



HOCHSCHULE **TRIER**

Trier University of Applied Sciences

Informatik - Computer Science

Konzeption und Implementierung eines Schlussfolgerungssystems für Hornlogik mit einer funktionalen Programmiersprache

Steven Kutsch

Master-Abschlussarbeit

Betreuer: Prof. Dr. Karl Hans Bläsius

Trier, 27. August 2015

Kurzfassung

Entwickelt wurde ein Schlussfolgerungssystem für Hornlogik basierend auf der SLD-Resolution. Logikprogramme werden aus einer Textdatei geparsst und in eine interne Repräsentation überführt. Auf Basis dieser internen Repräsentation wurden die Unifikation von Literalen sowie ein Resolutionsverfahren nach SLD-Strategie implementiert. Das System wurde in Lisp implementiert. Die Arbeit beschreibt nötige Grundlagen der Logik sowie das Vorgehen bei Entwurf und Implementierung.

Inhaltsverzeichnis

1	Einleitung	1
2	Logische Grundbegriffe	2
2.1	Prädikatenlogik	2
2.1.1	Syntax	2
2.1.2	Semantik	5
2.1.3	Resolutionskalkül	7
2.2	Resolutionsstrategien	9
2.2.1	lineare Resolution	9
2.2.2	Set of Support Strategie	9
2.2.3	SLD-Resolution	10
3	Prolog	11
3.1	Syntax	11
3.2	Semantik	12
3.2.1	Unvollständigkeit von Prolog	12
4	Konzeption	14
4.1	Beschreibungssprache	14
4.1.1	Lexikalische Analyse	14
4.1.2	Grammatik für Programme	15
4.1.3	Grammatik für Terme	16
4.2	Semantik	16
4.3	Algorithmen	17
4.3.1	Unifikationsalgorithmus	17
4.3.2	SLD-Resolutionsalgorithmus	18
5	Implementierung	22
5.1	Lexing und Parsing	22
5.1.1	Parser für Programme	22
5.1.2	Parser für Terme	24
5.2	Interpreter	25
5.2.1	Listen	26
5.3	Logisches Schlussfolgern	26

6 Ausblick	28
7 Zusammenfassung	29
Literaturverzeichnis	30
Erklärung des Kandidaten	31



1

Einleitung

Schlussfolgerungssysteme entscheiden, ob eine Aussage aus einer Menge von Axiomen folgt. Sie stellen damit eine unmittelbare Anwendung der formalen Logik dar und sind der Kern logischer Programmiersprachen. Als Inferenzmaschinen finden sie Anwendung in der künstlichen Intelligenz.

Aufgrund ihrer Relevanz für praktische Anwendungen sind Schlussfolgerungssysteme in Form der logischen Programmiersprache Prolog Teil der Lehrveranstaltung Angewandte Logik. Den Studenten sollen Grundlagen der logischen und symbolischen Programmierung sowie das Prinzip der Rekursion näher gebracht werden. Nicht logische Bestandteile von Prolog sind nicht Inhalt der Lehrveranstaltung.

Als Alternative für Prolog sollte ein Schlussfolgerungssystem entwickelt werden, das den Fokus auf logische und symbolische Programmierung anstatt auf praktische Relevanz legt. Es sollte einfacher sein, den Zusammenhang zwischen Formeln und Logik-Programmen und so die Relevanz der theoretischen Inhalte der Lehrveranstaltung zu erkennen.

Zu diesem Zweck soll in dieser Arbeit die formale Sprache CL-Reason entwickelt werden, die Nähe zur klassischen Darstellung von Formeln mit dem syntaktischen Komfort von Prolog verbindet. Für diese Sprache wird ein Parser entwickelt, mit dessen Hilfe es möglich ist, Formeln in eine interne Repräsentation zu überführen. Auf dieser internen Repräsentation der Formeln ist nun das automatisierte Durchführen von Schlussfolgerungen gemäß der in der Lehrveranstaltung vorgestellten Techniken möglich.

Die Arbeit behandelt Grundlagen der formalen Logik (Kapitel 2) und für Konzeption und Implementierung wichtige Erkenntnisse aus dem logischen Kern der Programmiersprache Prolog (Kapitel 3). Schließlich wird das Vorgehen bei der Konzeption (Kapitel 4) und der Implementierung (Kapitel 5) beschrieben. Ein Ausblick beschreibt denkbare und sinnvolle Erweiterungen des entwickelten Systems (Kapitel 6).

2

Logische Grundbegriffe

In diesem Kapitel sollen die für spätere Ausführungen benötigten Grundbegriffe der formalen Logik (insbesondere der Prädikatenlogik) eingeführt werden. Dabei soll auf Syntax und Semantik der Prädikatenlogik, Hornklauseln, Unifikation und Resolutionsverfahren sowie -strategien eingegangen werden.

2.1 Prädikatenlogik

Die Prädikatenlogik ist eine Erweiterung der Aussagenlogik, bei der Literale nicht als Aussagen mit einem fixen Wahrheitswert interpretiert werden, sondern als Prädikate, deren Wahrheitswert von ihren Argumenten abhängt. In diesem Abschnitt werden Syntax und Semantik der Prädikatenlogik eingeführt. Die hier verwendete Darstellung von Formeln ist umfangreicher in [Bec88] zu finden.

2.1.1 Syntax

Formeln der Prädikatenlogik sind Wörter über dem Alphabet

$$\Sigma = V \cup F \cup P \cup J \cup Q \cup I \quad (2.1)$$

wobei

- $V = \{x_1, \dots, x_n\}$ (Variablen symbole),
- F eine Menge von n-stelligen Funktionssymbolen (einschließlich von 0-stelligen Funktionen, den sog. Konstanten),
- P eine Menge von n-stelligen Prädikatssymbolen,
- $J = \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ (Junkturen),
- $Q = \{\exists, \forall\}$ (Quantoren),
- $I = \{\cdot, \cdot, \cdot, \cdot, \cdot\}$ (Interpunktionen)

sind.

Prädikatesymbole und Funktionssymbole bilden die Signatur einer prädikatenlogischen Sprache. Sie sind von der Anwendung abhängig und müssen disjunkt sein. Alle Funktions- oder Prädikatsymbole müssen von den in den anderen Teilmengen vorkommenden Symbolen verschieden sein.

Aus Variablen, Konstanten und Funktionssymbolen lassen sich Terme bilden.

Definition 2.1 (Terme).

(IA) Variablen und Konstantensymbole sind Terme

(IS) Sind t_1, \dots, t_n Terme und f ein Funktionssymbol, ist auch $f(t_1, \dots, t_n)$ ein Term.

Aus Termen und einem Prädikatsymbol lassen sich atomare Formeln bilden.

Definition 2.2 (atomare Formel).

Seien t_1, \dots, t_n Terme und P ein Prädikatssymbol. Dann ist $P(t_1, \dots, t_n)$ eine atomare Formel. Wir nennen atomare Formeln auch kurz Atome

Durch das Verknüpfen von atomaren Formeln mittels Junktoren und das Quantifizieren von Variablen lassen sich zusammengesetzte Formeln bilden.

Definition 2.3 (Formeln).

(IA) Atomare Formeln sind Formeln

(IS) Sind F und G Formeln und x_1, \dots, x_n Variablen, dann sind auch

- $\neg F$ (Negation),
 - $F \wedge G$ (Konjunktion),
 - $F \vee G$ (Disjunktion),
 - $F \Rightarrow G$ (Implikation),
 - $F \Leftrightarrow G$ (Äquivalenz),
 - $\forall x_1, \dots, x_n : F$ und
 - $\exists x_1, \dots, x_n : F$
- Formeln.

Mit diesen drei Definitionen lassen sich alle Formeln der Prädikatenlogik bilden. Der Begriff des Literals erlaubt uns für die folgenden Ausführungen eine kompaktere Darstellung von Formeln.

Definition 2.4 (Literal).

Ein Literal ist eine atomare Formel (positives Literal) oder eine negierte atomare Formel (negatives Literal).

Das in dieser Arbeit entwickelte Schlussfolgerungssystem arbeitet auf einer bestimmten Teilmenge prädikatenlogischer Formeln, die als Hornklauseln bezeichnet werden. Wir betrachten zunächst die allgemeine Darstellung prädikatenlogischer Formeln als Menge von Klauseln.

Definition 2.5 (Konjunktive und Klauselnormalfom).

Eine Formel ist in konjunktiver Normalform, wenn sie nur aus einer Konjunktion von Disjunktionen von Literalen besteht.

Sind L_{11}, \dots, L_{mn} Literale (für $m, n \geq 1$) ist die Formel

$$((L_{11} \vee \dots \vee L_{1n}) \wedge \dots \wedge (L_{m1} \vee \dots \vee L_{mn})) \quad (2.2)$$

in konjunktiver Normalform.

Die Klauselnormalfom ist eine Mengendarstellung einer Formel in konjunktiver Normalform.

$$\{\{L_{11}, \dots, L_{1n}\}, \dots, \{L_{m1}, \dots, L_{mn}\}\} \quad (2.3)$$

Ein Element der Klauselnormalfom nennen wir Klausel.

Jede prädikatenlogische Formel lässt sich in eine äquivalente Formel in Klauselnormalfom überführen.

Definition 2.6 (Hornklausel).

Eine Hornklausel ist eine Klausel die höchstens ein positives Literal enthält.

Hornklauseln lassen sich auf zwei verschiedene Arten bequem als prädikatenlogische Formeln darstellen.

Definition 2.7 (Fakten- und Regelform).

Eine Formel ist in Faktenform, wenn sie lediglich aus einem positiven Literal besteht.

$$P(x) \quad (2.4)$$

Eine Formel ist in Regelform, wenn sie aus einer Implikation zwischen einer Konjunktion und einem positiven Literal besteht

$$Q_1(x) \wedge \dots \wedge Q_n(x) \Rightarrow P(x) \quad (2.5)$$

Um Sinnvoll über die Bedeutung einer Formel sprechen zu können, müssen zwei Klassen von Variablen innerhalb einer Formel unterschieden werden.

Definition 2.8 (freie und gebundene Variablen).

Wird eine Variable x in einer Formel F durch einen Quantor (\forall, \exists) gebunden, heißt x in f **gebundene Variable**.

Alle Variablen die in einer Formel nicht gebunden sind, heißen **freie Variablen**.

Eine Formel die keine freien Variablen enthält, nennen wir **geschlossene Formel**.

Wir betrachten von nun an nur noch geschlossene Formeln und gehen davon aus, dass alle Variablen durch einen Allquantor gebunden sind. Man spricht dann vom universellen Abschluss einer Formel.

2.1.2 Semantik

Um einer prädikatenlogischen Sprache eine Semantik zuzuordnen, muss der Definitionsbereich (das sog. Universum) bekannt sein und den Symbolen der Signatur müssen Elemente über diesem Universum zugeordnet werden.

Definition 2.9 (Interpretation).

Eine Interpretation $\mathcal{I} = (\mathcal{U}, \mathcal{F}, \mathcal{P}, \mathcal{V})$ besteht aus

- dem Universum \mathcal{U} ,
- einer Funktion \mathcal{F} , die jedem Funktionssymbol eine Funktion über \mathcal{U} zuordnet,
- einer Funktion \mathcal{P} , die jedem Prädikatsymbol eine Relation über \mathcal{U} zuordnet und
- einer Funktion \mathcal{V} , die jeder Variable ein Element aus \mathcal{U} zuordnet.

Die Funktion \mathcal{V} lässt sich auf Funktionsterme ausweiten. Die Bewertung von Termen ergibt sich dann aus der Instantierung von Funktions- und Variablensymbolen.

$$\mathcal{V}(f(t_1, \dots, t_n)) = \mathcal{F}(f)(\mathcal{V}(t_1), \dots, \mathcal{V}(t_n)) \quad (2.6)$$

Daraus ergibt sich direkt der Wahrheitswert atomarer Formeln.

$$R(t_1, \dots, t_n) \text{ ist wahr gdw. } (\mathcal{V}(t_1), \dots, \mathcal{V}(t_n)) \in \mathcal{P}(R) \quad (2.7)$$

Der Wahrheitswert mittels Junktoren zusammengesetzter Formeln ergibt sich auch der üblichen Interpretation der Junktoren.

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \Rightarrow G$	$F \Leftrightarrow G$
falsch	falsch	wahr	falsch	falsch	wahr	wahr
falsch	wahr	wahr	falsch	wahr	wahr	falsch
wahr	falsch	falsch	falsch	wahr	falsch	falsch
wahr	wahr	falsch	wahr	wahr	wahr	wahr

Der Wahrheitswert quantifizierter Formeln ergibt sich aus der Bedeutung der Quantoren.

Definition 2.10 (All- und Existenzquantoren).

Eine Formel der Form $\forall x : F$ ist wahr gdw. alle Variablenbelegungen, die sich höchstens in der Belegung von x unterscheiden, F wahr machen.

Eine Formel der Form $\exists x : F$ ist wahr gdw. $\neg\forall x : \neg F$ wahr ist.

Ist eine Formel F unter der Interpretation \mathcal{I} wahr, schreiben wir

$$\mathcal{I} \models F \quad (2.8)$$

Wir nennen \mathcal{I} auch Modell der Formel F .

Über den Wahrheitswert einer Formel unter einer Interpretation lassen sich Erfüllbarkeit und Folgerung definieren.

Definition 2.11 (Erfüllbarkeit).

Eine Formel F heißt erfüllbar gdw. eine Interpretation \mathcal{I} existiert, sodass

$$\mathcal{I} \models F \quad (2.9)$$

Existiert keine solche Interpretation, heißt F unerfüllbar.

Definition 2.12 (Folgerung).

Sei Γ eine Menge geschlossener Formeln und F eine geschlossene Formel. Wir bezeichnen F als Folgerung von Γ falls für jede Interpretation \mathcal{I} gilt

$$\text{falls } \mathcal{I} \models \Gamma, \text{ dann } \mathcal{I} \models F \quad (2.10)$$

Wir schreiben dann,

$$\Gamma \models F \quad (2.11)$$

Eine für diese Arbeit wichtige Konsequenz ist, dass für eine Menge geschlossener Formeln Γ und eine geschlossene Formel F gilt, dass

$$\Gamma \cup \{\neg F\} \text{ ist unerfüllbar gdw. } \Gamma \models F \quad (2.12)$$

Aufgabe eines Schlussfolgerungssystems ist es nun für eine gegebene Menge an Formel (die Axiome) und eine weitere Formel (die Behauptung) zu entscheiden, ob die Behauptung aus den Axiomen folgt. Idealerweise liefert das System dabei auch die in der Interpretation verwendete Variablenbelegung.

Da dies direkt praktisch nicht automatisierbar ist, wird ein operationelles Verfahren, ein sogenannter Kalkül benötigt. Ein Kalkül besteht aus einer Menge von Axiomen und Schlussregeln durch die es schrittweise möglich ist, die Folgerungsbeziehung zwischen einer Menge von Formeln und einer Behauptung zu beweisen.

2.1.3 Resolutionskalkül

Der Resolutionskalkül liefert ein formales Verfahren, um eine solche Folgerungsbeziehung zu beweisen. Die Resolutionsregel basiert auf dem Instanzieren von Variablen mit konkreteren Termen wie Konstanten oder Funktionen. Das Instanzieren kann als Anwendung einer Substitution betrachtet werden.

Definition 2.13 (Substitution auf Termen).

Sei T die Menge aller Terme und V die Menge aller Variablen. Eine Substitution ist eine Abbildung $\sigma : T \rightarrow T$ sodass,

1. $\sigma(c) = c$ für alle Konstanten
2. $\sigma(f(t_1, \dots, t_n)) = f(\sigma(t_1), \dots, \sigma(t_n))$
3. $\{x \in V | \sigma(x) \neq x\}$ ist endlich

Eine Substitution heißt Idempotent wenn $\sigma(t) = \sigma(\sigma(t))$

Idempotente Substitutionen können auch als Menge von Paaren dargestellt werden

$$\{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\} \quad (2.13)$$

Wenn Mehrdeutigkeiten ausgeschlossen sind, schreiben wir statt $\sigma(T)$ auch einfacher σT

Die Substitutionsabbildung lässt sich wie folgt auf Formeln erweitern.

$$\sigma P(t_1, \dots, t_n) = P(\sigma t_1, \dots, \sigma t_n) \quad (2.14)$$

$$\sigma \neg P(t_1, \dots, t_n) = \neg P(\sigma t_1, \dots, \sigma t_n) \quad (2.15)$$

$$\sigma \{L_1, \dots, L_n\} = \{\sigma L_1, \dots, \sigma L_n\} \quad (2.16)$$

Unifikation beschreibt die systematische Vereinheitlichung von zwei Atomen durch anwenden von Substitutionen. Zwei Atome $P(s_1, \dots, s_n)$ und $Q(t_1, \dots, t_n)$ sind dabei unifizierbar, wenn $P = Q$ und die beiden Termlisten (s_1, \dots, s_n) und (t_1, \dots, t_n) unifizierbar sind.

Definition 2.14 (Unifikator).

Eine Substitution σ heißt Unifikator zweier Terme s und t , wenn $\sigma s = \sigma t$.

Ein Unifikator σ heißt allgemeinster Unifikator (mgu = most general unifier), falls es für jeden weiteren Unifikator τ eine Substitution λ gibt, sodass $\tau = \lambda\sigma$

Die Resolutionsregel beschreibt, wie aus zwei Klauseln einer Klauselmenge eine neue Klausel abgeleitet werden kann. Sie basiert auf dem Modus Ponens.

$$\frac{F \quad F \Rightarrow G}{G} \quad (2.17)$$

Gilt sowohl die Formel F als auch, das $F \Rightarrow G$ impliziert, so gilt auch G .

Die Resolutionsregel ist eine Verallgemeinerung des Modus Ponens. Sei σ ein mgu für F und F' . Dann gilt

$$\frac{K_1 = \{F, L_1, \dots, L_n\} \quad K_2 = \{\neg F', G_1, \dots, G_m\}}{R = \sigma \{L_1, \dots, L_n, G_1, \dots, G_m\}} \quad (2.18)$$

Wir nennen die Klauseln K_1 und K_2 Elternklauseln der Resolvente R .

Satz 1 *Die Resolutionsregel ist korrekt.*

Beweis:

Sei \mathcal{I} eine Interpretation unter der $(F \vee L_1 \vee \dots \vee L_n) \wedge (\neg F' \vee G_1 \vee \dots \vee G_m)$ wahr ist und σ ein Unifikator für F und F' . Dann sind

- $(\sigma F \vee \sigma L_1 \vee \dots \vee \sigma L_n)$ und
- $(\neg \sigma F' \vee \sigma G_1 \vee \dots \vee \sigma G_m)$

unter \mathcal{I} wahr. σF ist unter \mathcal{I} entweder wahr oder falsch.

Falls σF unter \mathcal{I} wahr ist, ist $\neg \sigma F'$ unter \mathcal{I} falsch (da $\sigma F = \sigma F'$). Dann ist $(G_1 \vee \dots \vee G_m)$ unter \mathcal{I} wahr. Also ist auch $(G_1 \vee \dots \vee G_m \vee L_1 \vee \dots \vee L_n)$ unter \mathcal{I} wahr.

Falls σF unter \mathcal{I} falsch ist, ist $(L_1 \vee \dots \vee L_n)$ unter \mathcal{I} wahr. Also ist auch $(G_1 \vee \dots \vee G_m \vee L_1 \vee \dots \vee L_n)$ unter \mathcal{I} wahr. \square

Durch Anwendung der Resolutionsregel auf zwei Klauseln der Form

$$\{F\} \quad \{\neg F'\} \quad (2.19)$$

mittels eines Unifikators für F und F' ist die Resolvente leer. Die leere Klausel lässt sich nur aus zwei widersprüchlichen Klauseln, die unter keiner Interpretation den gleichen Wahrheitswert haben können, bilden.

Definition 2.15 (Ableitbarkeit).

Kann durch wiederholte Anwendung der Resolutionsregel auf Klauseln K_1, \dots, K_n eine Klausel K gebildet werden, ist K aus K_1, \dots, K_n ableitbar. Wir schreiben dann

$$K_1, \dots, K_n \vdash K \quad (2.20)$$

Lässt sich durch wiederholte Anwendung der Resolutionsregel auf Klauseln einer Klauselmenge die leere Klausel ableiten, entspricht dies einem Widerspruch und beweist die Unerfüllbarkeit der initialen Klauselmenge.

Ist nun eine Menge von geschlossenen Formeln Γ und eine geschlossene Formel F gegeben, beweist eine Resolutionsbeweis, der die Unerfüllbarkeit der Klauselmenge $\Gamma \cup \{\neg F\}$ beweist, die Folgerung von F aus Γ

Zusammen mit der Faktorisierungsregel, die hier nicht näher betrachtet wird, bildet die Resolutionsregel den Resolutionskalkül.

Satz 2 (Widerlegungsvollständigkeit) Der Resolutionskalkül ist widerlegungsvollständig, d. h. für eine Menge geschlossener Formeln Γ und eine Behauptung F gilt

$$\Gamma \models F \text{ gdw. } \Gamma \cup \{\neg F\} \vdash \{\} \quad (2.21)$$

2.2 Resolutionsstrategien

Aufgrund der Widerlegungsvollständigkeit eignet sich der Resolutionskalkül als Widerlegungsverfahren für die Prädikatenlogik. Die Resolutionsregel gibt allerdings nur an, wie aus zwei Klauseln eine neue Klausel abgeleitet werden kann. Die Frage, welche zwei Klauseln für einen Resolutionsschritt herangezogen werden sollen, ist entscheidend für eine effektive und effiziente Ableitung der leeren Klausel. Resolutionstrategien liefern eine Antwort auf diese Frage.

Für Mengen von Hornklauseln existiert mit der SLD-Resolution eine korrekte und widerlegungsvollständige Resolutionsstrategie. Im Folgenden werden die grundlegenden Strategien für die SLD-Resolution vorgestellt.

2.2.1 lineare Resolution

Bei linearen Resolutionsstrategien ist eine der beiden Elternklauseln immer die Resolvente aus dem vorherigen Resolutionsschritt. Die zweite Elternklausel ist frei zu wählen.

Ist zum Beispiel folgende Klauselmenge gegeben

$$\{\{P(x, y), Q(x)\}^1, \{\neg P(f(a), y), R(b)\}^2, \{\neg Q(f(a))\}^3, \{\neg R(x)\}^4\} \quad (2.22)$$

wobei x und y Variablen und a und b Konstanten sind, ist

$\{Q(f(a)), R(b)\}^5$	Resolvente aus Klausel 1 und 2; mgu = $\{x \leftarrow f(a)\}$
$\{R(b)\}^6$	Resolvente aus Klausel 5 und 3; mgu = $\{\}$
$\{\}$	Resolvente aus Klausel 6 und 4; mgu = $\{x \leftarrow b\}$

ein linearer Resolutionsbeweis.

2.2.2 Set of Support Strategie

Zur Einschränkung des Suchraums wird bei Verwendung der Set of Support Strategie eine Teilmenge T der initialen Klauselmenge K gewählt, sodass $K \setminus T$ erfüllbar ist. Im Resolutionsbeweis relativ zur Stützmenge T , dürfen dann nie zwei Klauseln aus $K \setminus T$ miteinander resolviert werden.

Legt man fest, dass die Menge T nur die negierte Behauptung enthält, werden nie zwei Axiome der initialen Klauselmenge miteinander resolviert.

2.2.3 SLD-Resolution

Die SLD-Resolution (SLD = linear resolution with selection function for definite clauses) ist eine lineare Set of Support Resolution. Dabei wird zusätzlich die Einschränkung getroffen, dass in jedem Schritt eine Klausel der initialen Klauselmengen an der Resolution beteiligt ist. Diese Einschränkung wird als Input Restriktion bezeichnet. In einem SLD Resolutionsbeweis wird also, beginnend mit der initialen Zielklausel, eine Zielklausel mit einem Axiom der initialen Klauselmengen resolviert.

Der Suchraum einer SLD-Resolution entspricht einem Baum, dessen Knoten Zielklauseln sind. Die Nachfolger eines Knotens sind Zielklauseln, die durch einen Resolutionsschritt mit einem Axiom der initialen Klauselmengen entstehen.

Sei folgende Hornklauselmenge gegeben.

$$\{ \{vater(paul, klaus)\}^1, \{vater(klaus, steven)\}^2, \{vater(paul, peter)\}^3, \\ \{\text{großvater}(x, y), \neg vater(x, z), \neg vater(z, y)\}^4, \{\neg \text{großvater}(x, steven)\}^5 \}$$

Ist Klausel fünf die initiale Zielklausel, ergibt sich der SLD-Baum in Abbildung 2.1.

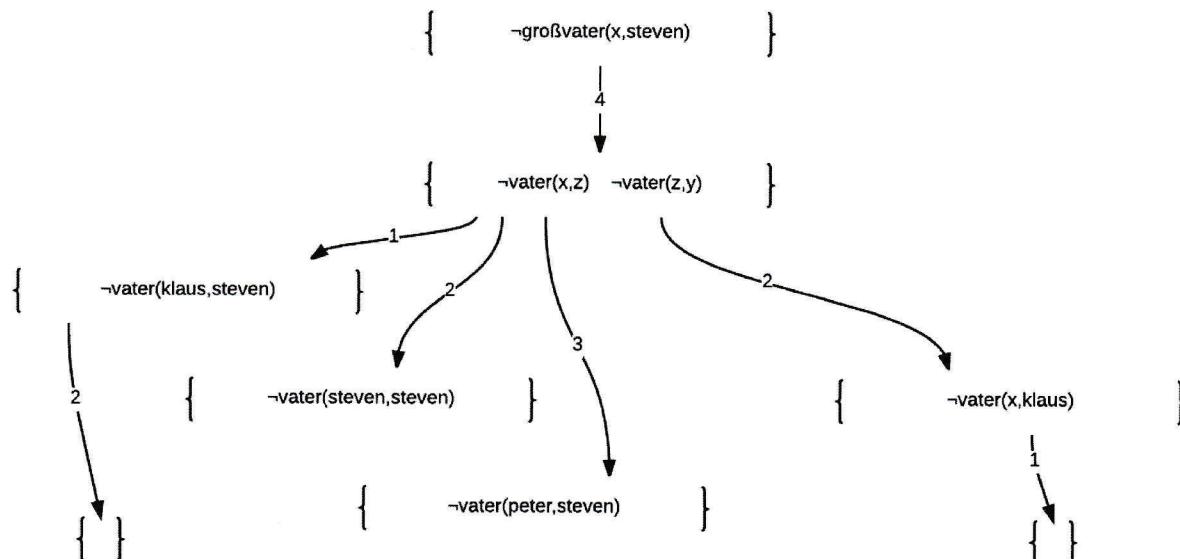


Abb. 2.1. SLD-Baum

Ein Resolutionsbeweis nach SLD-Strategie ist ein Wurzelpfad im SLD-Baum. Die SLD-Resolution ist widerlegungsvollständig für Mengen von Hornklauseln.

3

Prolog

Die derzeit wichtigste logische Programmiersprache ist Prolog [CR96]. Der rein logische Teil der Sprache besteht aus einem Schlussfolgerungssystem basierend auf der SLD-Resolution. Hinzu kommt ein nicht-logischer Teil, der es ermöglicht praktische Anwendungen rein in Prolog zu implementieren. Im folgenden wird der rein logische Kern von Prolog vorgestellt.

3.1 Syntax

Prologs Syntax entspricht weitgehend der Syntax der Prädikatenlogik, wobei nur Hornklauseln in Fakten- und Regelform zulässig sind.

Die Syntax für Terme entspricht Definition 2.1. Zwischen Variablen und Konstanten wird dabei durch Groß- und Kleinschreibung unterschieden. Bezeichner die mit einem Großbuchstaben beginnen sind Variablen, alle anderen Bezeichner werden als Konstanten interpretiert.

Listen stellen eine Erweiterung der Terme wie sie in Kapitel 2 vorgestellt werden dar.

Definition 3.1 (Prologlisten).

(IA₁) [] ist eine Liste (leere Liste)

(IA₂) Sind t_1 bis t_n Terme, dann ist $[t_1, \dots, t_n]$ eine Liste.

(IS) Wenn L eine Liste ist und X ein Term, dann ist $[X|L]$ eine Liste.

Mit den Operatoren :- (entspricht \Leftarrow) und , (entspricht \vee) können nun aus Prädikaten Klauseln gebildet werden. Ein Punkt (.) schließt eine Klausel ab.

Ein Beispiel für ein einfaches Prolog-Programm ist **alle-kleiner**.

```
alle-kleiner([], N).  
alle-kleiner([F|R], N) :- F < N, alle-kleiner(R, N).
```

3.2 Semantik

Das Programm `alle-kleiner` beschreibt eine Relation zwischen Listen von Zahlen und einer Zahl n , die jeweils erfüllt ist, wenn alle Elemente der Liste kleiner sind als n . Für das Schlussfolgerungssystem ist dieses Programm nun eine Menge geschlossener Formeln (durch universellen Abschluss). Stellt man nun die Anfrage

```
alle-kleiner([1,3,5],6).
```

führt Prolog eine SLD-Resolution auf der Klauselmenge

$$\{ \{alle - kleiner(empty, x)\}^1, \\ \{alle - kleiner(cons(f, r), x), \neg < (f, x), \neg alle - kleiner(r, x)\}^2, \\ \{\neg alle - kleiner(cons(1, cons(3, cons(5, empty))), x)\}^3 \}$$

durch. Die ersten beiden Klauseln ergeben sich aus dem Programm und werden als Stützmenge gewählt. Die dritte Klausel ergibt sich aus der negierten Behauptung und steht bereits als Elternklausel für den ersten Resolutionsschritt fest. Nun wird gemäß der Selektionsfunktion eine passende Klausel zur Resolution mit der Zielklausel gewählt.

Die in Prolog verwendete Selektionsfunktion liefert dabei die gemäß der Reihenfolge im Programm erste Klausel, deren positives Literal sich mit dem ersten Literal der Zielklausel unifizieren lässt.

In der Resolvente stehen die neuen Teilziele aus der Programmklause stets vor den Teilzielen aus der Zielklausel. Kann das erste Teilziel der Zielklausel nicht mit einem positiven Literal einer Programmklause unifiziert werden, wird ein Backtracking eingeleitet. Dazu wird die aktuelle Zielklausel verworfen und damit auch alle Variablenbelegungen aus der letzten Unifikation zurückgenommen. Dann wird nach einer alternativen Unifikationsmöglichkeit für das erste Teilziel gesucht.

Ist das nächste Teilziel ein vordefiniertes Prädikat (wie z. B. $<$), wird nicht nach einer passenden Zielklausel gesucht sondern eine Sonderbehandlung angestoßen, die eine neue Zielklausel liefert oder bei Nichterfüllung des Prädikats Backtracking auslöst.

Die Suchstrategie von Prolog entspricht aufgrund der Reihenfolge der Teilziel-auswertung einer Tiefensuche im SLD-Baum.

3.2.1 Unvollständigkeit von Prolog

Die Resolution nach der SLD-Strategie ist ein nichtdeterministisches Verfahren. In jedem Schritt liegen unter Umständen mehrere Ableitungsmöglichkeiten abhängig von

- der gewählten Programmklause und
- des gewählten Teilziels

vor. Die in Prolog gewählte Selektionsfunktion und die forcierte Reihenfolge der Teilziele in der Zielklausel stellen zwar Determinismus sicher, sind aber auch Grund für die größte Schwäche von Prolog: die Unvollständigkeit des Schlussfolgerungssystems. [Bec88]

Jedes erfolgreiche Blatt im SLD-Baum, das hinter einem unendlichen Wurzelpfad liegt, ist effektiv unerreichbar. Wir betrachten folgendes Prolog Programm aus [Llo93]:

```
p(a,b).  
p(c,b).  
p(X,Z) :- p(X,Y), p(Y,Z). -- transitivität  
p(X,Y) :- p(Y,X). -- kommutativität
```

Theoretisch sollte das Schlussfolgerungssystem die Anfrage

```
p(a,c).
```

bejahen. Aufgrund der Selektionsfunktion wird Prolog jedoch stets eine Unifikation mit der dritten Klausel versuchen, da das positive Literal $p(X,Z)$ mit jeder Anfrage zum Prädikat p unifizierbar ist. Für eine erfolgreiche Ableitung der leeren Klausel wird jedoch auch die vierte Klausel benötigt. Das Resultat ist ein Stackoverflow während der Lösungsfindung.

Die Lösung dieses Problems war nicht teil der Aufgabenstellung. Das in dieser Arbeit entwickelte Schlussfolgerungssystem leidet also unter derselben Schwäche.

4

Konzeption

In diesem Kapitel werden die entwickelten Grammatiken zur Beschreibung der Eingabesprache, die gewählte Methode zur Antwortbestimmung sowie die Algorithmen die das Schlussfolgerungssystem bilden beschrieben.

4.1 Beschreibungssprache

Das System verarbeitet Formeln definiert in einer formalen Sprache.

Zur Beschreibung der Sprache wurden zwei kontextfreie Grammatiken definiert, die jeweils eine Ebene der Sprache beschreiben. Auf der obersten Ebene werden Programme als Folge von Hornklauseln in Regel- oder Faktendarstellung betrachtet. Prädikate samt ihrer Argumente werden dabei als ein Eingabesymbol betrachtet. Dies wird durch eine vorherige lexikalische Analyse ermöglicht, die den Eingabestring in Tokens zerlegt, und dabei die Argumente von Prädikaten als Property in den erzeugten Literal-Symbolen speichert.

Auf der unteren Ebene werden Terme wie sie als Argumente von Prädikaten auftauchen geparst.

4.1.1 Lexikalische Analyse

Die Eingabe in den Lexer ist das vom Benutzer verfasste Programm als String. Mittels eines regulären Ausdrucks wird dieser String in die einzelnen Bausteine (sog. Tokens) des Programms zerlegt.

Definition 4.1 (CL-Reason Tokens).

Die in der lexikalischen Analyse zu identifizierenden Token sind Worte der Sprache

$$\{a, \dots, z\} \cup \{\text{forall}, \text{exists}, \text{and}, \leq, \geq, :, .\} \cup \\ \{w(t) \mid w \in \{a, \dots, z, A, \dots, Z, 0, \dots, 9\}, t \in \{a, \dots, z, 0, \dots, 9, (,), ,\}^*\}$$

Es ist anzumerken, dass die zu matchende Sprache nicht regulär ist. Moderne Regex-Engiens erlauben jedoch das verwenden von Lookaheads, wodurch eine Er-

kennung von korrekt geklammerten Prädikatsdeklarationen möglich ist. Die in Abschnitt 4.1.2 definierte Grammatik für Programme legt fest, dass auf ein Prädikat entweder eine Implikation, eine Konjunktion oder das Ende der Klausel folgen muss. Aufgrund dieser Tatsache lässt dich mittels einer Lookaheads entscheiden, ob eine schließende Klammer tatsächlich ein Prädikat abschließt, oder noch zur Liste der Terme innerhalb des Prädikats gehört.

4.1.2 Grammatik für Programme

Auf der obersten Ebene besteht ein Programm aus einer Folge von Hornklauseln in Regel- oder Faktendarstellung.

Definition 4.2 ($G_{cl-reason}$).

Die kontextfreie Grammatik $G_{cl-reason} = (\Sigma, N, P, S)$ mit

- $\Sigma = \{forall, exists, and, in, end, var, lit, <=, =>\}$,
- $N = \{S, V, K, F, F'\}$ und

$P : S \rightarrow forall VKS \mid exists VKS \mid lit end S \mid \epsilon$

$V \rightarrow var V \mid var in$

$K \rightarrow lit <= F \mid lit and F' \mid lit end$

$F \rightarrow lit and F \mid lit end$

$F' \rightarrow lit and F \mid lit => lit end$

erzeugt alle gültigen Programme der Sprache CL-Reason

Das Alphabet Σ enthält alle Token die in der vom Lexer erzeugten Ausgabe enthalten sein können. Ausgehend vom Startsymbol S lässt sich eine beliebig lange Folge von Klauseln bestehend aus einer Variablen Deklaration mit *forall* oder *exists* gefolgt von beliebig vielen Variablen und der eigentlichen Klausel in Fakt- oder Regeldarstellung erzeugen.

Mit dem Nichtterminal V lässt sich eine beliebig lange Folge von Variablen erzeugen.

Das Nichtterminal K leitet die Erzeugung einer Klausel in Regeldarstellung ein. Da sowohl links- als auch rechtsgerichtete Implikationen erlaubt sind, muss über die Unterscheidung zwischen den Nichtterminalen F und F' sichergestellt werden, dass Programme der Form

`forall x: p(x) <= q(x) and s(x) => r(x).`

nicht erzeugt werden können.

Die Frage nach der syntaktischen Korrektheit eines gegebenen Programms kann durch einen $LL(2)$ Parser beantwortet werden. Bei der Expansion eines Nichtterminal kann mit einem Lookahead von zwei Terminalsymbolen die zu wählende Produktion eindeutig bestimmt werden. Der Aufbau des entwickelten Parser ist im Detail im Abschnitt 5.1.1 beschrieben.

4.1.3 Grammatik für Terme

Ein Teil der Grammatik für Terme ergibt sich direkt aus der induktiven Definition von Termen aus Definition 2.1. In CL-Reason wird für die Darstellung von Listen eine spezielle Syntax verwendet.

Definition 4.3 (Listen).

(IA) Die Konstante *empty* ist eine Liste (leere Liste).

(IS) Wenn l eine Liste und x eine Term ist, dann ist auch $\text{cons}(x, l)$ eine Liste.

Mit dieser induktiven Definition lassen sich beliebige Listenstrukturen als Terme gemäß von Definition 2.1 darstellen. Da diese Darstellung zur Formulierung von geschachtelten Listen unhandlich ist, können in CL-Reason Listen in der aus Prolog bekannten Syntax formuliert werden (Definition 3.1).

Arithmetische Ausdrücke lassen sich in Infix-Notation formulieren. Dabei können die Operatoren + (Addition), - (Subtraktion), * (Multiplikation) und / (Division) verwendet werden. Des weiteren können arithmetische Ausdrücke beliebig geklammert werden.

Definition 4.4 (G_{Terme}).

Die kontextfreie Grammatik $G_{\text{Terme}} = (\Sigma, N, P, S)$ mit

- $\Sigma = \{LB, RB, LS, CONS, LE, +, -, *, /, SEP, INT, SYM\}$,
- $N = \{S, T, A, F, L\}$ und

$P : S \rightarrow LB \ T \ RB$

$T \rightarrow SYM \mid A \mid L \mid T \ SEP \ T \mid \epsilon$

$A \rightarrow A + A \mid A - A \mid A * A \mid A / A \mid LB \ A \ RB$

$F \rightarrow SYM \ LB \ T \ RB$

$L \rightarrow LS \ T \ LE \mid LS \ SYM \ CONS \ SYM \ LE \mid LS \ SYM \ CONS \ L \ LE$

erzeugt alle gültigen Terme.

Terme werden durch einen Bottom-Up Parser, der direkt aus der Grammatik generiert wurde, geparsst (siehe Abschnitt 5.1.2).

4.2 Semantik

Die Semantik von CL-Reason entspricht der Semantik von Prolog. Der SLD-Baum wird entsprechend einer Tiefensuche nach einem Blatt durchsucht.

Zur Bestimmung einer erfüllenden Variablenbelegung wird ein Antwortprädikat verwendet.

Definition 4.5 (Antwortprädikat).

Ein Antwortprädikat ist ein zu einer Zielklausel hinzugefügtes, positives Literal, das als Argumente alle in der Zielklausel vorkommenden Variablen enthält.

Aus der Anfrage

`exists x y: p(x,y).`

entsteht also die initiale Zielklausel $\{\neg p(x,y), \text{answer}(x,y)\}$.

Da bei der SLD-Resolution an jedem Resolutionsschritt eine Goalklausel beteiligt ist, ist immer auch ein Antwortprädikat beteiligt. Das Ziel der Resolution ist nun nicht mehr die leere Klausel abzuleiten, sondern eine Klausel die nur noch das Antwortprädikat enthält. Die erfüllende Variablenbelegung kann dann in der Argumentliste des Antwortprädikats abgelesen werden.

4.3 Algorithmen

Die zentrale Operation des Schlussfolgerungssystems ist die Unifikation von Literalen. In jedem Schritt der Lösungsfindung legt die SLD-Strategie fest, zwischen welchen Klauseln eine Unifikation durchgeführt werden soll, und was bei Nichterfolg zu tun ist.

In diesem Schritt werden der verwendete Unifikationsalgorithmus sowie der Resolutionsalgorithmus nach SLD-Strategie beschrieben.

4.3.1 Unifikationsalgorithmus

Die Unifikation von Literalen wurde im Abschnitt 2.1.3 auf die Unifikation von Termen zurückgeführt. Der verwendete Unifikationsalgorithmus ist eine rekursive Variante des Unifikationsalgorithmus für Termlisten nach [Rob65].

Wir betrachten zunächst die Komposition zweier idempotenter Substitutionen.

Definition 4.6 (Komposition von Substitutionen).

Seien

$$\sigma = \{x_1 \leftarrow s_1, \dots, x_n \leftarrow s_n\} \quad (4.1)$$

$$\tau = \{y_1 \leftarrow t_1, \dots, y_m \leftarrow t_m\} \quad (4.2)$$

zwei Substitutionen, sodass

$$\{x_1, \dots, x_n\} \cap \{y_1, \dots, y_m\} = \emptyset \quad (4.3)$$

Dann ist

$$\sigma \circ \tau = \{x_1 \leftarrow \tau s_1, \dots, x_n \leftarrow \tau s_n\} \cup \tau \quad (4.4)$$

Der folgende Algorithmus liefert den mgu für zwei Termlisten $s = (s_1, \dots, s_n)$ und $t = (t_1, \dots, t_n)$.

$$\text{unify}(s, t, \sigma) =_{\text{def}} \begin{cases} \sigma & \text{length}(s) = \text{length}(t) = 0 \\ \text{unify}(\text{rest}(s), \text{rest}(t), \sigma \circ \{\sigma s_1 \leftarrow \sigma t_1\}) & \sigma s_1 \text{ oder } \sigma t_1 \text{ ist Variable (o.B.d.A sei } \sigma s_1 \text{ Variable) und } \sigma s_1 \text{ kommt nicht in } \sigma t_1 \text{ vor} \\ \text{unify}(\text{rest}(s), \text{rest}(t), \sigma \circ \tau) & \sigma s_1 = f(x_1, \dots, x_m) \text{ und } \sigma t_1 = (y_1, \dots, y_m) \\ \text{fail} & \text{und } \tau = \text{unify}((x_1, \dots, x_m)(y_1, \dots, y_m), \{\}) \neq \text{fail} \\ & \text{sonst} \end{cases}$$

4.3.2 SLD-Resolutionsalgorithmus

Wie bereits im Kapitel 3 beschrieben, ist eine extreme Einschränkung der Auswahlmöglichkeiten in jedem Resolutionsschritt nötig, um die SLD-Resolution deterministisch implementierbar zu machen. Auch in diesem System werden Programmklauseeln anhand ihrer Position im Programm ausgewählt. Neu entstandene Teilziele werden vorne in die Resolvente eingefügt und direkt im nächsten Resolutionsschritt bearbeitet. Der Algorithmus entspricht also einer Tiefensuche im SLD-Baum.

Wir entwickeln den Algorithmus anhand eines Beispiels. Sei folgendes CL-Reason Programm gegeben.

```
q(3).
s(4).
p([]).
forall f r: p([f|r]) <= q(f) and p(r).
forall f r: p([f|r]) <= s(f) and p(r).
```

Anfragen an das Prädikat p liefern **true**, wenn alle Elemente der als Argument übergebenen Liste entweder Dreien oder Vieren sind. Wird nun die Anfrage

$p([3,4]).$

gestellt, wird eine SLD-Resolution auf der Klauselmenge

$$\begin{aligned} & \{ \{q(3)\}^1, \\ & \{s(4)\}^2, \\ & \{p(empty)\}^3, \\ & \{p(cons(f,r), \neg q(f), \neg p(r))\}^4, \\ & \{p(cons(f,r), \neg s(f), \neg p(r))\}^5, \\ & \{\neg p(cons(3, cons(4, empty)))\}^6 \} \end{aligned}$$

angestoßen. Klausel sechs ist dabei die erste Zielklausel.

Wie in der Tiefensuche üblich, werden die zu expandierenden Knoten auf einen Stack gelegt. Da das erzeugen von Resolventen der aufwendigste Schritt in diesem Algorithmus ist, wird immer nur eine Resolvente erzeugt und diese direkt als neue Goalklausel betrachtet.

Um an jedem inneren Knoten des Baumes die nächste Ableitung bestimmen zu können, werden die potenziell zum ersten Teilziel passenden Programmklauseeln auf einem Stack gelegt. Wir bezeichnen diesen Stack als aktuelle Prozedur. Diese Prozedur wird dann auf einen Stack von Prozeduren gelegt, dessen Zweck nach dem nächsten Schritt deutlich wird. Abbildung 4.1 zeigt die Startkonfiguration.

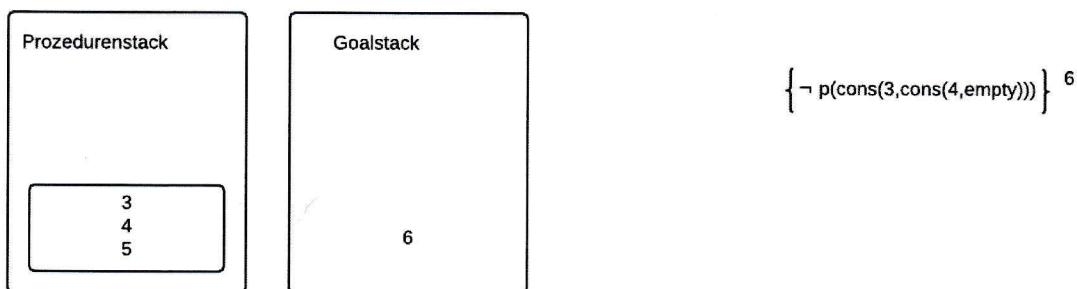


Abb. 4.1. Startkonfiguration des Algorithmus

Nun werden der Reihe nach Unifikationen mit den Programmklauseeln der aktuellen Prozedur versucht. Die erste erfolgreiche Unifikation findet mit Klausel vier statt. Die daraus resultierende Resolvente wird auf den Stack der Goalklauseln gelegt. Sollte eines der Teilziele in dieser Goalklausel nicht gelöst werden können, gelangen wir durch Backtracking (entfernen von Goalklauseln vom Stack) wieder zur ersten Goalklausel. An dieser Stelle muss bekannt sein, mit welchen Programmklauseeln bereits eine Unifikation versucht wurde. Zu diesem Zweck legen wir die aktuelle Prozedur auf einen Stack. Auf diesen Stack wird nach der Ableitung einer neuen Klausel auch direkt die zum ersten Teilziel der Resolvente passende Prozedur gelegt. Die Abbildung 4.2 verdeutlicht die neue Konfiguration.

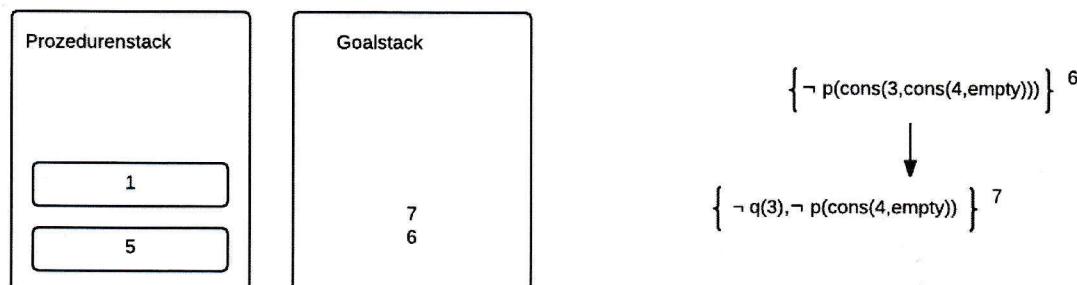


Abb. 4.2. Konfiguration nach der ersten Ableitung

Nun werden erneut Unifikationen zwischen dem ersten Teilziel $\neg q(3)$ und den Programmklauseln der obersten Prozedur versucht. Dies gelingt und es entsteht die Resolvente $\{\neg p(\text{cons}(4, \text{empty}))\}$. Die neue Zielklausel wird auf den Stack der Zielklauseln gelegt und die zum ersten Teilziel $\neg p(\text{cons}(4, \text{empty}))$ passende Prozedur wird auf den Prozedurenstack gelegt.

Dieses Verfahren wird fortgesetzt, bis es zur in Abbildung 4.3 gezeigten Situation kommt.

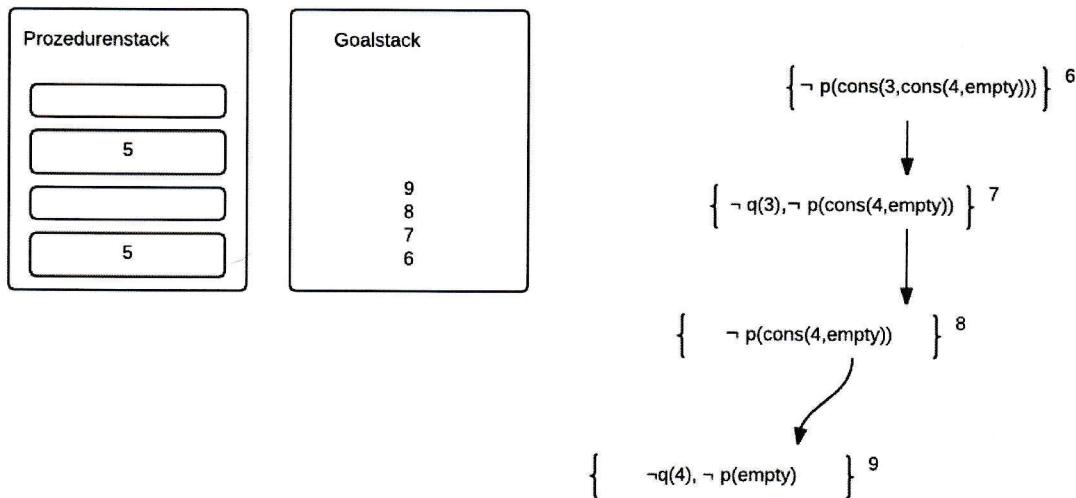


Abb. 4.3. Konfiguration vor erstem Backtracking

Das durch die Resolution der Klauseln acht und vier eingeführte Teilziel $\neg q(4)$, kann nicht durch die Programmklauseln in der obersten Prozedur gelöst werden. Eine leere Prozedur auf dem Prozedurenstack ist das Signal zum Backtracking, da dies nur in Situationen auftritt, in denen ein Teilziel nicht durch das Programm gelöst werden kann. Nun wird sowohl die oberste Prozedur als auch die oberste Goalklausel verworfen und das Verfahren mit den neuen obersten Objekten beider Stacks vorgesetzt. Als nächstes wird also nach einer alternativen Ableitung für Klausel acht gesucht. Dies gelingt mit der Klausel fünf und es entsteht die Situation in Abbildung 4.4.

Nun kann das Verfahren ohne weiteres Backtracking fortgesetzt werden. Durch die zweite Klausel kann das Teilziel $\neg s(4)$ aufgelöst werden und durch Klausel drei schließlich das Teilziel $\neg p(\text{empty})$. Der Algorithmus bricht mit **FAIL** ab, wenn der Prozedurenstack leer ist.

Die Implementierung dieses Verfahren stellt den Kern des Schlussfolgerungssystems dar.

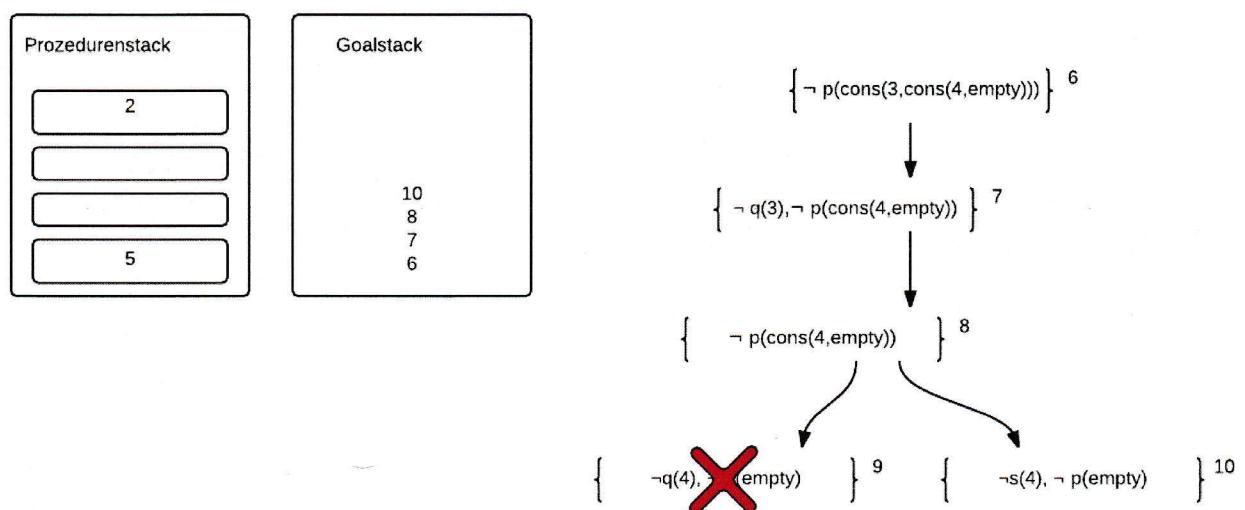


Abb. 4.4. Konfiguraton nach finden einer alternativen Ableitung

5

Implementierung

In diesem Kapitel wird auf Implementierungsdetails des Parsers und die durch ihn erzeugte interne Repräsentation von Formeln eingegangen. Der Unifikationsalgorithmus und das Resolutionsverfahren wurden bereits in Kapitel 4 ausführlich beschrieben. Die Implementierung dieser beiden Algorithmen wird nicht näher beschrieben.

5.1 Lexing und Parsing

Geparst werden Programme aus einer einfachen Textdatei. Im ersten Schritt wird der Eingabetext durch einen regulären Ausdruck in Tokens zerlegt (das sog. lexing). Der reguläre Ausdruck wurde mit der Library cl-ppcre [cl-a] erstellt. Die Syntax für reguläre Ausdrücke in dieser Library entspricht der Syntax von Perl.

```
forall|exists|and|<=|=>|:|[a-zA-Z0-9]+\\(.*)\\)(?=\\s*<=)
| [a-zA-Z0-9]+\\(.*)\\)(?=\\s*and) | [a-zA-Z0-9]+\\(.*)\\)(?=\\s*=>)
| [a-zA-Z0-9]+\\(.*)\\)(?=\\.) | [a-z]+\\.
```

Die ersten sechs Fälle entsprechen vordefinierten Schlüsselwörtern. Die nächsten vier Fälle matchen Prädikatsdefinitionen. Das Ende einer Prädikatsdefinition (die letzte schließende Klammer) wird über einen Lookahead identifiziert. So wird eine Prädikatsdefinition nur dann gematcht, wenn auf sie entweder `<=`, `and`, `=>` oder `.` folgt.

Nachdem der Eingabetext so in Schlüsselwörter und Prädikatsdefinitionen zerlegt wurde, werden für Prädikatsdefinitionen Symbole angelegt und der Name sowie die Argumentenliste in Properties gespeichert. Die Argumentenliste liegt nach diesem Schritt noch als String vor und muss in einem späteren Schritt durch den Termparser geparsed werden (Abschnitt 5.1.2). Auch für Variablen werden Symbole angelegt, die nach dem Parsen der Terme an den entsprechenden Stellen in die Terme eingefügt werden. So ist sichergestellt, dass die Variablen innerhalb einer Klausel eindeutig, von Klausel zu Klausel jedoch unterschiedlich sind.

5.1.1 Parser für Programme

Der erste Parser stellt fest, ob die vom Lexer erzeugte Folge von Tokens ein gültiges Programm ist. Es handelt sich dabei um einen *LL(2)*-Parser, da die Eingabe von

links nach rechts abgearbeitet wird und anhand eines Lookaheads von zwei Terminalsymbolen eine Ableitung für das am weitesten Links stehende Nichtterminal ausgewählt wird. Geparste Prädikatssymbole werden je nach Position im Programm als positives oder negatives Literal einer Klausel interpretiert und in entsprechenden Properties eines Klauselsymbols gespeichert.

Der Parser ist in mehrere Funktionen aufgeteilt. Gesteuert wird das Parsing in der Hauptfunktion `parse-program`, die die als nächstes zu expandierenden Nichtterminale, die schon teilweise geparste Tokenfolge sowie die bereits erstellte Klauselmenge verwaltet.

```
(defun parse-program (input)
  (do* ((ntl '(s)) ;; Liste der Nichtterminale, zu Beginn zur s
        (program input)
        (output NIL)
        (la (take 2 program)
             (take 2 program))
             ;; zwei Terminalsymbole als Lockahead
        (expansion (funcall (next-nt ntl) la)
                   (funcall (next-nt ntl) la)))
             ;; Aufruf der Funktion zum am weitesten
             ;; links stehenden Nichtterminal
        (production (nth 2 expansion) (nth 2 expansion))
             ;; Zahlencode der gewählten Ableitung
        (to-match (car expansion) (car expansion))
             ;; zu matchende Terminalsymbole
        (new-nt (cadr expansion) (cadr expansion)))
             ;; als nächstes zu expandierende Nichtterminale
        ((and (not program) (not ntl)) (reverse output))
             ;; Abbruchbedingung
        (case production
          ...
          ;; Erweiterung des Output je nach Produktion
          ...
        )
        (setq program (match (car expansion) program))
             ;; konsumieren der Eingabe von links
        (setq ntl (append new-nt (cdr ntl)))
             ;; erweitern der Nichtterminalliste)))
```

Das expandieren von Nichtterminalen wird anhand eines Lookaheads von gesonderten Funktionen übernommen. Dabei gibt es für jedes Nichtterminal eine Funktion, die anhand des Lookaheads entscheidet, welche Terminalsymbole als nächstes gematcht werden können und welche Nichtterminale als nächstes expandiert werden müssen. Die Regeln nach denen diese Nichtterminalfunktionen arbeiten ergeben sich aus der Grammatik aus Definition 4.2. Wir betrachten hier beispielhaft die Funktion zum Nichtterminal S.

```
(defun s (la)
  (cond ((la-check la '(forall var))
         '((forall) (v k s) 0))
        ((la-check la '(exists var))
         '((exists) (v k s) 1))
        ((la-check la '(lit end))
         '((lit end) (s) 2))
        ((la-check la NIL)
         '(nil nil 3)))
  (T (error "no fitting production"))))
```

Die Funktion erhält aus der Hauptfunktion die nächsten zwei Symbole der Eingabe als Parameter `la`. Die Funktion `la-check` liefert true, falls der Wert der Symbole in `la` den Symbolen im zweiten Argument entsprechen. Diese Funktion ist notwendig, das z. B. die Literaltokens in der Eingabe des Parsers durch den Lexer bereits mit Propertys versehen sind.

Passt der Lookahead zu einer möglichen Produktion wird ein Trippel zurückgeliefert. Das erste Element dieses Trippels ist eine Liste mit Terminalsymbolen, die nun durch die Hauptfunktion gematcht werden können. Die Anzahl der Terminalsymbole ist situationsbedingt. Ist es anhand des Lookahead möglich mehr als ein oder zwei Terminalsymbole zu matchen so wird dies auch getan.

Das zweite Element des Trippels ist eine Liste mit Nichtterminalen, die als nächstes zu expandieren sind. Diese Liste wird in der Hauptfunktion vorne an die Liste der als nächstes zu expandierenden Nichtterminale angefügt.

Das letzte Element des Trippels ist ein Zahlencode, der die in der Hauptfunktion auszuführenden Aktionen beschreibt. Je nach Antwort der Nichtterminalfunktion werden dann in der Hauptfunktion die nötigen Aktionen angestoßen. Dazu gehören das Anlegen einer neuen Klausel beim Parsen eines Quantors, das Hinzufügen von geparssten Variablensymbolen zur Liste von Variablen einer Klausel oder das setzen eines geparssten Literals als positives oder negatives Literal einer Klausel. So wird bereits beim Parsen eine gewisse Interpretation der Eingabe vorgenommen, die sich in der internen Repräsentation der Daten widerspiegelt.

Die interne Struktur eines geparssten Programms wie sie vom Parser erzeugt wird entspricht der Abbildung 5.1

5.1.2 Parser für Terme

Der Parser für Terme wurde durch einen Parsergenerator direkt aus der Grammatik aus Definition 4.4 generiert. Verwendet wurde die Parsergenerator Library cl-yacc [cl-b]. Der erzeugte Parser arbeitet auf einer Folge von Tokens die aus den in Properties der Literale gespeicherten Strings erzeugt werden.

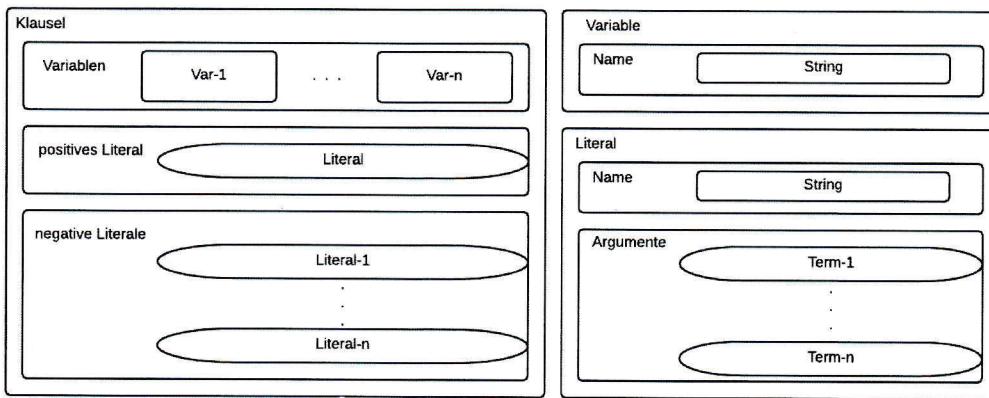


Abb. 5.1. interne Repräsentation von Programmen

```
(defun tokenize-terms (termstring)
  (mapcar (lambda (x)
    (cond ((string= x "(") 'LB)
          ((string= x ")") 'RB)
          ((string= x "[") 'LS)
          ((string= x "|") 'CONS)
          ((string= x "]") 'LE)
          ((string= x "+") '+)
          ((string= x "-") '-)
          ((string= x "*") '*)
          ((string= x "/") '/)
          ((string= x ",") 'SEP)
          ((cl-ppcre:scan "\\d+" x) (parse-integer x))
          (T (read-from-string x))))
    (remove " " (all-matches-as-strings *regex-term* termstring)
           :test 'string))))
```

Auch bei diesem Parser werden nach dem Parsen eines Teils der Eingabe bestimmte Operationen ausgeführt, die die geparsten Terme in eine interne Repräsentation überführen.

5.2 Interpreter

Nachdem ein Großteil der für die Resolution benötigten Informationen bereits durch die Parser in einer geeigneten internen Repräsentation gespeichert wurde, sind nur noch Terme, insbesondere Listen gesondert zu interpretieren.

Da für jede Klausel explizit angegeben werden muss, welche Symbole Variablen sind und für jede Variable bereits durch den Lexer ein Symbol angelegt wurde, müssen lediglich alle Vorkommnisse dieses Symbols in allen Termen einer Klausel durch das bereits existierende Variablen-Symbol ersetzt werden.

5.2.1 Listen

Der Termparser unterscheidet zwischen konkreten Listen der Form $[t_1, \dots, t_n]$ und abstrakten Listen der Form $[x|y]$. Für abstrakte Listen kann direkt eine entsprechende Darstellung als Term $cons(x, y)$ abgelesen werden.

Konkrete Listen müssen in einem gesonderten Schritt in diese Form überführt werden. Dazu wird rekursiv aus den geparssten Elementen der Liste ein Funktionsterm erzeugt.

Aus der Liste $[1, 2, 3, 4]$ wird so der Funktionsterm

$$cons(1, cons(2, cons(3, cons(4, empty)))) \quad (5.1)$$

Abbildung 5.2 zeigt die finale interne Repräsentation von Termen.

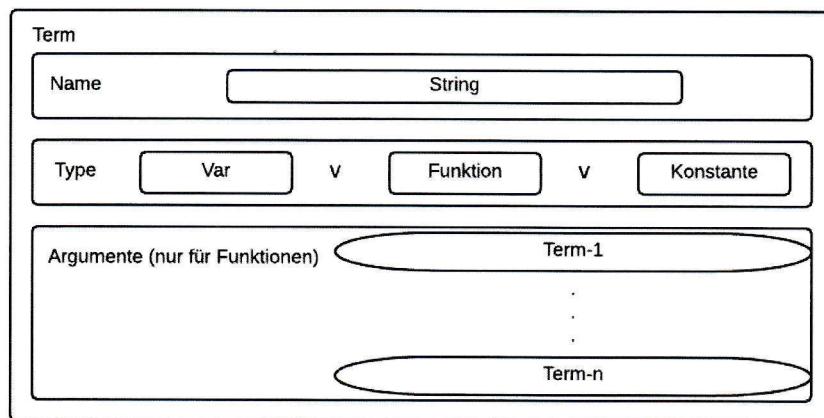


Abb. 5.2. interne Repräsentation von Termen

5.3 Logisches Schlussfolgern

Die beiden Kernfunktionen für die Implementierung des Schlussfolgerungssystems sind die Implementierungen des Unifikationsalgorithmus und des SLD-Resolutionsalgorithmus.

Die im Abschnitt 4.3.1 angegebene rekursive Funktion zur Unifikation von Termlisten wurde rekursiv implementiert. Erweitert auf Literale wird sie während der SLD-Resolution verwendet.

```

(defun unify-literals (lit1 lit2)
  (let ((name1 (get lit1 'name))
        (name2 (get lit2 'name)))
    (args1 (mapcar 'copy-term-symbol (get lit1 'args)))
    (args2 (mapcar 'copy-term-symbol (get lit2 'args))))
  (if (and (string= name1 name2)
           (= (length args1)
              (length args2)))

```

```
(unify-termlists args1 args2 '())
'fail)))
```

Da die Anwendung von Substitutionen Teil des Algorithmus ist, ist es nötig, dass während der Unifikation lediglich auf Kopien der Termlisten gearbeitet wird.

Das im Abschnitt 4.3.2 beschriebene Verfahren wurde als iterativer Algorithmus in der **Funktionsold-resolution** implementiert.

Gestartet werden können Programme nach dem Laden des Systems durch einen Aufruf der Funktion **run-program**. Als Argument erhält diese Funktion den Dateipfad des Programms. Da Anfragen direkt mit im Programm stehen, müssen sie durch einen Existenzquantor eingeleitet werden, auch wenn keine Variablen in der Anfrage vorkommen.

6

Ausblick

Das entwickelte System stellt lediglich den logischen Kern eines Schlussfolgerungssystems für Hornklauseln dar. In diesem Kapitel sollen kurz denkbare und sinnvolle Erweiterungen dargestellt werden.

User Interface

Aktuell liegt das Schlussfolgerungssystem lediglich als Lisp Code vor, der in einem Lispsystem geladen werden kann. Programme können dann durch einen Aufruf der Funktion `run-program` über Angabe des Pfades ausgewertet werden. Ein User Interface zum laden von Programmen und interaktivem Stellen von Anfragen wäre eine sinnvolle Erweiterung. Dadurch könnte auch das Problem der unnötigen existenzquantifizierten Variablen in Anfragen gelöst werden.

Vordefinierte Prädikate

Zahlen und arithmetische Ausdrücke erfüllen bisher keinen besonderen Zweck. Zahlen werden als Konstanten und arithmetische Ausdrücke als Funktionsterme interpretiert. Eine gesonderte Verarbeitung über vordefinierte Prädikate sollte leicht in die bestehenden Algorithmen einzubauen sein.

Sorten für Terme

Funktionsterme sind als Symbole realisiert. Dies ermöglicht eine beliebige Erweiterung um Property's. So könnte z. B. jedem Term eine Sorte zugewiesen werden.

Zusammenfassung

Im Zuge dieser Arbeit ist ein Schlussfolgerungssystem für Hornklauselmengen entstanden. Einfache logische Schlüsse werden durch das System bereits gezogen, zum vollständigen Ersatz von Prolog in der Lehrveranstaltung Angewandte Logik ist es jedoch noch nicht geeignet.

Das Auswerten von arithmetischen Ausdrücken durch vordefinierte Prädikate oder das Vergleichen von Zahlen ist noch nicht möglich. Viele der Übungsaufgaben der Veranstaltung erfordern allerdings diese Möglichkeiten.

Dennoch ist durch den Parser und durch die implementierten Verfahren eine gute Grundlage geschaffen worden, die direkt auf den theoretischen Grundlagen der Logik und Informatik aufbaut. Die Grundlagen der logischen und symbolischen Programmierung sowie Rekursion über Listen lassen sich bereits jetzt mit dem System üben.

Die gewählte interne Repräsentation von Formeln ermöglicht eine einfache Erweiterung und Weiterentwicklung des Systems.

Literaturverzeichnis

- Bec88. BECKSTEIN, CLEMENS: *Zur Logik der Logik-Programmierung*. Springer-Verlag, 1988.
- cl-a. *cl-ppcre*. <http://weitz.de/cl-ppcre/>.
- cl-b. *cl-yacc*. <http://wwwpps.univ-paris-diderot.fr/~jch/software/cl-yacc/>.
- CR96. COLMERAUER, ALAIN und PHILIPPE ROUSSEL: *History of Programming languages—II*. Kapitel The Birth of Prolog, Seiten 331–367. ACM, New York, NY, USA, 1996.
- Llo93. LLOYD, JOHN WYLIE: *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd Auflage, 1993.
- Rob65. ROBINSON, J. A.: *A Machine-Oriented Logic Based on the Resolution Principle*. J. ACM, 12(1):23–41, Januar 1965.

A

Erklärung des Kandidaten

Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

Datum

Unterschrift des Kandidaten