 ← cons(v, zlist)} onto unifierstack and goto  a2
          elseif top-symbol(A3) ≠ cons then  return with failure
          elseif unify(arg(A1, 1), arg(A3, 1)) fails then return with failure
          else push unify(arg(A1, 1), arg(A3, 1)) onto unifierstack
          push {zlist ← arg(A3, 2)} onto unifierstack
a2        put xlist into A1
          put ylist into A2
          put zlist into A3
          call append1
```

Der Code für die Speicherallokation und -deallokation wurde weggelassen. Jede Zeile des Codes besteht eigentlich aus einer einzigen WAM-Instruktion mit der hier ausformulierten Bedeutung. Bei 'return with failure' muß der 'Top-of-Unifierstack' zurückgesetzt werden, und bei der Initialisierung der Argumentregister ('put …') werden die Bindungen der Variablen, im Beispiel xlist, ylist und zlist, verwendet. Die Anfrage wird wie der Rumpf einer Regelklausel übersetzt, d.h. die einzelnen Zielliterale werden wie beim Schritt 'a2' transformiert. ■

Da gewisse Variablenbindungen erst dynamisch auftreten, benötigt man manchmal einen allgemeinen Unifikationsalgorithmus 'unify'; dies tritt etwa bei mehrfach vorkommenden Kopfvariablen auf (im Beispiel sind das x in append1 und v in append2). Natürlich kann der angegebene Code noch stark optimiert werden. Dazu verweisen wir auf die Orginalliteratur etwa [War 83] oder auch das WAM-Tutorium [GLLO 84].

Anwendungen dieser Kompilationsmethoden für Deduktionssysteme, die nicht auf Hornlogik beschränkt sind, stehen erst in den Anfängen [GLLO 84], [Sti 85]. Probleme bereiten hier vor allem die Unvollständigkeit der Tiefensuche und die Kompilation von Nicht-Hornklauseln.

Spezielle logische Programme können natürlich zur Effizienzsteigerung oder auch aus anderen

Gründen - etwa weil die beschriebene Tiefensuche unvollständig ist - besonders (‚von Hand') kompiliert werden. In logischen Programmiersprachen wird dies für die ‚eingebauten' Prädikate, Ein-Ausgabe-Prozeduren usw., gemacht. Aber auch im Automatischen Beweisen werden solche Verfahren im Prinzip angewandt. Theorieresolution oder Gleichheitsbehandlung (siehe in vorhergehenden Abschnitten dieses Buches) sind gewissermaßen „Handkompilate" spezieller logischer „Programme". Weitere solche Methoden wollen wir in den nächsten Abschnitten behandeln.

4 Theorieunifikation in logischen Programmen

Angenommen ein logisches Programm P enthält eine Faktklausel 'P(g(a, b)).' und wir stellen eine Anfrage '⇐ P(g(x, y)).' an das Programm. Dann müssen die Terme g(x, y) und g(a, b) unifiziert werden. Wenn nun im Programm P die Funktion g als kommutativ spezifiziert wird - wie das geschehen kann werden wir gleich betrachten -, dann können die Terme zunächst *syntaktisch* gleich gemacht werden mit der Substitution {x ← a, y ← b}, aber sie können auch *semantisch* , d.h. relativ zum Programm P, gleich gemacht werden, mit der Substitution {x ← b, y ← a}: $g(a, b) =_P g(b, a)$.

Beispiel: Das folgende Horn-Programm ist eine Spezifikation der Mendel'schen dominant-rezessiven Vererbung am Beispiel roter und weißer Blumen. Es beschreibt die Wahrscheinlichkeiten für die Phänotypen rot und weiß der Nachkommen von Blumen mit den Genotypen reinrassig rot (rr), reinrassig weiß (ww) oder gemischtrassig (rw). Die darin vorkommende Nachkommen-Funktion d ist kommutativ, da die Reihenfolge der Genotypen des Elternpaares keine Rolle spielt.

Programm:

P(d(rr, rr), red, 100). *** Alle Nachkommen reinrassig roter Blumen sind rotfarbig. ***

P(d(rr, rw), red, 100). *** Alle Nachkommen reinrassig roter und gemischtrassiger Blumen sind rot. ***

P(d(rr, ww), red, 100). *** Alle Nachkommen reinrassig roter und reinrassig weißer Blumen sind rot. ***

P(d(rw, rw), red, 75). *** 75% der Nachkommen gemischtrassiger Blumen sind rot. ***

P(d(rw, ww), red, 50). *** 50% der Nachkommen gemischtrassiger und reinrassig weißer Blumen sind rot. ***

P(d(ww, ww), red, 0). *** Kein Nachkomme reinrassig weißer Blumen ist rot. ***

P(x, white, y) ⇐ P(x, red, 1-y).

*** Ein Nachkomme ist mit Wahrscheinlichkeit y weiß, wenn er mit W. 1-y rot ist. ***

Anfrage:

⇐ P(d(x, rw), red, 50). *** Welchen Genotyp hat der eine Elternteil, wenn der andere gemischtrassig ist, ***

*** und wenn 50% ihrer Nachkommen rotfarbig sind? ***

Antwort:

x = ww. *** Der gesuchte Genotyp ist reinrassig weiß. ***

Das Zielliteral der Anfrage unifiziert syntaktisch mit keinem der Klauselköpfe. Aber unter Berücksichtigung der Kommutativität von d gelingt eine semantische Unifikation mit der fünften Faktklausel. Um dasselbe ohne die Kommutativität zu erreichen, müßten etwa alle

Klauseln, die d enthalten, noch mit vertauschten Argumenten für d hinzugefügt werden - im allgemeinen für jedes Vorkommen von d in der Klausel. Bei anderen Gleichheitseigenschaften von Funktionen, z.B. Assoziativität, ist das nicht so einfach möglich. ■

In Abschnitt 2 haben wir bereits das Gleichheitssymbol - für syntaktische Gleichheit - in logischen Programmen zugelassen; es war durch die Klausel x = x (alles ist zu sich selbst gleich) definiert. Aber wir können auch noch weitere Horn-Klauseln für das Gleichheitsprädikat hinzufügen, so daß '=' semantisch immer noch einer Gleichheit entspricht. Dazu muß '=' eine Äquivalenzrelation sein, wir benötigen also die folgenden Klauseln:

(r)	$x = x.$	(Reflexivität)
(s)	$x = y \Leftarrow y = x.$	(Symmetrie)
(t)	$x = z \Leftarrow x = y, y = z.$	(Transitivität)

Um mehr als nur die syntaktische Gleichheit zu erhalten, müssen wir die Gleichheit initialisieren, indem wir gewisse syntaktisch verschiedene Terme als ‚gleich' definieren; z. B.:

(C)	$g(x, y) = g(y, x).$	(Kommutativität)
(A)	$g(x, g(y, z)) = g(g(x, y), z).$	(Assoziativität)

Damit auch Terme wie f(x, g(a, b), z) und f(x, g(b, a), z) in diesem semantischen Sinne gleich sind, benötigt man noch Klauseln, die das Ersetzen von Gleichem durch Gleiches in Funktionen und Prädikaten ermöglichen:

(f) $f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n) \Leftarrow x_1 = y_1,\ldots, x_n = y_n.$
(für jede im Programm vorkommende Funktion f),

(p) $P(x_1,\ldots,x_n) \Leftarrow P(y_1,\ldots,y_n), x_1 = y_1,\ldots, x_n = y_n.$
(für jedes im Programm vorkommende Prädikat P).

Mit diesen Klauseln haben wir die zweistellige Funktion g als assoziativ und kommutativ spezifiziert. Natürlich können die Klauseln (C) und (A) durch beliebige andere Faktklauseln mit dem Gleichheitssymbol ersetzt werden. Prinzipiell können sogar beliebige Regelklauseln, deren Kopf das Gleichheitsprädikat enthält, hinzugefügt werden (bedingte Gleichheiten). Wir nennen alle Klauseln mit '=' als Kopfprädikat *funktionale Klauseln*, im Gegensatz zu den *relationalen Klauseln*, deren Kopfprädikat ein beliebiges anderes Prädikat ist. Die Klauseln (r), (s), (t), (f) und (p) heißen *Gleichheitsaxiome* und die Initialisierungsklauseln, im Beispiel (C) und (A), sind die *Axiome* der modellierten Gleichheitstheorie.

Nun treten bei diesen Klauseln einige Schwierigkeiten auf. Man kann a priori für ein interaktiv nutzbares Programm nicht festlegen, welche Funktionssymbole in allen möglichen Anfragen vorkommen sollen. Solche können etwa durch die sogenannte Skolemisierung in die Anfrage geraten. Möchte man z.B. wissen, ob in einem Programm P mit einer dreistelligen Prozedur P gilt, daß Werte x und y existieren, so daß P(x, y, z) für beliebige z gilt; d.h. ob die Formel $P \Rightarrow \exists x\, \exists y\, \forall z.\ P(x, y, z)$ allgemeingültig ist, dann kann man dies mit der Anfrage '$\Leftarrow$ P(x, y, h(x, y)).' feststellen, wobei h eine im Programm sonst nicht auftretende Funktion ist (Skolemfunktion).

Ein anderes Problem ist, zumindest von der Implementierung her, wesentlich schwerwiegender. Jede fest vorgegebene Reihenfolge der Klauseln bewirkt bei einer darauf basierenden Ableitungsstrategie eine Endlosschleife, wenn etwa die *beiden* Klauseln (s) und (r) benötigt werden (Implementierungsunvollständigkeit). Das liegt daran, daß beide Klauseln mit jedem Gleichheitsziel unifizieren. Ähnlich unangenehm sind die Klauseln (f) und (p). Üblicherweise versucht man dies dadurch zu vermeiden, daß man die Gleichheitsaxiome aus der Klauselmenge entfernt und stattdessen eine weitere Regel wie zum Beispiel die Paramodulation einführt; man hat die Gleichheitsaxiome wegkompiliert. Diese Paramodulationsregel läßt sich für Horn-Programme so formulieren:

(P) $P(\dots f(t_1,\dots,t_n) \dots) \wedge G \rightarrow \sigma P(\dots t \dots) \wedge \sigma B \wedge \sigma G$ (Paramodulation)

falls '$f(s_1,\dots,s_n) = t \Leftarrow B$' oder '$t = f(s_1,\dots,s_n) \Leftarrow B$' Variante einer Klausel in P und σ allgemeinster Unifikator der s_i und t_i ist.

Hierbei ist die Schreibweise $P(\dots f(t_1,\dots,t_n) \dots)$ des fokussierten Literals so zu verstehen, daß der Term $f(t_1,\dots,t_n)$ irgendwo in diesem Literal als Unterterm vorkommt, ebenfalls fokussiert und dann in der Zielklausel rechts vom Pfeil durch t ersetzt wird. Es ist zugelassen, daß das Prädikat des fokussierten Literals das Gleichheitssymbol ist.

Die Regeln (P) und (R) mit der Unifikation (T) - (O) bilden dann zusammen einen korrekten und vollständigen Kalkül für Horn-Klauseln mit Gleichheit [Höl 87]. Es ist bisher nicht bekannt, ob der Kalkül auch stark vollständig ist.

Natürlich kann man auch die Paramodulationsregel wieder entsprechend abgewandelt mit den Unifikationsregeln kombinieren (Lazy-Paramodulation):

(P') $P(\dots f(t_1,\dots,t_n) \dots) \wedge G \rightarrow s_1 = t_1 \wedge \dots \wedge s_n = t_n \wedge P(\dots t \dots) \wedge B \wedge G$

falls '$f(s_1,\dots,s_n) = t \Leftarrow B$' oder '$t = f(s_1,\dots,s_n) \Leftarrow B$' eine Variante einer Klausel in P ist.

Man kann die starke Vollständigkeit erhalten, indem man stärkere Bedingungen an die funktionalen Klauseln stellt oder zusätzliche Regeln einführt. Wir wollen hier nur kurz auf die erste Möglichkeit eingehen.

Falls das Programm keine funktionalen *Regel*klauseln enthält, d.h. falls alle funktionalen Klauseln Faktklauseln sind, und falls die Menge der funktionalen Faktklauseln ein kanonisches Termersetzungs- system bildet, wenn man sie orientiert, dann kann man (P) und (P') zur folgenden *Narrowing*-Regel vereinfachen:

(N) $P(\dots f(t_1,\dots,t_n) \dots) \wedge G \rightarrow \sigma P(\dots t \dots) \wedge \sigma G$ (Narrowing)

falls '$f(s_1,\dots,s_n) = t$' Variante einer von links nach rechts gerichteten funktionalen Faktklausel in P ist und σ allgemeinster Unifikator der s_i und t_i ist.

Auch hier gibt es wieder eine Version, die man mit den Unifikationsregeln kombinieren kann (Lazy-Narrowing):

(N') $P(\dots f(t_1,\dots,t_n) \dots) \wedge G \rightarrow s_1 = t_1 \wedge \dots \wedge s_n = t_n \wedge P(\dots t \dots) \wedge G$

falls '$f(s_1,\ldots,s_n) = t$' Variante einer von links nach rechts gerichteten funktionalen Faktklausel in P ist.

Die Regeln (R) und (N) mit der Unifikation (T) - (O) bzw. die Regeln (R'), (N'), (T) - (O) bilden korrekte und stark vollständige Kalküle für Horn-Programme mit kanonischen Termersetzungssystemen, d.h. es gelten die zu Theorem 2.1 und 2.2 analogen Aussagen.

Leider ist die Anwendung dieser Kalküle auf Horn-Programme etwa mit der Kommutativität nicht möglich, da die Gleichung (C) sich nicht richten läßt.

Man kann aber noch einen Schritt weitergehen und die funktionalen Klauseln ganz herausnehmen und durch neue parametrisierte Regeln ersetzen. Man versucht also Regelsysteme bzw. Algorithmen für die Anwendung gewisser spezieller funktionaler Klauseln zu entwickeln. Dies ist eine der Hauptaufgaben der Unifikationstheorie; zumindest unter Beschränkung auf Mengen funktionaler Faktklauseln. Man ersetzt dabei im wesentlichen die Berechnung des Unifikators in der Regel (R) durch die Berechnung von verallgemeinerten Unifikatoren, die die Terme unter Berücksichtigung der durch die entfernten Klauseln definierten Gleichheitstheorie unifizieren (Theorieunifikation).

Eine solche Menge $E := \{s_1 = t_1,\ldots,s_n = t_n\}$ funktionaler Faktklauseln axiomatisiert eine Gleichheitstheorie. Dies ist gerade die Menge aller Literale $s = t$, so daß $P_E \Rightarrow \forall.s = t$ allgemeingültig ist. Dabei ist P_E die Horn-Klauselmenge, die aus den Faktklauseln in E und den Gleichheitsklauseln (r), (s), (t) und (f) für die vorkommenden Funktionen besteht. Ein Unifikationsproblem unter der Theorie E - ein E-Unifikationsproblem - ist eine beliebige Anfrage Γ der Form '$\Leftarrow s_1 = t_1,\ldots,s_n = t_n$.' an das Programm P_E. Jede korrekte Antwortsubstitution σ auf die Anfrage Γ an P_E, also jede Substitution σ mit $\sigma s_i =_E \sigma t_i$, ist ein E-Unifikator von Γ.

Wie wir gesehen haben, hat die Menge der berechneten Antworten U die Eigenschaft, daß zu jedem E-Unifikator δ ein $\sigma \in U$ existiert mit $\delta = \lambda\sigma$ (Theorem 2.1). Sie repräsentiert also in diesem Sinne alle Antworten. Nun will man allerdings die Lösungen möglichst nicht mittels SLD-Resolution berechnen, sondern man entwickelt für feste Theorien E spezifische Algorithmen, die E-Unifikationsalgorithmen, die die Menge $U_E(\Gamma)$ - oder besser noch eine möglichst kleine, repräsentative Teilmenge $\mu U_E(\Gamma)$ - der E-Unifikatoren berechnet. Vorausgesetzt dies gelingt, dann könnten wir unsere Regel (R) wie folgt abwandeln:

(R_E) $P(t_1,\ldots,t_n) \wedge G \rightarrow \sigma B \wedge \sigma G$

falls '$P(s_1,\ldots,s_n) \Leftarrow B$' Variante aus P

und $\sigma \in U_E(s_1 = t_1,\ldots,s_n = t_n)$ (bzw. $\sigma \in \mu U_E(s_1 = t_1,\ldots,s_n = t_n)$) ist.

Wir erhalten damit einen korrekten und stark vollständigen Kalkül für Horn-Programme mit eingebauter Gleichheitstheorie E, die keine funktionalen Klauseln enthalten und bei denen somit höchstens in den Klauselrümpfen noch Gleichheitsliterale vorkommen. Letztere werden wieder mit der Klausel $x = x$ wegresolviert.

Ein interessantes Beispiel für eine solche Gleichheitstheorie sind die Booleschen Ringe. Logische Programme mit eingebauter Gleichheit der Booleschen Ringen können dazu

verwendet werden, Schaltungen zu spezifizieren, zu simulieren und zu verifizieren [BS 87], [MN 87].

5 Sorten und Typen

Eine andere Art der Effizienzsteigerung durch Spezialbehandlung gewisser Formelteile erhält man durch die sogenannten Sorten. Sorten kann man sehen als syntaktische Einschränkungen an den Aufbau der Terme, durch die man beispielsweise vorschreibt, daß eine Variable vom Typ (von der Sorte) Integer oder Real ist, d.h. nur an ganze bzw. reelle Zahlen gebunden werden darf. Dies entspricht der Typendeklaration von Variablen in Programmiersprachen wie PASCAL, man erhält also typisierte logische Programmiersprachen. Modelltheoretisch spielen die Sorten die Rolle ausgezeichneter Untermengen eines Modells des vorgegebenen Programms (bzw. der zugehörigen Formeln). Dem liegt im wesentlichen die Idee zugrunde, daß - wie auch in der Mathematik üblich - die Variablen einer Formel nicht über dem ganzen Universum, sondern über speziellen eingeschränkten Bereichen variieren.

Beispiel: Man schreibt gewöhnlich

Alle Menschen sind sterblich

$\forall x \in \text{Nat}: x > 0$

anstelle von

$\forall x$ (x ist Mensch $\Rightarrow$ x ist sterblich).

$\forall x (\text{Nat}(x) \Rightarrow x > 0)$ ■

Man betrachtet also einstellige Prädikate S als Sorten, die die Menge aller Objekte mit der Eigenschaft S modellieren. Man kann dann gewisse Regelklauseln als Mengeninklusion oder als Urbild- und Bildbereichbeschränkungen für Funktionen auffassen. Konstanten sind einerseits nullstellige Funktionen - wie bisher -, können aber andererseits auch als Basissorten aufgefaßt werden, die einelementige Mengen modellieren.

Beispiel:

ohne Sorten	mit Sorten
$\forall x.\text{Nat}(x) \Rightarrow \text{Int}(x)$	$\text{Nat} \subseteq \text{Int}$
$\forall x \forall y.\text{Nat}(x) \wedge \text{Nat}(y) \Rightarrow \text{Nat}(x + y)$	$+: \text{Nat} \times \text{Nat} \rightarrow \text{Nat}$
$\text{Nat}(0)$	$0: \text{Nat}$
$\forall x.\text{Nat}(x) \Rightarrow x > 0$	$\forall x: \text{Nat}.\ x > 0$

■

Klauseln, die nur Sorteninformation enthalten (wie z.B. die ersten drei), werden nun aus dem eigentlichen Programm entfernt und stattdessen wird bei der Resolution von Klauseln wie der letzten, bei der zum Teil Sorteninformation vorhanden ist, eine besondere *Sortenunifikation* verwendet, die die gesamten Sorteninformationen des Programmes – vereinbart etwa durch Sortendeklarationen – ausnutzt. Zum Beispiel müssen jetzt eine Variable und ein Term - auch wenn er die Variable nicht enthält - nicht mehr unbedingt unifizierbar sein; etwa wenn die

Variable einer Untersorte der Sorte des Termes angehört (semantisch heißt das, daß eine Variable aus einer Menge nicht mit einem Element einer Obermenge belegt werden kann). Zwei Variablen verschiedener Sorte können nur unifiziert werden, wenn ihre Sorten eine gemeinsame Untersorte haben (semantisch, wenn der Schnitt der entsprechenden Mengen nicht leer ist). Wir erhalten also die folgenden Unifikationsregeln:

(T) $x = x \wedge \Gamma \rightarrow \Gamma$ (Tautologie)

(B) $x = t \wedge \Gamma \rightarrow x = t \wedge \{x \leftarrow t\}\Gamma$ (Bindung)

falls x eine Variable ist, die noch in Γ aber nicht in t vorkommt
und falls die Sorte von t kleiner oder gleich der Sorte von x ist.

(B') $x = y \wedge \Gamma \rightarrow x = z \wedge y = z \wedge \{x \leftarrow z, y \leftarrow z\}\Gamma$

falls die Sorten der Variablen x und y eine maximale gemeinsame Untersorte besitzen
und z eine *neue* Variable dieser gemeinsamen Untersorte ist.

(D) $f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n) \wedge \Gamma \rightarrow s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge \Gamma$ (Dekomposition)

(O) $t = x \wedge \Gamma \rightarrow x = t \wedge \Gamma$ (Orientierung)

Falls die Sortenstruktur die unten genannten Bedingungen erfüllt, definieren diese Regeln einen Sortenunifikationsalgorithmus, der eine minimale und vollständige Menge von Sortenunifikatoren mit genau einem Element berechnet. Das heißt, es gibt immer einen allgemeinsten Unifikator wie bei der normalen Robinson-Unifikation. Jeder andere Unifikator kann durch Instantiierung gebildet werden. Die Bedingungen an die Sortenstruktur sind die folgenden:

1. Es gibt nur endlich viele Sorten und die Untersortenrelation ist eine partielle Ordnung.
2. Zwei verschiedene Sorten haben höchstens eine maximale gemeinsame Untersorte.
3. Die Sorten sind nicht leer, d.h. zu jeder Sorte gibt es eine Konstante dieser Sorte.
4. Für jede Funktion f der Stelligkeit n gibt es genau eine Sortendeklaration
   $f: S_1 \times \ldots \times S_n \rightarrow S$.

Die Sorte eines Terms ist die Bildsorte seines Topsymbols. Man nimmt gewöhnlich an, daß es zu jeder Sorte unendlich viele Variable dieser Sorte gibt (u.a. um in der Regel (B') immer neue Variable zur Verfügung zu haben). Ferner müssen die Unifikationsprobleme, auf die der Sortenunifikationsalgorithmus angewandt wird, immer aus *wohlsortierten* Termen bestehen, d.h. die Sorten der Argumentterme eines Terms mit Topsymbol $f: S_1 \times \ldots \times S_n \rightarrow S$ dürfen nur Untersorten der S_i sein.

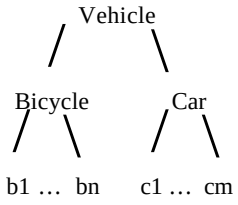
Wenn nun P ein *wohlsortiertes* Programm mit einer solchen Sortenstruktur ist, dann gelten die Theoreme 2.1 und 2.2 analog, wenn man die Unifikation durch Sortenunifikation bzw. durch obige fünf Regeln ersetzt; man erhält einen korrekten und stark vollständigen Kalkül für Horn-Programme mit Sorten. Die Sorteninformation wird also anders als bei den Typen herkömmlicher Programmiersprachen auch dynamisch im Programmablauf verwendet. Sie wirkt hier als Suchraumbeschränkung, z.B. indem die Variablen nicht der Reihe nach durch alle möglichen Konstanten instantiiert werden, bis nach mehreren Rücksetzungen die „richtige“ Einsetzung gefunden wurde. Stattdessen werden all diese Konstanten in einer Sorte

„zusammengefaßt“ und die Variable wird zu einer dieser Sorten, und erst so spät wie möglich zum konkreten Wert instantiiert.

Beispiel: (x: Car an einer Argumentposition steht für eine allquantifizierte Variable der Sorte 'Car')

Programm:	ohne Sorten	mit Sorten
	Vehicle(x) ⇐ Bicycle(x)	Bicycle ⊆ Vehicle
	Vehicle(x) ⇐ Car(x)	Car ⊆ Vehicle
	Bicycle(b1)	b1: Bicycle
	…	…
	Bicycle(bn)	bn: Bicycle
	Car(c1)	c1: Car
	…	…
	Car(cm)	cm: Car
	Car(mycar)	mycar∈ Car
	Has(x, tires) ⇐ Vehicle(x)	Has(x:Vehicle, tires)
	Has(x, doors) ⇐ Car(x)	Has(x:Car, doors)
	Owns(alan, mycar)	Owns(alan, mycar)
Anfrage:	⇐ Has(x, tires), Has(x, doors), Owns(alan, x).	
Antwort:	x = mycar	

Sortenstruktur



Ein gewöhnlicher PROLOG-Interpretierer testet alle angegeben Fahrrad- und Autoinstanzen, bevor er die Instanz 'mycar' erfolgreich ausgibt, während ein Interpretierer, der Sorten bearbeiten kann, die Variable zuerst zu 'Vehicle', dann zu 'Car' abschwächt und letztlich 'mycar' ebenfalls erfolgreich ausgibt. Man spart hier also je nach Anordnung der Klauseln und Anzahl der Instanzen umfangreiches Backtracking. ■

Die aktuelle Forschung in diesem Bereich beschäftigt sich damit, allgemeinere Sortenstrukturen zu untersuchen - nicht nur im Bereich des logischen Programmierens, sondern auch für die volle Logik erster Stufe beim automatischen Beweisen [GM 86], [Sch 86].

6 Logische Programme mit Constraints

Eine relativ neue Entwicklung im logischen Programmieren ist die Erweiterung mit Constraint-Solving-Methoden (vgl. [Hen89]). Die Grundidee dabei ist recht einfach: Wenn wir Unifikationsprobleme als Gleichungen betrachten, die wir lösen müssen, wenn wir einen Resolutionsschritt ausführen wollen, warum werden wir dann nicht gleich allgemeiner? Wir könnten ja auch negative Gleichungen zulassen oder ganz allgemein irgendwelche Ausdrücke mit Variablen, für die wir Lösungen suchen müssen. Nun, solche zu lösenden Anforderungen an Variablen sind in der KI auch unter dem Begriff „Constraints“ bekannt.

Erinnern wir uns nochmal an die „Lazy“-Versionen der Resolution, Paramodulation usw. Dort wurden Gleichungen in die Resolvente eingeführt, welche dann mit speziellen Verfahren (etwa mit E-Unifikation) gelöst wurden oder, wie man es auch sehen kann, in Lösungen transformiert wurden. Diese Idee können wir wie oben angedeutet verallgemeinern und gewisse syntaktisch

ausgezeichnete Teile der Rümpfe unserer Programmklauseln – die wir als Constraints bezeichnen – nicht mittels Resolution, sondern durch spezielle Constraint-Löseverfahren lösen. Wir können sogar noch einen Schritt weiter gehen und Constraints gar nicht mehr lösen, sondern diese nur in einfachere Constraints transformieren, die wir als „gelöste“ Formen betrachten und etwa als Antwort-Constraints akzeptieren können.

Etwas Ähnliches ist uns bereits im letzten Abschnitt begegnet. Dort hatten wir noch zusätzlich Typ- oder Sorteninformation für die Variablen unserer Gleichungen. Sorten sind aber einfach ausgezeichnte einstellige Prädikate, die wir nicht wegresolvieren wollten, sondern – sozusagen versteckt – als Teil der Antwort ausgegeben haben. Zumindest in den Zwischenschritten des obigen „Car“-Beispiels war die Variable nicht an einen Term gebunden, sondern sozusagen an eine Sorte. In den Zwischenschritten wäre also eine Antwort x:Car gewesen. Hier sieht man auch gleich, warum solche Constraint-Antworten sinnvoll sein können: Sie fassen ganze Mengen von Antworten zu einer sogar „sinnvolleren“ Antwort zusammen: im Beispiel gerade all die Antworten x = c1 … x = cm und x = mycar (ein solches Zusammenfassen von Antworten ist unter dem Schlagwort „intensional answers“ Forschungsgegenstand in der Logik-Programmierung und bei deduktiven Datenbanken).

Die zwei wichtigsten Vertreter – im Sinne einer formal-logischen Beschreibung – des „Constraint Logic Programming“ sind das CLP-Schema von Jaffar und Lassez [JL 87] und das allgemeinere Schema von Höhfeld und Smolka [HS 91]; vgl. aber auch [Hen 89]. Der wesentliche Unterschied zwischen den Jaffar-Lassez- und dem Höhfeld-Smolka-Schema liegt in der Interpretation der Constraints. Beim Jaffar-Lassez-Schema werden nur Constraints über einem einzigen festen Bereich, z.B. (lineare) Gleichungs- und Ungleichungssysteme, über den reellen Zahlen zugelassen, während das Höhfeld-Smolka-Schema Constraints über mehreren Bereichen zuläßt. Bei Ersterem ist dann die Lösung eines Constraints eine Belegung der Constraint-Variablen – also Werte im ausgezeichneten Bereich. Hier fällt auch die Unifikation darunter: Wenn man als Bereich für die Term-Gleichungen gerade die Term-Algebra wählt, sind die Unifikatoren nichts anderes als solche Belegungen der Variablen mit Werten in dieser Term-Algebra. Beim Höhfeld-Smolka-Schema ist die Lösung eines Constraints ein Paar bestehend aus einem der Bereiche und der Belegung der Variablen in diesem Bereich (der Leser beachte die Analogie zur Semantik existenzquantifizierter Formeln in Kapitel 2, die aus einer Interpretation – Diskursbereich und Interpretationsfunktion der Symbole – zusammen mit einer Belegung der Variablen besteht, welche die Formel wahr macht!).

Wir wollen das etwas präziser einführen. Dabei werden wir uns auf den Fall mit mehreren Constraint-Bereichen konzentrieren, das Jaffar-Lassez-Schema mit nur einem Constraint-Bereich ist ja dann ein „einfacher“ Spezialfall. Wir definieren also ein Constraint-System über einer abzählbaren Menge V von Variablen als ein Paar, bestehend aus einer Menge von Constraint-Bereichen, die Constraint-Theorie, und einer Menge von Constraints. Dabei soll jedem Constraint C eine endliche Sequenz von Variablen $(x_1,\ldots,x_n)$, die von C beschränkten Variablen, zugeordnet sein; wir schreiben dann kurz $C(x_1,\ldots,x_n)$. Ein Constraint-Bereich ist einfach eine beliebige Werte-Menge.

Für jeden Constraint $C(x_1,\ldots,x_n)$ verlangen wir, daß es zu jedem Constraint-Bereich $\Im$ eine

eventuell leere Menge $SOL_{\mathfrak{I}}(C)$ von Lösungen des Constraints in $\mathfrak{I}$ gibt. Solche $\mathfrak{I}$-Lösungen sind dabei einfach ausgezeichnete Zuordnungen von Variablen-Werte-Paaren, also Abbildungen von V in den Bereiche $\mathfrak{I}$. Da für die Lösungen ausschließlich die dem Constraint zugeordneten Variablen von Bedeutung sind, kann man die Lösungen ähnlich wie Substitutionen durch endliche Erzeugersequenzen $\sigma = (x_1 \leftarrow a_1,\ldots, x_n \leftarrow a_n)$ von Variablen-Wert-Paaren mit $a_i \in \mathfrak{I}$ beschreiben. Offensichtlich sind die Lösungen jedoch durch solche Sequenzen nicht eindeutig festgelegt, sondern sie können in den nicht vorkommenden Variablen variieren: $\sigma = (x_1 \leftarrow a_1,\ldots, x_n \leftarrow a_n)$ spezifiziert alle $\mathfrak{I}$-Lösungen von $C(x_1,\ldots,x_n)$, welche den Variablen x_i des Constraints C die Werte a_i zuweisen. In den Beispielen werden wir etwas unsauber auch oft die Erzeuger selbst als Lösungen bezeichnen.

Um die Constraints verwenden zu können, braucht man zusätzlich, daß die Menge der Constraints bezüglich ihrer Lösungsmengen abgeschlossen ist unter Variablen-Umbenennung und Schnitt-Bildung. Zu jedem Constraint $C(x_1,\ldots,x_n)$ muß es für jede Variablen-Umbennung $\rho = \{x_1 \leftarrow y_1,\ldots, x_n \leftarrow y_n\}$ seiner Variablen einen Constraint $C^*(y_1,\ldots,y_n)$, der in jedem Bereich dieselben Lösungen hat unter obiger Variablenumbenennung: Die Erzeuger σ^* der Lösungen von C* erhält man, wenn man in den Erzeugern σ der Lösungen von C die x_i durch die y_i ersetzt. Wir denotieren den umbenannten Constraint C* auch mit ρC. Außerdem muß es zu je zwei Constraints C und D einen Constraint E geben, so daß $SOL_{\mathfrak{I}}(E) = SOL_{\mathfrak{I}}(C) \cap SOL_{\mathfrak{I}}(D)$. Wir nennen E auch die *Konjunktion* der Constraints C und D.

Beispiel: (a) Eine Constraint-Theorie könnte etwa durch eine Menge von Axiomen (in Prädikatenlogik erster Stufe oder einer syntaktischen Variante) spezifiziert sein, so daß die Modelle – oder genauer ihre Universen – gerade die Constraint-Bereiche sind. Für die Constraints kann man irgendeine Menge von Formeln mit freien Variablen auszeichnen, die unter Umbennung der freien Variablen und unter logischer Konjunktion abgeschlossen ist. Eine solche Menge erfüllt gerade die obigen Anforderungen: Die Lösungen eines Constraints $C(x_1,\ldots,x_n)$ in einem Bereich $\mathfrak{I}$ sind gerade diejenigen Variablen-Belegungen, die den existentiellen Abschluß des Constraints, also die Formel $\exists x_1,\ldots,x_n\, C(x_1,\ldots,x_n)$, im Modell $\mathfrak{I}$ wahr machen.

(b) Nehmen wir an, wir hätten nur einen einzigen Constraint-Bereich, die reelen Zahlen, und betrachten wir Systeme linearer Gleichungen als Constraints. Auch diese erfüllen die Anforderungen, da Umbennung der Variablen nichts an den Lösungen ändert. Die Konjunktion zweier solcher Gleichungssysteme ist gerade die Konkatenation der beiden Systeme. Ein Constraint könnte etwa gegeben sein durch das Gleichungssystem $3x_1 + x_2 - x_3 = 0$, $x_1 + x_2 - 5 = 0$. Er hätte dann die Lösungen $(x_1 \leftarrow 0, x_2 \leftarrow 5, x_3 \leftarrow 5)$, $(x_1 \leftarrow 1, x_2 \leftarrow 4, x_3 \leftarrow 7)$, usw.

(c) Weitere Beispiele für Constraints sind, wie bereits erwähnt, Unifikationsprobleme. Hier haben wir wiederum einen einzigen Constraint-Bereich, die Term-Algebra. Eine Lösung eines Constraints gegeben durch die Term-Gleichungen $f(x_1, x_2) = f(a, x_3)$, $g(x_1, a) = g(x_4, x_4)$ wäre etwa durch den Unifikator $\{x_1 \leftarrow a, x_2 \leftarrow x_3, x_4 \leftarrow a\}$ definiert. Genau genommen muß man hier vorsichtiger sein und darf bei den Lösungen Constraint-Variablen und Term-Variablen der Werte nicht mischen. Man muß also zwei verschiedene Variablenmenge für die Terme in den Constraints und für die Terme in den Lösungen nehmen. Aus dem Unifikator ergäbe sich

dann etwa folgende Lösung: ($x_1 \leftarrow a$, $x_2 \leftarrow v$, $x_3 \leftarrow v$, $x_4 \leftarrow a$).

(d) Eine viel bessere Sichtweise erhält man jedoch, wenn man im Beispiel (c) als Constraint-Bereich die Grundterm-Algebra (das Herbrand-Universum) wählt. Die (allgemeinsten) Unifikatoren eines Constraints, also eines Gleichungssystems, kann man dann gerade als Darstellung für die gelöste Form des Gleichungssystems betrachten: Die Grundinstanzen des Unifikators sind exakt die Lösungen des Gleichungssystems im Herbrand-Universum. Der Unifikationsalgorithmus transformiert also das Gleichungssystem

$$f(x_1, x_2) = f(a, x_3),\ g(x_1, a) = g(x_4, x_4)$$

in die gelöste Form $x_1 = a,\ x_2 = x_3,\ x_4 = a$

die dem Unifikator $\{x_1 \leftarrow a, x_2 \leftarrow x_3, x_4 \leftarrow a\}$ entspricht. ■

Wie diese Beispiele zeigen, kommt es eigentlich nicht darauf an, daß der Constraint-Bereich wirklich angegeben wird, – einige Leser werden sich vielleicht schon gefragt haben, wie denn der Bereich der reellen Zahlen spezifiziert sein soll, in Logik erster Stufe ist dies jedenfalls nicht möglich –, sondern daß es ausreicht, Lösungsverfahren für die Constraints zu kennen. Man könnte demnach ein Constraint-System auch leicht modifiziert dadurch definieren, daß es aus einer Menge von Constraints besteht, sowie ein (oder mehrere) Lösungsverfahren, die jedem Constraint C – evtl. in Abhängigkeit von einem *Parameter* B – eine Lösungsmenge $SOL_B(C)$ zuweist. Ein Constraint wäre dann lösbar, wenn *eines* der Lösungsverfahren SOL_B für den Constraint C eine nicht-leere Lösungsmenge errechnet. Wie wir gleich sehen werden, kann man sogar darauf verzichten, die Lösungsmenge zu berechnen: Es reicht aus, wenn das Verfahren sagt, ob der Constraint *lösbar* ist. Wie oben bereits angesprochen, kann man stattdessen auch verlangen, daß man ein Verfahren hat, daß jeden (lösbaren) Constraint in eine gelöste Form transformiert, also in einen einfacheren Constraint, dem man die Lösbarkeit direkt „ansieht“ oder von dem man weiß, daß er lösbar ist. Die von uns oben eingeführte Version hat jedoch den Vorteil, daß sie mit der üblichen prädikatenlogischen Semantik verträglich ist, wie wir gleich sehen werden. Die Varianten kann man sich einfach als *Realisierungen* eines Constraint-Systems vorstellen, die der Designer einer CLP-Sprache über einem gegebenen Constraint-System festzulegen hat.

Wenn wir also ein solches Constraint-System vorgegeben haben, betrachten wir im folgenden *Constrained-Klauseln*; das sind Horn-Klauseln mit zusätzlichem Constraint:

(i)	$H \Leftarrow \ \| \ C$	(Fakten)
(ii)	$H \Leftarrow B_1,\ldots,B_n \ \| \ C$	(Regeln)
(iii)	$\Leftarrow B_1,\ldots,B_n \ \| \ C$	(Ziele)

sowie der zusätzliche Sonderfall einer „leeren“ Constrained-Klausel, die nur aus einem Constraint besteht

(iv)	$\Leftarrow \ \| \ C$	(Antwort)

Dabei sind H und die B_i Atome und C ist ein Constraint. Wir lassen nur einen Constraint zu, was aber keine Einschränkung ist, da die Constraints abgeschlossen unter Konjunktion sind. Die im Constraint C vorkommenden Variablen nennen wir die durch C *beschränkten* Variablen.

Nicht alle Variablen einer Klausel müssen beschränkt sein und wir wollen auch zulassen, daß im Constraint Variablen vorkommen, die in den Klausel-Atomen selbst nicht auftauchen. Für den mit Logik vertrauteren Leser möchten wir hier nur erwähnen, daß man das vermeiden kann, indem man solche Variablen im Constraint existentiell bindet. Dann muß man natürlich fordern, daß die Constraints in Bezug auf ihre Lösungen – wie oben bei der Variablenumbenennung und der Konjunktion von Constraints – abgeschlossen sind unter solch einer Operation.

Die Bedeutung der Constraints ist als eine zusätzliche Bedingung oder Einschränkung an die Klausel zu sehen: Eine Constrained-Klausel ist wahr über einem Constraint-Bereich, wenn sie für alle Lösungen ihres Constraints in diesem Bereich zu wahr interpretiert wird (nicht im Constraint vorkommende Variablen der Klausel werden als universell quantifiziert betrachtet und dürfen dabei beliebig belegt werden). Dabei spielen die Constraint-Bereiche gerade die Rolle der zugelassenen Universen für die Interpretationen; d.h. die Constraint-Theorie schränkt die möglichen Interpretationen im Hinblick auf die Universen ein. Wenn wir also nur einen einzigen Constraint-Bereich haben, etwa die reellen Zahlen, so werden unsere Constrained-Klauseln ausschließlich über diesem Bereich interpretiert, wobei die Constraints gewissermaßen als *Filter* für zulässige Belegungen der beschränkten Variablen fungieren. Man liest also eine Constrained-Klausel K || C, wobei K eine Horn-Klausel (Fakt, Regel oder Ziel) ist, als K $\Leftarrow$ C, wobei alle Variablen allquantifiziert sind. Man sieht übrigens leicht, daß der Constraint auch äquivalent als zusätzliche Bedingung durch *Konjunktion* in den Rumpfteil der Horn-Klausel aufgenommen werden kann, anstatt ihn per *Implikation* als Vorbedingung an die ganze Horn-Klausel zu hängen. Dies liegt einfach daran, daß die beiden Formeln '(F $\Leftarrow$ G) $\Leftarrow$ H' und 'F $\Leftarrow$ (G $\wedge$ H)' äquivalent sind. Damit ist es auch gerechtfertigt daß wir in der Notation für unsere Constrained-Klauseln auf Klammern verzichtet haben.

Ein logisches Programm über einem Constraint-System ist nun eine Menge von Fakten und Regeln mit (lösbaren) Constraints. Eine Anfrage an das Programm ist eine Zielklausel mit Constraint. Ähnlich wie ohne Constraints wollen wir die Anfrage in eine Antwort transformieren, wobei wir jetzt als Antworten nicht mehr Substitutionen, sondern Constraints (evtl. jedoch nur gelöste Formen der Constraints) verwenden wollen. Die folgende Constrained-Resolutionsregel leistet dies. Zur Vereinfachung nehmen wir dabei an, daß die Klauseln keine Terme enthalten. Man kann dazu fordern, daß Terme nur Teil der Constraints sein dürfen, über deren syntaktische Form wir ja keinerlei Annahmen gemacht hatten. Will man Terme in den Klauseln zulassen, so kann man diese herausfalten, bevor man die Regel anwendet: Man ersetzt einfach jeden vorkommenden Term t durch eine neue Variable x und erweitert den Constraint jeweils um die Gleichung x = t. Wie schon erwähnt, müssen die Constraints dann solche Gleichungen zulassen und das Lösen der Gleichungen muß durch das Constraint-Löseverfahren gewährleistet sein.

(CR) $P(x_1,\ldots,x_n) \wedge G \parallel D \quad \rightarrow \quad B \wedge \rho G \parallel E$ (Constrained Resolution)

falls '$P(y_1,\ldots,y_n) \Leftarrow B \parallel C$' eine Variante einer Klausel des Programms ist,
und $SOL_{\Im}(E) \neq \emptyset$ ist für einen Constraint-Bereich $\Im$,
wobei E die Konjunktion von ρC und D und $\rho := \{y_1 \leftarrow x_1,\ldots, y_n \leftarrow x_n\}$ ist.

Wie schon angedeutet, kann man natürlich den Test auf Lösbarkeit des Constraints E weglassen und stattdessen (direkt oder in einem Folgeschritt) E in eine gelöste Form transformieren, was natürlich fehlschlagen kann, wenn E keine Lösungen hat (vgl. die Kombination der Lazy-Resolution mit den Unifikationsregeln, die Gleichungssysteme in ihre gelöste Form, den Unifikator, transformieren).

Diese Constrained-Resolution-Regel sollte nun ähnlich wie in den Fällen ohne Constraints zweierlei leisten: Zum einen sollte man damit jede Anfrage verifizieren können wie in Theorem 2.1a, zum anderen sollte man damit stets Antworten berechnen können, die allgemeiner sind als jede vorgegebene korrekte Antwort wie in Theorem 2.1b. Leider ist dies so nicht direkt der Fall und außerdem haben wir noch die Schwierigkeit, daß wir den Begriff „allgemeinere Antwort" präzisieren müssen; Antworten sind ja jetzt nicht mehr Substitutionen, sondern beliebige Constraints oder gelöste Formen.

Zunächst zu letzterem: Wir erinnern uns an die Sicht in obigem Beispiel (d), wo Unifikatoren als gelöste Constraints betrachtet wurden, deren Grundinstanzen (also deren Lösungen) gerade die Lösungen der Ausgangs-Gleichungssysteme über dem Herbrand-Universum darstellten. Damit liegt die Idee nahe die Allgemeiner-Relation für Constraints über ihre Lösungen zu definieren. Ein Constraint C ist allgemeiner als ein Constraint D, falls C mindestens die Lösungen von D hat, d.h. falls für jeden Bereich $\mathfrak{I}$ gilt: $SOL_{\mathfrak{I}}(C) \supseteq SOL_{\mathfrak{I}}(D)$. Auch mit dieser Verallgemeinerung ist man noch nicht am Ziel, denn es zeigt sich, daß man mehr als eine Antwort berechnen muß, so daß deren Lösungen mindestens die Lösungen der vorgegebenen korrekten Antwort umfassen.

Wir haben noch nicht definiert, was korrekte Antworten sein sollen: Ein Antwort-Constraint C heißt *korrekte* Antwort auf die Anfrage Q an das Constrained-Programm P über einem gegebenen Constraint-System, falls die Formel $P \Rightarrow \forall(C \Rightarrow Q)$ wahr ist in allen Interpretation, deren Universen Constraint-Bereiche sind, oder kurz, falls $P \Rightarrow \forall(C \Rightarrow Q)$ *gültig* ist über dem Constraint-System. Die Formel $P \Rightarrow \forall(C \Rightarrow Q)$ ist die entsprechende Verallgemeinerung der Formel $P \Rightarrow \forall.\sigma Q$, die wir für die Definition der korrekten Antwort-Substitution verwendet hatten.

Theorem 6.1: Die Regel (CR) definiert einen korrekten und stark vollständigen Kakül für Constrained-Horn-Programme P und Anfragen Q über einem Constraint-System:

a) Berechnete Antwort-Constraints sind korrekt: Terminiert (CR) mit C, dann ist die Formel $P \Rightarrow \forall(C \Rightarrow Q)$ gültig über dem Constraint-System.

b) Zu jeder korrekten Antwort C kann man mit (CR) Antworten $C_1,\ldots,C_n$ ableiten, so daß $SOL_{\mathfrak{I}}(C) \subseteq SOL_{\mathfrak{I}}(C_1) \cup \ldots \cup SOL_{\mathfrak{I}}(C_n)$ für alle Constraint-Bereiche $\mathfrak{I}$.

Im Gegensatz zum klassischen Fall ohne Constraints muß man hier also mehrere Antworten berechnen, um die starke Vollständigkeit zu erhalten. Dies erinnert stark an das eingangs erwähnte Beispiel, wo wir indefinite Antworten hatten. In der Tat gilt folgendes: Wenn das Constraint-System – etwa in einer Axiomatisierung – selbst wieder durch definite Horn-Klauseln spezifiziert ist und die Constraints nur Konjunktionen von Atomen sind, also selbst Zielklauseln sind, dann gibt es zu jeder korrekten Antwort C *eine* mit (CR) berechnete Antwort

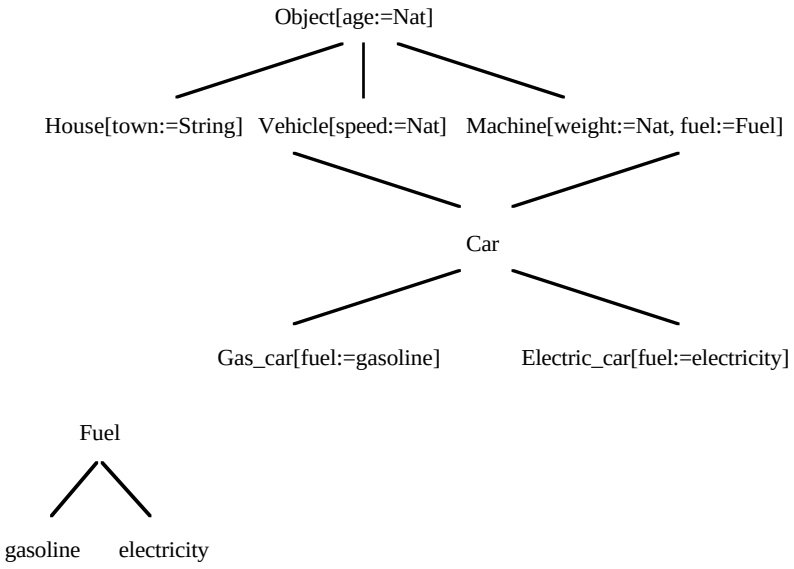
C' mit $SOL_{\mathfrak{I}}(C) \subseteq SOL_{\mathfrak{I}}(C')$ für alle Constraint-Bereiche $\mathfrak{I}$.

Ferner muß man anmerken, daß man im allgemeinen sogar unendlich viele Antworten berechnen muß. Teil (b) des Theorems gilt eigentlich nur für sogenannte lösungskompakte Constraint-Systeme, etwa solche, die durch eine Axiomatisierung in Prädikatenlogik erster Stufe gegeben sind (vgl. [Mah 87], [HS 91]).

Für den Teil (a) ist zu sagen, daß man hier ebenfalls anders als im klassischen Fall mit der Berechnung einer Antwort *nicht* gezeigt hat, daß die Existenzhülle der Anfrage eine logische Konsequenz des Programmes (unter der Constraint-Theorie) ist. Man hat damit tatsächlich nur die angegebene bedingte Aussage, die man in etwa lesen kann als: In den Modellen des Programms, in denen C gilt (also Lösungen hat), ist auch die (Existenzhülle) der Anfrage wahr. Man kann zeigen, daß man für den Beweis der Anfrage, also für die Gültigkeit von $P \Rightarrow \exists.Q$ (ähnlich wie in (b)) mehrere (i.A. unendlich viele) Antworten generieren muß (vgl. [Bür 91]).

Wir wollen abschließend noch eine interessante Instanz des Höhfeld-Smolka-Schemas streifen: Logisches Programmieren mit Feature-Unifikation [AS 87] oder auch Ψ-Unifikation [AN 85], die z.B. in den Unifikationsgrammatiken eine zentrale Rolle spielt [Shi 86]. Bei den Features handelt es sich um eine Möglichkeit die logischen Programmiersprachen um taxonomische Information - wie die Sorten - zu erweitern, die aber stärker strukturiert sind. Die Sorten können jetzt noch gewisse Eigenschaften mit zugehörigen Zugriffsfunktionen besitzen – eben die Features –, entsprechend etwa den Slots der Frames bei frame.basierten oder bei objekt-orientierten Programmiersprachen. In [AN 85] wird diese Methode explizit für eine logische Programmiersprache mit eingebauter ‚IS-A'-Taxonomie eingeführt. Das nachfolgende Beispiel soll dies verdeutlichen.

Beispiel: Features sind spezielle einstellige Funktionen, die Notation Object[age:=Nat] entspricht daher der oben benutzten Sortennotation age: Object → Nat und die Notation wird auch mit Konstanten oder komplexeren Feature-Ausdrücken benutzt; so steht Gas_car[fuel:=gasoline] für die Sortennotation fuel: Gas_car → gasoline. Dabei beachte man, daß Konstanten hier auch als Sorten aufgefaßt werden, die einelementige Mengen modellieren.



Die Untersorten erben die Features der Obersorten; entprechend haben Konstante einer Sorte die Features dieser Sorte. So hat etwa die Sorte 'Car' die Features 'age', 'speed', 'weight' und 'fuel', obwohl bei ihr kein Feature explizit spezifiziert wurde. Eine Sorte braucht natürlich nicht unbedingt Features zu haben - im Beispiel die Sorte 'Fuel' -, wenn für eine feature-lose Sorte keine Obersorten deklariert oder alle Obersorten ebenfalls feature-los sind. Diese entsprechen dann den Sorten wie sie im Abschnitt 5 behandelt wurden. ■

Eine solche Feature-Hierarchie wird nun als Spezifikation einer Constraint-Theorie aufgefaßt. Als Constraints läßt man Feature-Ausdrücke zu, die ähnlich wie die Sorten mit Variablen kombiniert werden: x:Electric_car[speed:=50, age:=1, weight:=2]. Man beachte, daß die Feature-Constraints und die Feature-Hierarchie ähnlich wie die Sorten lediglich eine syntaktische Variante einer eingeschränkten Prädikatenlogik erster Stufe ist, die allerdings recht anschaulich die Mengen- und Attributinterpretation der Sorten und Features widerspiegelt.

Ein Programm besteht nun aus einer Spezifikation der Feature-Hierarchie und Constrained-Klauseln mit Feature-Constraints. Es sieht dann für obige Feature-Sortenstruktur etwa wie folgt aus.

Beispiel:

Program:	Object[age:=Nat]
	House[town:=String] ⊆ Object
	Vehicle[speed:=Nat] ⊆ Object
	Machine[weight:=Nat, fuel:=Fuel] ⊆ Object
	Car ⊆ Vehicle
	Car ⊆ Machine
	Gas_car[fuel:=gasoline] ⊆ Car
	Electric_car[fuel:=electricity] ⊆ Car
	gasoline: Fuel
	electricity: Fuel
	mycar1: Gas_car[age:=5, weight:=2, speed:=100]
	mycar2: Electric_car[speed:=50, age:=1, weight:=2]
	Is_town_car(x) ⇐ Is_slow(x) || x:Car
	Is_slow(x) || x:Car[speed:=50]
	Owns(alan, mycar1)
	Owns(alan, mycar2)
Anfrage:	⇐ Is_town_car(x) || x:Car.
Antwort:	x:Car[speed:=50] ■

Der Vollständigkeit halber wollen wir hier auch ein allerdings ziemlich kompliziertes Regelsystem für die Transformation der Constraints in einfachere Antwort-Constraints angeben. Dazu werden wir allerdings annehmen, daß die Constraints in einer Gleichungsnotation dargestellt sind. Constraints bestehen also aus normalen Termgleichungen und den u.U. relativ komplizierten Feature-Constraints. Letztere werden ebenfalls in Gleichungen transformiert. Dazu erlauben wir noch Featurezuweisungen $\varphi(x) = t$, wobei φ eine Featurefunktion ist und x in keiner rechten Seite - eventuell aber noch als Variable oder in anderen Featurezuweisungen auf den linken Seiten - vorkommt. Dann wird jeder Feature-Constraint $x:S[\varphi_1:=\Phi_1,\ldots, \varphi_n:=\Phi_n]$ übersetzt in $x:S$, $\varphi_1(x) = x_1$, $x_1:\Phi_1,\ldots, \varphi_n(x) = x_n$, $x_n:\Phi_n$, wobei die

$x_i:\Phi_i$ wieder Feature-Constraints mit neuen Variablen sind, so daß das Verfahren rekursiv angewendet werden muß. Der Antwort-Constraint im Beispiel sieht in dieser Notation dann so aus: x:Car, speed(x) = 50; also x ist ein Auto mit der Geschwindigkeit 50. Näheres dazu findet man etwa in [AS 87].

(T) $x = x \wedge \Gamma \rightarrow \Gamma$ (Tautologie)

(B) $x = t \wedge \Gamma \rightarrow x = t \wedge \{x \leftarrow t\}\Gamma$ (Bindung)

falls x eine Variable ist, die noch in Γ aber nicht in t vorkommt

und falls die Sorte von t kleiner als die Sorte von x ist.

(B') $x = y \wedge \Gamma \rightarrow x = z \wedge y = z \wedge \{x \leftarrow z, y \leftarrow z\}\Gamma$

falls die Sorte von x keine Untersorte der Sorte von y ist,

und falls die Sorten von x und y eine gemeinsame Untersorte besitzen,

und falls z eine *neue* Variable dieser gemeinsamen Untersorte ist.

(B'') $\varphi(x) = y \wedge \Gamma \rightarrow \varphi(x) = z \wedge y = z \wedge \{y \leftarrow z\}\Gamma$

falls die Sorte der Feature $\varphi(x)$ keine Untersorte der Sorte von y ist,

und falls beide Sorten eine gemeinsame Untersorte haben,

und falls z eine *neue* Variable dieser gemeinsamen Untersorte ist.

(M) $\varphi(x) = s \wedge \varphi(x) = t \wedge \Gamma \rightarrow \varphi(x) = s \wedge s = t \wedge \Gamma$ (Merging)

(D) $f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n) \wedge \Gamma \rightarrow s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge \Gamma$ (Dekomposition)

(U) $\varphi(s) = t \wedge \Gamma \rightarrow \varphi(x) = t \wedge x = s \wedge \Gamma$ (Unfolding)

falls x eine *neue* Variable der Sorte von s ist

(O) $t = x \wedge \Gamma \rightarrow x = t \wedge \Gamma$ (Orientierung)

(O') $t = \varphi(x) \wedge \Gamma \rightarrow \varphi(x) = t \wedge \Gamma$

Anmerkungen: 1. Features können also nie als Unterterme und auch nicht gleichzeitig auf beiden Seiten einer Gleichung auftreten.
2. Um genau zu sein, müßte die Sorte jeder Variablen angeben sein: $x = t \wedge x:S$ statt $x = t$, wenn x die Sorte S hat. Die Untersortenbeziehungen erhält man aus der Constraint-Theorie.

7 Schlußbemerkungen

In diesem Kapitel haben wir gesehen, daß und wie man Logik als Programmiersprache und Deduktion als Rechnen in dieser Sprache sehen kann. Die bekannteste Realisierung solcher logischen Programmiersprachen sind die verschiedenen PROLOG-Versionen. Natürlich fehlen bei unserer Darstellung hier noch die „außerlogischen“ Möglichkeiten die PROLOG zur Verfügung stellt und die es erst zu einer praktisch einsetzbaren, effizienten Programmiersprache werden lassen.

Stattdessen haben wir uns auf eine relativ knapp gehaltene Darstellung der logischen Grundlagen einer solchen Programmiersprache beschränkt – mehr dazu findet man in

zahlreichen PROLOG-Büchern; bezogen auf die logischen Grundlagen gibt insbesondere das Buch von J.W. Lloyd einen umfassenden Überblick [Llo 88]. Unser Anliegen war es, die Erweiterungsmöglichkeiten für logisches Programmieren, wie sie aus der theoretischen Forschung zum Automatischen Beweisen und Logischen Programmieren kommen und die in Büchern zum Logischen Programmieren bisher nicht zu finden sind, wieder zu geben: Sorten, Gleichheit und Constraints. Gänzlich ausgespart haben wir auch die Problematik der Verwendung von Negation in den Rümpfen der Regel- und Zielklauseln. Die unter der Bezeichnung „Negation as Failure“ bekannte Verwendung führt jedoch aus der in diesem Buch behandelten Sicht der Logik heraus, da diese Negation, wie sie auch in den meisten PROLOG-Systemen vorkommt, von der üblichen logischen Negation abweicht (vgl. ebenfalls [Llo 88] für die formalen Grundlagen von „Negation as Failure“). Eine gewisse Verwendung von Negation in Klauselrümpfen haben wir allerdings bei den Constrained-Klauseln wieder dabei, da hier zumindest für die Constraints Negation grundsätzlich möglich ist.

Mit der hier getroffenen Auswahl und Darstellung sollte eine Übersicht über die Verwendungsmöglichkeit der Logik als Programmiersprache etwa für die Inferenzkomponente eines wissensbasierten Systems gegeben werden. Die Darstellung sollte es den programmiererfahrenen Lesern ermöglichen, selbst ein solches auf Horn-Logik (mit den dargebotenen Erweiterungen) basierendes Inferenzsystem als Kern wissensbasierter Systeme zu implementieren – PROLOG ist aufgrund seiner „nicht-logischen“ Eigenschaften, aufgrund der Unvollständigkeit der implementierten Strategie und natürlich aufgrund seiner Inkorrektheit (fehlender ‚occur check‘) bekanntermaßen weniger als Inferenzsystem selbst geeignet als vielmehr eine Programmiersprache, die es allerdings erlaubt relativ leicht ein solches Inferenzsystem zu programmieren.

Literatur

AN 86 H. Ait-Kaci, R.Nasr: *LOGIN: A Logic Programming Language with Built-In Inheritance.* Journal of Logic Programming, No. 3, (1986)

AS 87 H. Ait-Kaci, G. Smolka: *Inheritance Hierarchies: Semantics and Unification.* MCC Technical Report, (1987)

BS 87 W. Büttner, H. Simonis: *Embedding Boolean Expressions into Logic Programming.* J. of Symbolic Computation, (1987)

Bür 91 H.-J. Bürckert: *A Resolution Principle for a Logic with Restricted Quantifiers*. Springer LNAI 568, (1991)

Cla 79 K.L. Clark: *Predicate Logic as a Computational Formalism.* Research Report, Imperial College, (1979)

Llo 88 J.W. Lloyd: *Foundations of Logic Programming.* 2nd edition, Springer, (1988)

Gal 86 J. Gallier: *Logic for Computer Science. Foundations of Automatic Theorem Proving.* Harper & Row, (1986)

GLLO 84 J. Gabriel, T. Lindholm, E.L. Lusk, R.A. Overbeek: *Tutorial on the Warren Abstract Prolog Machine.* Technical Report, Argonne National Laboratory, (1984)

GM 86 J.A. Goguen, J. Meseguer: *EQLOG: Equality, Types, and Generic Modules for Logic Programming*. In D. DeGroot and G. Lindstrom (Hrsg.): *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, (1986)

Hen 89 P. van Hentenryck: *Constraint Satisfaction in Logic Programming*. MIT Press, (1989)

HS 91 M. Höhfeld, G. Smolka: *Definite Relations over Constraint Languages*. Erscheint im J. of Logic Programming, (1991)

Höl 87 S. Hölldobler: *Equational Logic Programming*. Dissertation, Universität der Bundeswehr München, (1987)

Hay 73 P.J. Hayes: *Computation and Deduction.* Proc. MFCS Conference, Tschechoslowakische Akademie der Wissenschaften, (1973)

Hil 74 R. Hill: *LUSH-Resolution and its Completeness.* Technical Report, University of Edinburgh, (1974)

JL 87 J. Jaffar, J.-L. Lassez: *Constrained Logic Programming*. Proc. ACM Symposium on Principles of Programming Languages, (1987)

Kow 79 R.A. Kowalski: *Logic for Problem Solving.* North Holland, (1979)

Mah 87 M.J. Maher: *Logic Semantics for a Class of Committed-Choice Programs*. Proc. 4th Intern. Conf. on Logic Programming, (1987)

MN 87 U. Martin, T. Nipkow: *Unification in Boolean Rings.* J. of Automated Reasoning, (1987)

Rob 65 J.A. Robinson: *A Machine-Oriented Logic Based on the Resolution Principle.* JACM 12, (1965)

Sch 86 M. Schmidt-Schauß: *Unification in Many-Sorted Equational Theories.* Proc. of Conf. on Automated Deduction, Springer LNCS 230, (1986)

Shi 86 S.M. Shieber: *An Introduction to Unification-Based Approaches to Grammar.* Stanford University, CSLI Lecture Notes, (1986)

Sti 85 P. Stickel: *Automated Deduction by Theory Resolution*. J. of Automated Reasoning 1,4, (1985)

War 77 D.H.D. Warren: *Applied Logic - Its Use and Implementation as Programming Tool.* Ph.D. Thesis, University of Edinburgh, (1977)

War 83 D.H.D. Warren: *An Abstract Prolog Instruction Set.* Technical Report, SRI International, (1983)