Logikprogrammierung mit Abhängigkeiten

Bachelorarbeit

Martin Lück
Matrikel-Nr. 2787070

26. Juni 2013

Institut für Theoretische Informatik
Leibniz Universität Hannover

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen genutzt habe.

Hannover, den 26. Juni 2013

Martin Lück

Inhaltsverzeichnis

1 Logische Programmierung mit Prolog **7**
1.1 Grundlegende Einführung in Prolog 7
1.2 Die Syntax von Prolog-Programmen 9
1.3 Prädikatenlogische Semantik von Prolog 11
1.3.1 Prolog-Regeln und äquivalente Axiome 11

2 Prädikatenlogische Resolution **15**
2.1 Erfüllbarkeit und Modelle 15
2.1.1 Aussagenlogische Erfüllbarkeit 15
2.1.2 Prädikatenlogische Erfüllbarkeit 15
2.2 Herbrand-Theorie 17
2.2.1 Skolemformen 18
2.2.2 Herbrand-Modelle 19
2.3 Grundresolution 22
2.4 Verfeinerung der Resolution 23
2.4.1 Unifikation 24
2.4.2 Die SLD-Resolution 25

3 Dependence Logic und Teamresolution **33**
3.1 Einführung in Dependence Logic 33
3.1.1 Funktionale Abhängigkeiten 34
3.1.2 Teams und Erfüllbarkeit in der Dependence Logic 35
3.2 Verträglichkeit mit Prolog 37
3.2.1 Teams in Prolog 37
3.2.2 Erweiterte Hornformeln 38
3.2.3 Resolution in $\mathcal{D}$ 39
3.2.4 Prolog und relationale Algebra 40
3.3 Implementationsansätze 41
3.3.1 Einschränkungen der Sprache 41
3.3.2 Interpretation der Lösungsmenge als Team 44
3.3.3 Binäre Teamresolution 46

4 Entwurf eines Interpreters für Dependence-Prolog 53
4.1 Lexikalische Analyse . . . 54
4.2 Syntaktische Analyse . . . 57
4.3 Weiteres Vorgehen bei der Implementierung . . . 61
4.3.1 Semantische Analyse und Übersetzung . . . 61
4.3.2 Hinweise zu Prolog/Datalog . . . 61
4.4 Bedienung des Programms . . . 62

5 Abgrenzung und Fazit 65

Anhang 67

A Beweise 67
A.1 Erfüllbarkeitsäquivalenz des Herbrand-Modells . . . 67

B Algorithmen 69
B.1 Unifikation zweier Literale . . . 69
B.2 Allgemeine SLD-Resolution . . . 70
B.3 Rekursive SLD-Resolution mit Tiefensuche . . . 71
B.4 Überprüfung eines Abhängigkeitsausdrucks δ . . . 72
B.5 SLD mit Tiefensuche und subtree-cutting . . . 73
B.6 Lexikalischer Scanner (Tokenizer) . . . 74
B.7 Syntaktische Analyse (Parser) . . . 75

Index 77

Literaturverzeichnis 79

Zusammenfassung

Das Ziel dieser Arbeit ist, eine Implementierung für Jouko Väänänens *Dependence Logic* zu schaffen, die eine Erweiterung der gewöhnlichen Prädikatenlogik um sogenannte *funktionale Abhängigkeiten* darstellt.[Vää07] Die Dependence Logic soll in die logische Programmiersprache *Prolog* eingebettet werden, da sich logische Formeln in Prolog bereits effizient auswerten lassen. Es soll erarbeitet werden, wie funktionale Abhängigkeiten in Prolog formuliert werden können und ob diese Formeln überhaupt so effizient wie die Prädikatenlogik erster Stufe ausgewertet werden können. Die Implementierung als praktischer Teil dieser Arbeit soll aus einem eigenständigen Parser und Interpreter bestehen.

Beim Leser werden im Folgenden grundlegende und im Bachelorstudiengang Informatik vermittelte Kenntnisse über Logik, Algorithmen, formale Sprachen und über den Compilerbau angenommen.

Zunächst soll gezeigt werden, wie logische Programmierung auf Basis von Hornformeln in Prolog funktioniert. Dafür wird in Kapitel 1 auf die Syntax und Semantik eingegangen und es werden die einzelnen Sprachbausteine auf ihren Ursprung in der Prädikatenlogik zurückgeführt.

Mit formalen Mitteln werden dann in Kapitel 2 die theoretischen Grundlagen für die Arbeitsweise von Prolog hergeleitet. Es wird erläutert, warum und wie prädikatenlogische Formeln überhaupt algorithmisch gelöst werden können. Eine bekannte und in Prolog üblicherweise verwendete Umsetzung für Hornformeln ist der vorgestellte SLD-Algorithmus, eine Variante des Resolutionskalküls.

Anschließend werden in Kapitel 3 Konzepte der funktionalen Abhängigkeiten und der Dependence Logic erläutert. Es wird untersucht, welche Schwierigkeiten es bei der Einbettung dieser in die Sprache Prolog gibt. Es werden Lösungsansätze aufgezeigt. Das Ziel ist die Definition einer um Abhängigkeitsausdrücke erweiterten Sprache und der Entwurf eines Algorithmus, der Anfragen in der erweiterten Sprache beantworten kann.

Zuletzt wird in Kapitel 4 mit formalen Methoden ein Parser und Interpreter entworfen, der *Dependence Prolog*-Programme, also Prolog um Abhängigkeitsausdrücke erweitert, analysiert und verarbeitet. Der Parser basiert auf einer kontextfreien Grammatik für die erweiterte Programmiersprache.

Der in dieser Arbeit entwickelte Interpreter stellt dabei nur ein Grundgerüst dar, so werden arithmetische Ausdrücke, Listen und andere nichtlogische Prolog-Befehle wie etwa *cut* nicht unterstützt.

1 Logische Programmierung mit Prolog

Deklarative Programmiersprachen ermöglichen eine Auswertung von Eingaben, ohne dass eine genaue Berechnungsvorschrift dafür angegeben werden muss. Der Programmierer beschreibt lediglich das Problem, das gelöst werden soll. Das System ermittelt selbstständig eine Lösung und gibt diese aus.

Ein bekanntes Beispiel ist neben funktionalen Programmiersprachen wie Haskell und Datenbanksprachen wie SQL auch die logische Programmiersprache *Prolog (programmation en logique)*, die hier näher betrachtet wird.

Ein Prolog-System verfügt über eine aus logischen Aussagen — sogenannten *Fakten* und *Regeln* — bestehende Wissensdatenbank, die beispielsweise aus einer Datei gelesen wird. Eingaben vom Programmierer erfolgen dann genau wie Fakten und Regeln als logische Aussagen in Form von *Anfragen*, denen sich Wahrheitswerte zuordnen lassen. Das System versucht diese mit dem vorhandenen Wissen automatisch zu erschließen. Die beteiligten syntaktischen Elemente sollen im folgenden Abschnitt erläutert werden.

Ein großer Teil der Nützlichkeit von Prolog ergibt sich aus der Möglichkeit, Variablen in Anfragen einzubringen, so dass nicht nur die Aussage selbst abgeleitet wird, sondern im Erfolgsfall auch alle passenden Werte ausgegeben werden.

In diesem Kapitel soll der Aufbau der Sprache analysiert werden und an Beispielen ein Grundverständnis für die Arbeitsweise von Prolog-Interpretern vermittelt werden, wobei wir uns auf eine Teilmenge der Sprache beschränken, die *Pure Prolog* genannt wird. Diese arbeitet nur mit logische Aussagen, während Prolog nach [ISO96] noch zahlreiche Kontrollstrukturen, Ein- und Ausgabeprozeduren und Arithmetik unterstützt. Durch die Nutzung dieser Möglichkeiten erhalten Prolog-Programme oft einen prozeduralen Charakter, der in dieser Arbeit aber keine Rolle spielen soll.

1.1 Grundlegende Einführung in Prolog

Beispiel 1.1.
Ein in Prolog formuliertes Programm könnte so aussehen:

```
q :- p, r.
p.
r :- p.
```

Der Ausdruck `q :- p, r.` ist eine Regel und damit eine logische Aussage. Das Symbol `:-` kann dabei als ein Implikationspfeil verstanden werden, der von rechts nach links zeigt: Aus `p, r` folgt `q`. Das Komma steht dabei für ein logisches „und“. Man kann `p`, `q` und `r` hier als aussagenlogische Variablen interpretieren.

Die Zeile `p.` ist ebenfalls eine Aussage. Aussagen ohne Vorbedingung heißen auch *Fakt*: `p` gilt also immer, unabhängig von den restlichen Regeln und Fakten.

In Prolog ist es möglich, anstelle von aussagenlogischen Variablen auch sogenannte zusammengesetzte Terme zu formulieren. Diese können abhängig vom Kontext als prädikatenlogische Relationen, als Funktionsaufrufe oder als hierarchische Strukturen interpretiert werden.

Beispiel 1.2.
Zu einer Menge von Prolog-Regeln können Anfragen gestellt werden, die durch das Symbol `?-` eingeleitet werden.

```
neumond.

scheint(sonne, tag).
scheint(mond, nacht) :- vollmond.

hell(X) :- scheint(sonne, X).
hell(X) :- scheint(mond, X).

?- hell(tag).
?- hell(nacht).
```

Um eine Anfrage zu beantworten, das heißt, `yes` oder `no` zurückzugeben, versucht Prolog, die Anfrage syntaktisch aus den bisher bekannten Aussagen abzuleiten. In Kapitel 2 wird gezeigt, dass die Existenz einer syntaktischen Ableitung zum semantischen Wahrheitsbegriff äquivalent ist.

Damit der Aussage `hell(tag)` der Wert „wahr“ zugeordnet werden kann, muss sie eine Folgerung aus bekannten Regeln oder Fakten sein. Prolog sucht also in der Datenbank nach einer Aussage, die etwas über den Ausdruck `hell(tag)` aussagt.

Enthält die linke Seite einer Regel oder die Anfrage selbst jedoch Variablen, reicht kein bloßer lexikalischer Vergleich mit der Wissensdatenbank. Stattdessen werden die zu vergleichenden Ausdrücke *unifiziert*. Beim Vorgang der *Unifikation* werden Variablen gerade soweit wie nötig durch Terme ersetzt, so dass die beiden Ausdrücke übereinstimmen. Das Verfahren der Unifikation wird in Abschnitt 2.4 noch weiter erläutert.

Für die Anfrage `hell(tag).` findet Prolog zwar keine übereinstimmenden Fakten, gewinnt durch Unifikation aber zwei „passende“ Regeln aus der Datenbank:

```
hell(tag) :- scheint(sonne, tag).
hell(tag) :- scheint(mond, tag).
```

Die Auswertung einer Regel ergibt dann „wahr“, wenn der Rumpf rekursiv abgeleitet werden kann. Mehrere gefundene Regeln werden als Alternativen, d. h. als Disjunktion betrachtet. Da hier sofort `scheint(sonne, tag).` als Fakt gefunden wird, ist die Ausgabe insgesamt `yes`.

Andersherum kann die Anfrage `hell(nacht).` nicht auf Fakten zurückgeführt werden. Prolog nimmt in diesem Fall automatisch an, dass die Anfrage eine falsche Aussage ist. Dieses Verhalten heißt auch *Closed World Assumption*[Par11]: Alles, was nicht explizit wahr ist, ist falsch.

Eine Anfrage mit einer Variablen, etwa `hell(T).`, wird genauso ausgewertet. Es werden lediglich zusätzlich alle möglichen Belegungen für `T` ausgegeben, für die sich eine Ableitung aus der Regelmenge ergibt, in diesem Fall: `T = day.` Somit sind auch konkretere Ausgaben als ein Wahrheitswert möglich.

1.2 Die Syntax von Prolog-Programmen

Die syntaktischen Bausteine von Prolog-Programmen lassen sich wie folgt definieren:

Atomare Terme. Dies sind Terme, die nicht weiter zerlegbar sind, genauer:

Atome. Dies sind die meisten Zeichenketten, die in einem Programm auftreten. Sie dienen in Prolog als Bezeichner und bestehen aus Buchstaben, Zahlen und Unterstrichen (`_`). Sie beginnen entweder mit einem Kleinbuchstaben oder sind in einfache Anführungszeichen (`' '`) gesetzt. Soll innerhalb der Anführungszeichen ein weiteres verwendet werden, so muss dieses durch Verdopplung maskiert werden (`''`). Auch nur aus Sonderzeichen bestehende Zeichenketten werden manchmal als Atome betrachtet, etwa für mathematische Operatoren. Solche Atome werden in *Pure Prolog* aber nicht benötigt.

Variablen. Diese können anstelle von Atomen in Formeln benutzt werden und erlauben, allgemeinere Regeln zu definieren. Variablen bestehen aus einem Großbuchstaben oder Unterstrich, eventuell gefolgt von weiteren Buchstaben, Zahlen oder Unterstrichen.

Zahlen. Prolog erlaubt nach ISO sowohl Ganzzahl- als auch Fließkommaarithmetik, worauf hier zugunsten von *Pure Prolog* verzichtet werden soll. Zahlen sind daher nur eine Art von Atomen.

Zusammengesetzte Terme. Ein Term `f(x_1, x_2, ..., x_n)` heißt *komplexer Term*, *zusammengesetzter Term* oder *Struktur*. `f` heißt dann *n-stelliger Funktor* und ist stets ein Atom, während die Argumente beliebige Terme sind. Diese Definition ist rekursiv zu verstehen, es sind beliebig tief verschachtelte Strukturen erlaubt. Der äußerste Funktor in einer solchen Struktur ist der sogenannte *Hauptfunktor*.

Logische Verknüpfungen. Hier gibt es die Verknüpfungen Konjunktion (`,`), Disjunktion (`;`) und Implikation (`:-`), wobei eine Regel entweder aus genau einer Implikation besteht oder ein Fakt ist. Die Konjunktion hat Priorität vor der Disjunktion, Klammerung boolescher Ausdrücke ist nicht vorgesehen.

Ein Fakt kann auch verstanden werden als eine Implikation mit leerer Konjunktion auf der rechten Seite, die dem Wahrheitswert $\top$ (`true`) entspricht. Regeln werden mit einem Punkt (`.`) abgeschlossen.

In Prolog wird `%` als Kommentarzeichen benutzt, der Kommentar reicht von `%` bis zum Zeilenende. Gelegentlich werden auch die C-Kommentare `//` und `/* ... */` unterstützt.

Als nächstes wird nun der Aufbau und die Semantik eines gültigen Programms rekursiv definiert. In jedem Punkt wird jeweils der Aufbau des fettgedruckten Elements erläutert.

- Ein **Programm** besteht aus einer Menge von Klauseln. Jede Klausel ist ein Fakt oder eine Regel.

- Ein **Fakt** ist ein positives Literal.

- Eine **Regel** ist eine Implikation eines positiven Literals (dem *Regelkopf*) durch eine logische Verknüpfung von einem oder mehreren positiven Literalen[1] (dem *Regelrumpf*).

- Ein **Literal** ist ein Atom oder eine Struktur. Hierbei ist zu beachten, dass nicht jede Struktur oder jedes Atom ein Literal ist! Einzig der äußerste Funktor einer Struktur, der **Hauptfunktor**, bildet mit seinen Argumenten ein Literal, andere Strukturen werden lediglich als „Funktionsaufrufe“ beziehungsweise hierarchische Anordnungen von Termen interpretiert.

- Alle Literale mit gleichem Funktor f und gleicher Stelligkeit n werden auch als ein **Prädikat** zusammengefasst und `f/n` genannt. Atome zählen hierbei als 0-stellige Prädikate.

- Disjunktionen im Rumpf einer Regel werden so ausgewertet wie mehrere Einzelklauseln mit den Disjunktionsgliedern als Rümpfen. Konjunktionen haben in Regeln also Priorität.

- Eine **Anfrage** ist eine nichtleere Konjunktion von positiven Literalen.

- Gleichnamige Variablen innerhalb einer Klausel bezeichnen identische logische Variablen. Variablen verschiedener Klauseln sind jedoch auch bei Namensgleichheit stets verschieden.

- Variablen, die im Kopf einer Klausel auftauchen, sind automatisch allquantifiziert, Regeln, die nur im Rumpf auftauchen, existenzquantifiziert. Alle Variablen sind also im prädikatenlogischen Sinne lokal in einer Klausel gebunden.

- Die nur aus dem Unterstrich (`_`) bestehende Variable ist anonym und ist verschieden von allen anderen anonymen oder benannten Variablen.

[1] Negative Literale sind in ISO-Prolog durch das Prädikat `not` ausdrückbar, erfordern dann aber eine komplexere Semantik. Dies wollen wir hier umgehen. Das implizierte Literal darf jedoch nie negativ sein.

1.3 Prädikatenlogische Semantik von Prolog

Prolog-Aussagen entsprechen prädikatenlogischen Formeln, die keine Quantifizierung über Prädikate oder Funktionen zulassen. Damit entsprechen sie Formeln der Prädikatenlogik erster Stufe *(first-order logic, $\mathcal{FO}$)*.

Deren Syntax und Semantik soll an dieser Stelle jener Definition aus [Vol12] folgen, die die Formeln der Aussagenlogik um Relationen, Terme und Quantoren erweitert.

Zweck dieses Abschnitts ist, die in Prolog ausdrückbaren Formeln auf die Sprache der Prädikatenlogik zurückzuführen. Auf dieser Basis wird in Kapitel 2 dann eine theoretische Grundlage für einen eigenen Interpreter geschaffen bzw. die Grundlagen hinter den in der Praxis verwendeten Algorithmen erläutert.

Definition 1.3. Jede Prolog-Wissensbasis lässt sich auf eine endliche Menge $\Phi \subset \mathcal{FO}$ von prädikatenlogischen Formeln zurückführen. Für ein festes Programm werden die Elemente $\varphi \in \Phi$ auch **Axiome**[ISO96] genannt, Φ ist die dazugehörige **Axiomenmenge**.

Das Beantworten einer Anfrage $\psi \in \mathcal{FO}$ unter Berücksichtigung der *Closed World Assumption* fällt dann auf das Problem zurück, ob ψ aus der Axiomenmenge Φ semantisch folgt.

1.3.1 Prolog-Regeln und äquivalente Axiome

Die Bestandteile der Sprache Prolog, die jeweils eigene Formeln darstellen, sind Fakten, Regeln und Anfragen. Man stellt fest, dass die äquivalenten prädikatenlogischen Formeln allesamt in der gleichen Form vorliegen: Alle Prolog-Formeln sind *geschlossene prädikatenlogische Hornklauseln*, d. h. in der Form

$$\varphi = \forall \overline{x} \bigvee_{i=1}^{n} L_i$$

wobei höchstens ein Literal L_i positiv sein darf und $\forall \overline{x}$ eine Abkürzung für die Allquantifizierung mehrerer Variablen ist: $\forall x_1 \forall x_2 \ \dots \ \forall x_k$.

Beispiel 1.4.
Für folgendes Prolog-Programm ergeben sich Axiome φ_1, φ_2 und φ_3.

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
taut(W).
dog(spot).
```

Diese sind bis auf die Benennung der Prädikate, Variablen und Konstanten eindeutig:

$$\begin{aligned}
&\varphi_1 \equiv \forall x \forall y \ \ (GP(x,y) \leftarrow \exists z \ P(x,z) \wedge P(z,y)) \\
&\varphi_2 \equiv \forall w \ T(w) \text{ und} \\
&\varphi_3 \equiv D(s).
\end{aligned}$$

Intuitiv lassen sich diese Regeln verstehen als:

- Für alle X und Y gilt, dass X ein Großelternteil von Y ist, wenn es ein Z gibt, so dass X Elternteil von Z ist und Z von Y.
- Alle Werte erfüllen die Relation „taut“.
- Die Konstante „spot“ erfüllt die Relation „dog“.

Regeln

Gegeben sei eine Prolog-Klausel K der Form

```
p(t_01, ..., t_0n) :- l1(t_11, ..., t_1s), ..., lk(t_k1, ..., t_kt).
```

und sei $k > 0$. K ist also eine Regel. Dann enthält K die Prädikate `p/n` sowie `l1/s` bis `lk/t`, wobei `l1` bis `lk` Atome sind und die `t_ij` beliebige Prolog-Terme. Die Formel

$$\varphi = \forall \overline{x} \left(P(\overline{x}) \leftarrow \exists \overline{y} \bigwedge_i L_i(\overline{x}, \overline{y}) \right)$$

ist dann ein Axiom in Φ, wobei das i-te auftretende Literal im Rumpf hier durch L_i abgekürzt ist:

- Ein Literal, das aus einem Atom `a` besteht, wird als Prädikat `a/0` und damit als nullstellige prädikatenlogische Relation A interpretiert.
- Eine Struktur `f(t_1, t_2, ..., tn)` mit n Komponenten, deren Funktor `f` ein Hauptfunktor ist, beschreibt das Prädikat `f/n` und damit die n-stellige Relation $F(t_1, t_2, \ldots, t_n)$.
- Atome, Variablen und Funktoren, die in einem Term auftreten, entsprechen prädikatenlogischen Konstanten, Variablen und Funktionen, wobei Strukturen wie Funktionsterme beliebig tief verschachtelt sein können.

In der Formel schreibe man $\forall \overline{x}$ für die Allquantifizierung aller Prolog-Variablen, die im Kopf der Regel K vorkommen. Dazu analog steht $\exists \overline{y}$ für die Existenzquantifizierung aller Prolog-Variablen, die nicht im Kopf, aber im Rumpf von K mindestens einmal auftauchen. Enthalten die Prädikate keine Variablen, sondern nur Atome oder andere Terme, so ist die Anzahl der jeweiligen Quantoren null. Aus Prolog-Regeln abgeleitete Axiome enthalten generell nur *gebundene Variablen*.

Jede Regel der obigen Form kann auch umgeschrieben werden zu

$$\varphi = \forall \overline{x} \forall \overline{y} \left(P(\overline{x}) \vee \bigvee_i \neg L_i(\overline{x}, \overline{y}) \right)$$

und ist damit nach Definition eine Hornklausel, da alle $\neg L_i$ negative Literale sind und es somit höchstens ein positives Literal P gibt.

Fakten

Liegt eine Regel K vor und ist $k = 0$, dann ist K in der Form

```
p(t_1, ..., t_n).
```

K ist dann ein Fakt und

$$\varphi = \forall \overline{x}\ P(\overline{x})$$

ist ein Axiom in Φ und ebenfalls eine prädikatenlogische Hornklausel.

Anfragen

Wird eine Anfrage an einen Prolog-Interpreter gestellt, so kann diese ebenfalls als logische Formel interpretiert werden.

Die Anfrage

```
?- l1(t_11, ..., t_1m), ..., lk(t_k1, ..., t_ko).
```

entspricht der Formel

$$\psi = \exists \overline{y}\ (L_1(\overline{y}) \wedge L_2(\overline{y}) \wedge \ldots \wedge L_k(\overline{y})).$$

Dies ist offensichtlich keine Hornklausel nach obiger Definition. Wird ψ aber negiert, so erhält man

$$\neg\psi = \forall \overline{y}\ (\neg L_1(\overline{y}) \vee \neg L_2(\overline{y}) \vee \ldots \vee \neg L_k(\overline{y})).$$

Prolog kann die Wahrheit einer Anfrage ψ versuchen abzuleiten, indem die Unerfüllbarkeit von $\neg\psi$ auf Basis der Axiomenmenge Φ gezeigt wird. Im Gegensatz zur Aussagenlogik ist die Erfüllbarkeit oder Nichterfüllbarkeit für allgemeine Formeln $\varphi \in \mathcal{FO}$ aber nicht entscheidbar (siehe Abschnitt 2.1.2).

Da alle beteiligten Formeln in Klauselform vorliegen, liegt für den Unerfüllbarkeitsnachweis ein Resolutionsalgorithmus nahe. Tatsächlich wollen wir die Hornklausel-Eigenschaft der Formeln nutzen, um im Verlauf des nächsten Kapitels einen konkreten Algorithmus herzuleiten.

Die Klauselresolution ist als Verfahren zunächst nur für aussagenlogische Formeln definiert. Es ist aber möglich, die Unerfüllbarkeit prädikatenlogischer Formeln mit gewissen Einschränkungen auf die von aussagenlogischen Formeln zurückzuführen.

Das resultierende Verfahren ist die *prädikatenlogische Resolution.*

2 Prädikatenlogische Resolution

2.1 Erfüllbarkeit und Modelle

2.1.1 Aussagenlogische Erfüllbarkeit

Wir wollen zunächst einen Schritt zurück zur Aussagenlogik machen. Eine aussagenlogische Formel φ heißt *erfüllbar*, wenn es eine erfüllende Belegung $\alpha : \mathit{Var}(\varphi) \to \{\top, \bot\}$ gibt, wobei $\mathit{Var}(\varphi)$ die Menge der (stets frei) in der Formel auftretenden aussagenlogischen Variablen ist. Dann heißt α auch *Modell* von φ.

Es ist auch möglich, diejenigen Variablen, die den Wert „wahr" haben sollen, als Teilmenge $T \subseteq \mathit{Var}(\varphi)$ zu betrachten. Durch diese Teilmenge ist automatisch eine Belegung $\alpha(x) = \top \; \forall x \in T$, $\alpha(x) = \bot \; \forall x \notin T$ definiert. Für ein φ mit n verschiedenen auftretenden Variablen ergeben sich $\|\mathcal{P}(\mathit{Var}(\varphi))\| = 2^n$ mögliche Teilmengen T und damit Belegungen.

Man kann nun diese 2^n Belegungen konstruieren und die in φ auftretenden Variablen für jede Belegung durch ihre zugewiesenen Werte substituieren. Sobald φ für eine Belegung zu „wahr" evaluiert wird, ist φ erfüllbar. Ein Algorithmus zur Bestimmung der Erfüllbarkeit für eine Formel heißt auch *SAT-Algorithmus*, wobei nicht bekannt ist, ob es effizientere Algorithmen gibt als solche mit exponentieller Laufzeit.

Liegt φ jedoch in bestimmten Normalformen vor — wie etwa als Hornformel oder in disjunktiver Normalform (DNF) — verbessert sich die Laufzeit möglicher Erfüllbarkeitstests auf $O(n)$ für eine Formel der Länge n. Anzumerken ist aber, dass die Umformung in DNF selbst bis zu $O(2^n)$ Zeit benötigt und sich nicht jede Formel als Hornformel schreiben lässt, was der Grund dafür ist, dass Prolog-Regeln von vornherein auf solche eingeschränkt sind.

2.1.2 Prädikatenlogische Erfüllbarkeit

Im Gegensatz zur Aussagenlogik ist ein Modell einer Formel der Prädikatenlogik nicht nur eine Belegung der freien Variablen, sondern enthält zusätzlich eine sogenannte *Struktur*[1] oder *Interpretation*.

Definition 2.1. Sei

$$L = (R_1, \dots, R_k, f_1, \dots, f_l, c_1, \dots, c_m)$$

[1] Diese hat jedoch nichts mit dem syntaktischen Element namens *Struktur* in Prolog zu tun.

eine prädikatenlogische Signatur, die die syntaktischen Bestandteile für alle daraus herstellbare Formeln, sogenannte L-Formeln bereitstellt. Dann ist eine Struktur ein Tupel

$$\mathcal{A} = (U, R_1^{\mathcal{A}}, \ldots, R_k^{\mathcal{A}}, f_1^{\mathcal{A}}, \ldots, f_l^{\mathcal{A}}, c_1^{\mathcal{A}}, \ldots, c_m^{\mathcal{A}})$$

und ordnet den syntaktischen Symbolen aus L tatsächliche Relationen, Funktionen und Konstanten zu, das heißt, *interpretiert* diese.

Außerdem enthält eine Struktur eine nichtleere Menge U, die *Grundmenge* oder *Universum* heißt. Freie und quantifizierte Variablen sowie Konstanten in L-Formeln beziehen sich stets auf Elemente aus U, und auch die Relationen und Funktionen mit Stelligkeit k sind Abbildungen $U^k \to \{\top, \bot\}$ beziehungsweise $U^k \to U$.

Bemerkung: Im Folgenden werden Strukturen auch direkt als *Modell* einer Formel bezeichnet, wenn in dieser Formel keine freien Variablen vorkommen.

Bemerkung: Wäre hier die Quantifizierungen auch über Relationen erlaubt, so wären die entstehenden Formeln in der mächtigeren Prädikatenlogik zweiter Stufe ($\mathcal{SO}$). *Pure Prolog* beschränkt sich auf Formeln aus $\mathcal{FO}$, was sich syntaktisch darin widerspiegelt, dass Funktoren nur Atome sein können.

Definition 2.2. Ein Modell $\mathcal{A}$ erfüllt eine L-Formel φ, wenn $\mathcal{A}$ die Signatur L interpretiert und sich φ anhand der semantischen Bedeutungen der Elemente zu „wahr“ evaluieren lässt (siehe [Vol12]). Schreibe dafür $\mathcal{A} \models \varphi$.

Genauso erfüllt ein Modell eine Formelmenge, $\mathcal{A} \models \Phi$, wenn $\mathcal{A} \models \varphi$ für alle $\varphi \in \Phi$ gilt. Schreibe des Weiteren $\Phi \models \varphi$ (bzw. $\psi \models \varphi$), wenn alle Φ (bzw. ψ) erfüllenden Modelle auch φ erfüllen, und nenne φ und ψ *logisch äquivalent*, $\varphi \equiv \psi$, wenn $\varphi \models \psi$ und $\psi \models \varphi$.

Satz 2.3 (Turing, Church, 1936)**.** Es ist nicht entscheidbar, ob eine gegebene Formel $\varphi \in \mathcal{FO}$ ein Modell besitzt, das heißt, ob es ein Modell $\mathcal{A}$ gibt mit $\mathcal{A} \models \varphi$.

Beweis. Ein Beweis, der auf Turings Ansatz basiert, ist in [Hof11] zu finden. Hierfür wird aus einer beliebigen Turingmaschine M eine solche erstellt, deren Band einseitig beschränkt ist, und aus dieser wiederum eine Formel in $\mathcal{FO}$ konstruiert, die genau dann allgemeingültig ist, wenn M auf leerem Band hält. Durch Negation der Formel erhält man aus dem Allgemeingültigkeitsproblem leicht ein Erfüllbarkeitsproblem.

[Sch87] enthält einen weiteren Beweis, der eine Reduktion vom *Postschen Korrespondenzproblem* vornimmt und ohne Turingmaschinen auskommt.

Die Aufgabe des Prolog-Interpreters, der eine Anfrage ψ erhält, ist es dennoch, auf Basis einer Formelmenge Φ zu entscheiden, ob $\Phi \models \psi$. Angenommen, es gilt

$$\Phi \models \psi$$

Daraus folgt nach dem Prinzip des Widerspruchsbeweises:

$$\Leftrightarrow \Phi \cup \{\neg\psi\} \models \bot$$

Da $\bot$ kein Modell haben kann:

$$\Leftrightarrow \Phi \cup \{\neg\psi\} \text{ hat kein Modell.}$$
$$\Leftrightarrow \Phi \cup \{\neg\psi\} \text{ ist unerfüllbar.}$$

Handelt es sich bei der Formelmenge $\Phi \cup \{\neg\psi\}$ wie in Prolog um eine Klauselmenge, so bietet sich ein Resolutionsalgorithmus an. Jedoch ist die Klauselresolution für die Aussagenlogik nicht ohne Weiteres auf die Prädikatenlogik anwendbar, was bereits aus der Unentscheidbarkeit von $\mathcal{FO}$-SAT folgen muss.

Im Folgenden wird aber gezeigt, dass zumindest semi-entscheidbar ist, ob eine Menge prädikatenlogischer Formeln unerfüllbar ist. Der Ansatz ist dabei, eine Formel oder Formelmenge auf eine erfüllbarkeitsäquivalente (möglicherweise unendliche) Menge aussagenlogischer Formeln zurückzuführen.

2.2 Herbrand-Theorie

Ein Algorithmus, der die Erfüllbarkeit einer prädikatenlogischen Formel nachweisen soll, könnte versuchen, ein möglichst einfaches Modell zu finden. Optimal ist ein Modell, das alle Terme der Formel mit sich selbst interpretiert, denn dann sind syntaktische und semantische Auswertung einer Formel identisch. Ein solches Modell heißt auch *Herbrand-Modell* oder *Herbrand-Struktur*. Aus diesem werden dann erfüllbarkeitsäquivalente Formeln der Aussagenlogik konstruiert.

Zunächst sind mehrere Definitionen erforderlich:

Definition 2.4. Eine Formel $\varphi \in \mathcal{FO}$ heißt *bereinigt*, wenn keine freie Variable den gleichen Namen besitzt wie eine gebundene, und wenn alle Quantoren verschiedene Variablen binden.

Definition 2.5. Eine Formel $\varphi \in \mathcal{FO}$ ohne freie Variablen heißt *geschlossen*.

Definition 2.6. Eine Formel $\varphi \in \mathcal{FO}$ ist in *pränexer Normalform* oder *Pränexform*, wenn alle Quantoren vor der eigentlichen Formel stehen. Es gilt also $\varphi = Q_1x_1Q_2x_2 \dots Q_nx_nF$ mit $Q_i \in \{\forall, \exists\}$ und F ist eine quantorenfreie Formel. Nenne $Q_1x_1Q_2x_2 \dots Q_nx_n$ dann *Präfix* und F *Matrix* von φ.

Lemma 2.7. Jede prädikatenlogische Formel lässt sich in eine *bereinigte Pränex-Normalform (BPF)* überführen, die logisch äquivalent zur Ausgangsformel ist. Jedes Prolog-Axiom ist geschlossen und lässt sich daher in eine geschlossene BPF überführen, die logisch äquivalent zum Ausgangsaxiom ist.

Beweis. Umformungen nach [Vol12].

2.2.1 Skolemformen

Nachdem eine Formel $\varphi \in \Phi$ in bereinigter Pränexform vorliegt, kann man versuchen, die Variablen samt Quantoren zu eliminieren, um letztendlich eine aussagenlogische Formel zu erhalten.

Als erstes werden die Existenzquantoren ersetzt. Dies wird auch *Skolemisierung* genannt, die resultierende Formel ist dann eine *Skolemformel.* Grundlage für die Skolemisierung ist die Beobachtung, dass $\exists$-quantifizierte Variablen durch die links von ihnen $\forall$-quantifizierten Variablen funktional bestimmt werden, d.h. von diesen und nur von diesen abhängen.

Beispiel 2.8.
Sei $\mathcal{N} := (\mathbb{N}, <^{\mathcal{N}}, +^{\mathcal{N}}, 0^{\mathcal{N}})$ die Struktur der natürlichen Zahlen mit den üblichen Interpretationen für die Ordnungsrelation und die Addition.

$$\varphi_1 = \forall x \exists y \; x < y$$
$$\varphi_2 = \exists x \forall y \; x < y$$

Dann ist φ_1 sicherlich erfüllt, da jede Zahl kleiner ist als beispielsweise ihr Nachfolger. Jedoch gibt es keine Zahl, die kleiner ist als alle Zahlen (inklusive sich selbst). Somit ist φ_2 (in diesem Modell) nicht erfüllt.

Der Unterschied zwischen den beiden Formeln ist lediglich die Position des Existenzquantors. Eine existenzquantifizierte Variable hängt funktional von allen links von ihr stehenden allquantifizierten Variablen ab. So darf man in φ_1 für jedes x ein y „wählen“, während man sich in φ_2 auf ein x so festlegen muss, dass die Aussage letztendlich allen möglichen Belegungen von y standhält.

Bemerkung: Die Notwendigkeit, mehr funktionale Abhängigkeiten formal zu beschreiben, als durch die bloße Anordnung der Quantoren ausdrückbar ist, ist Grundlage der sogenannten *Dependence Logic*, die Thema des nächsten Kapitels ist.

Die Tatsache, dass jede existenzquantifizierte Variable implizit von den davor positionierten allquantifizierten Variablen abhängt, macht sich die Skolemisierung zunutze. Dabei wird jedes Auftreten einer solchen Variable x durch einen neuen Funktionsterm ersetzt, dessen Argumente genau die Variablen sind, die links von $\exists x$ allquantifiziert wurden.

Definition 2.9. Um eine Formel ψ als *Skolemform* einer bereinigten Pränexformel φ zu erzeugen, ersetze Existenzquantoren wie folgt:

Sei φ eine Formel in bereinigter Pränexform und noch keine Skolemform. Dann ist

$$\varphi = \forall x_1 \forall x_2 \dots \forall x_k \exists y \; F$$

und F ist der Teil von φ hinter der ersten $\exists$-Quantifizierung, $k \geq 0$. (F muss dabei nicht quantorenfrei sein.) Erzeuge nun

$$\psi := \forall x_1 \forall x_2 \dots \forall x_k \; F[y/f_y(x_1, x_2, \dots, x_k)] \; .$$

Die Variable y wurde also substituiert und taucht nun nicht mehr in ψ auf. Wiederhole den Schritt, solange sich ψ nun in der obigen Form (mit einem Existenzquantor) schreiben lässt, ansonsten ist ψ die Skolemform von φ.

$f_y(x_1, x_2, \ldots, x_k)$ heißt hier *Skolemfunktion* der Variable y, im Spezialfall $k = 0$ heißt f_y auch *Skolemkonstante*.

Beispiel 2.10.

$$\varphi \equiv \exists \boldsymbol{x} \forall y \forall z \exists \boldsymbol{w} \; P(\boldsymbol{x}, y) \wedge y \neq \boldsymbol{w} \vee Q(z, \boldsymbol{w}, \boldsymbol{x})$$
$$\psi \equiv \forall y \forall z \; P(\boldsymbol{f_x}, y) \wedge y \neq \boldsymbol{f_w(y, z)} \vee Q(z, \boldsymbol{f_w(y, z)}, \boldsymbol{f_x})$$

Die Skolemform einer Formel ist offensichtlich nicht immer logisch äquivalent zu ihr, da jedes Modell auch die neuen Funktionssymbole und Konstanten interpretieren müsste. Es gilt jedoch

Satz 2.11. Wenn eine Formel φ in bereinigter Pränexform erfüllbar ist, dann ist es auch ihre Skolemform ψ.

Beweis. Der Satz ist bewiesen, wenn φ selbst in Skolemform ist, da ψ und φ bis auf Isomorphie der Skolemsymbole identisch sind. Sei nun φ nicht in Skolemform.

Sei $\mathcal{M} = (U, L^{\mathcal{M}})$ ein Modell für φ mit Grundmenge U. Dann muss es für jede $\exists$-quantifizierte Variable y in φ und für jede Kombination von Belegungen von vorher stehenden, $\forall$-quantifizierten Variablen $x_1, \ldots, x_k$ irgendein Element $u \in U$ geben, durch das y ersetzt werden kann, so dass die Formel nach wie vor erfüllt ist. [2] Man bilde also ein Modell $\mathcal{M}'$ aus $\mathcal{M}$, das für jedes y eine definierte Skolemfunktion f_y besitzt mit $f_y(x_1, \ldots, x_k) = u$ für jeweils passende $u \in U$. Damit ist $\mathcal{M}'$ ein Modell für ψ. □

Korollar 2.12. Wenn eine beliebige Skolemform ψ einer bereinigten Pränexform φ unerfüllbar ist, so ist auch φ unerfüllbar.

2.2.2 Herbrand-Modelle

Für den nächsten Schritt, der Eliminierung der $\forall$-Quantoren, wird zuerst eine besondere Form von Grundmenge benötigt, aus der Belegungen für Variablen und Konstanten geschöpft werden können. Es folgt

Definition 2.13. Das *Herbrand-Universum* $U_H(\varphi)$ einer Formel φ besteht anschaulich aus allen Termen, die mit den in φ verwendeten Funktionssymbolen und Konstantensymbolen konstruierbar sind. Nutze dafür eine rekursive Definition:

Sei C die Menge aller Konstantensymbole und F die Menge aller Funktionssymbole in φ.

[2] Nach Definition von Erfüllbarkeit für Formeln mit Quantoren nach [Vol12].

1. $U_0 = C$, falls $C \neq \emptyset$, sonst $U_0 = \{a\}$. Dabei ist a ein neues Symbol.
2. $U_{k+1} = U_k \cup \{f(u_1, \ldots, u_n) \mid f \in F \text{ ist } n\text{-stellige Funktion}, u_1, \ldots, u_n \in U_k\}$
3. $U_H = \bigcup_{k=0}^{\infty} U_k$

Beispiel 2.14.
Betrachte die Formel

$$\varphi = \forall x \exists y\ P(x) \wedge Q(f(x), g(y))$$

Dann gibt es zwei Funktionssymbole f und g, aber keine Konstanten. Das zu φ gehörende Herbrand-Universum ist dann

$$U_H = \{a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), \ldots\}\,.$$

Bemerkung: $U_H(\varphi)$ ist endlich, wenn φ keine Funktionssymbole enthält, ansonsten abzählbar unendlich.

Existenz des Herbrand-Modells

Für die Konstruktion eines Modells $\mathcal{H}$ mit der Grundmenge U_H werden noch Interpretationen für Konstanten, Funktionen und Relationen benötigt. Wie gefordert gibt es für Konstanten und Funktionen keinen Unterschied zur Syntax:

$$c^{\mathcal{H}} = c$$
$$f^{\mathcal{H}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$$

Definition 2.15. Das Modell $\mathcal{H} = (U_H, R_1^{\mathcal{H}}, \ldots, R_n^{\mathcal{H}}, f_1, \ldots, f_m, c_1, \ldots, c_l)$ mit beliebigen Interpretationen für die Relationen heißt dann *Herbrand-Struktur* oder auch *Herbrand-Modell* von φ, falls zusätzlich $\mathcal{H} \models \varphi$.

Satz 2.16 (Löwenheim, Skolem)**.** Eine geschlossene Formel $\varphi \in \mathcal{FO}$ hat genau dann ein Modell, wenn φ ein Herbrand-Modell hat. Damit hat jede erfüllbare geschlossene Formel insbesondere ein abzählbares Modell.

Beweis (sinngemäß nach [Sch87])**.** Nur „$\Rightarrow$“ ist zu zeigen.

Es habe φ ein Modell und damit auch die Skolemform ψ von φ nach Satz 2.11. Nenne dieses $\mathcal{A}$ und dessen Grundmenge A. Wir konstruieren wie eben eine Herbrand-Struktur $\mathcal{H}$, wählen jedoch ein Element aus A, falls wir eine Konstante einfügen müssen. Gesucht ist nun eine Interpretation der Relationen aus $\mathcal{A}$ in $\mathcal{H}$, so dass $\mathcal{H} \models \psi$.

Sei R eine n-stellige Relation aus $\mathcal{A}$ mit Interpretation $R^{\mathcal{A}}$ und seien die t_i Terme aus U_H. Dann wähle $R^{\mathcal{H}}$ als die Menge mit

$$R^{\mathcal{H}} = \left\{(t_1, t_2, \ldots, t_n) \in (U_H)^n \mid (t_1^{\mathcal{A}}, t_2^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}}) \in R^{\mathcal{A}}\right\}$$

Die Interpretation $R^{\mathcal{H}}$ beschreibt also einfach, dass alle n-Tupel von Termen aus U_H (diese bestehen aus beliebig tief verschachtelten Funktionen und Konstanten, sind aber immer variablenfrei) die Relation R erfüllen, wenn die Formel $R(t_1, t_2, \ldots, t_n)$ auch im Modell $\mathcal{A}$ interpretiert erfüllt ist. Dies ist für ein gegebenes $R(t_1, t_2, \ldots, t_n)$ eindeutig.

Nun ist noch zu zeigen, dass $\mathcal{H} \models \psi$. Dafür führen wir eine Induktion über die Anzahl n der in ψ enthaltenen Quantoren durch. (siehe Anhang A.1) □

Herbrand-Expansion

In einem letzten Schritt sollen nun die Allquantoren eliminiert werden, um die prädikatenlogische Erfüllbarkeit auf die der Aussagenlogik zurückzuführen.

Definition 2.17. Sei $\varphi \equiv \forall x_1 \forall x_2 \ldots \forall x_n\, F$ eine prädikatenlogische Formel in geschlossener Skolemform mit Matrix F. Dann ist die *Herbrand-Expansion* von φ definiert als die Menge

$$\Phi_H(\varphi) := \{F[x_1/t_1][x_2/t_2]\ldots[x_n/t_n] \;\mid t_1, t_2, \ldots, t_n \in U_H(\varphi)\}$$

Anschaulich werden also die Quantoren aus φ entfernt, so dass die x_i zu freien Variablen werden. Sie werden anschließend durch alle möglichen Kombinationen von Termen aus U_H substituiert und bilden eine Menge variablenfreier Formeln, die dadurch wie Formeln aus der Aussagenlogik behandelt werden können.

Bemerkung: Φ_H ist endlich, wenn U_H endlich ist, und abzählbar unendlich, wenn U_H unendlich ist.

Satz 2.18 (Gödel, Herbrand, Skolem)**.** Sei $\varphi \in \mathcal{FO}$. Es existiert genau dann ein Modell für $\Phi_H(\varphi)$, wenn auch eines für φ existiert.

Beweis. Man kann nach Satz 2.11 annehmen, dass φ in Skolemform ist. Man weiß nach Satz 2.16 auch, dass eine erfüllbare Formel φ ein Herbrand-Modell besitzt.

Schreibe $\varphi = \forall x_1 \ldots \forall x_n\, F$ mit quantorenfreiem F. Genau dann, wenn φ erfüllbar ist, gibt es auch ein Herbrand-Modell $\mathcal{H}$ mit $\mathcal{H} \models \varphi$ und es gilt

$$\mathcal{H} \models \forall x_1 \ldots \forall x_n\, F$$

Tauchen freie Variablen auf, muss das Modell diese belegen:

$$\begin{aligned}
&\Leftrightarrow \mathcal{H}[x_1/t_1^{\mathcal{H}}]\ldots[x_n/t_n^{\mathcal{H}}] \models F \;\text{ für alle } t_1, \ldots, t_n \in U_H \\
&\Leftrightarrow \mathcal{H} \models F[x_1/t_1]\ldots[x_n/t_n] \;\text{ für alle } t_1, \ldots, t_n \in U_H \\
&\Leftrightarrow \mathcal{H} \models \psi \text{ für alle } \psi \in \Phi_H \\
&\Leftrightarrow \mathcal{H} \models \Phi_H
\end{aligned}$$

(Zu bemerken ist, dass es sich in den letzten drei Zeilen bereits um variablenfreie Formeln handelt.) □

Der Endlichkeitssatz besagt, dass eine möglicherweise unendliche Formelmenge Φ genau dann unerfüllbar ist, wenn irgendeine endliche Teilmenge von ihr unerfüllbar ist.[Vol12]

Ist Φ eine Menge von Klauseln, kann man einen Resolutionsalgorithmus konstruieren, der terminiert, wenn die leere Klausel ($\square$) abgeleitet wird. Aus dem Endlichkeitssatz folgt, dass es dann eine Resolutionswiderlegung mit endlich vielen Klauseln aus Φ geben muss, wenn Φ unerfüllbar ist.

2.3 Grundresolution

Liegt eine Herbrand-Expansion $\Phi_H(\varphi)$ einer geschlossenen Skolemform φ mit einer Matrix in KNF vor, dann ist $\Phi_H(\varphi)$ eine Klauselmenge. Ein Ansatz für eine Resolutionswiderlegung ist dann, die Klauseln aus Φ_H aufzuzählen (Φ_H ist abzählbar) und nach jedem Schritt alle möglichen Resolventen aus den bisher erfassten Klauseln zu bilden, bis die leere Klausel gefunden wird. Dieses Verfahren heißt auch *Grundresolution*, da alle Klauseln nur variablenfreie Literale enthalten. (Ein variablenfreier Term heißt *Grundterm.*)

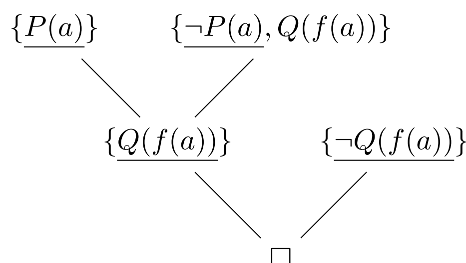
Beispiel 2.19.
Sei

$$\varphi := \forall x\ P(x) \wedge \neg Q(f(x)) \wedge (\neg P(x) \vee Q(f(x)))$$

eine Skolemform mit Matrix in KNF. Dann ist Γ die dazugehörige Herbrand-Expansion. Γ ist eine Klauselmenge. Es ist

$$\Gamma = \{\{P(a)\}, \{\neg Q(f(a))\}, \{\neg P(a), Q(f(a))\}, \{P(f(a))\}, \{\neg Q(f(f(a)))\}, \ldots\}$$

Ein passender Resolutionsgraph ist:



Hier führt bereits die Resolution der ersten drei aufgezählten Klauseln zum Ziel. φ ist also unerfüllbar.

Im Allgemeinen werden bei der Grundresolution aber sehr viele Klauseln erzeugt, die für die Resolutionswiderlegung nicht benötigt werden. Ist Φ_H unendlich und erfüllbar, werden unter Umständen sogar unendlich viele Klauseln erzeugt und der Algorithmus terminiert nicht.[3]

[3] Dies zu umgehen ist auch nicht möglich, da sonst $\mathcal{FO}$-Erfüllbarkeit entscheidbar wäre.

Im Folgenden wird ein Verfahren namens *SLD-Resolution* erläutert, das die Unerfüllbarkeit effizienter zeigen kann. Allerdings ist es nur auf Hornklauseln anwendbar[4] und leider ist auch das Erfüllbarkeitsproblem für prädikatenlogische Hornformelmengen bereits nicht entscheidbar.[MP92]

2.4 Verfeinerung der Resolution

Um auf die teure Grundresolution verzichten zu können, wählt man einen anderen Ansatz: Anstatt bei der Resolution alle Klauseln einer abzählbar unendlichen Klauselmenge Φ zu resolvieren, behält man eine endliche prädikatenlogische Ausgangsklauselmenge Γ (die auch freie Variablen enthalten kann) und resolviert zueinander „passende“ Klauseln aus Γ, wobei Variablen per *Unifikation* ersetzt werden, damit Klauseln „passen“.

Bevor auf einen Algorithmus dafür eingegangen wird, wollen wir uns noch einmal vor Augen führen, welche Arten von prädikatenlogischen Formeln in Prolog überhaupt vorkommen.

Zur Erinnerung: Die Wissensbasis besteht aus Fakten und Regeln in folgender Form:

```
% Eine Regel:
P :- L1, L2, ..., Ln.

% Ein Fakt:
F.
```

Anfragen haben die Form:

```
?- Q1, Q2, ..., Qn.
```

Wobei die `Li`, `P`, `F` und `Qi` Literale sind. Die entsprechenden Formeln lauten:

$$\begin{aligned} \varphi_{Regel} &= \forall\,\overline{x}\;\; (P \leftarrow\; \exists\,\overline{y}\;\; L_1 \wedge L_2 \wedge \ldots \wedge L_n) \\ \varphi_{Fakt} &= \forall\,\overline{x}\;\; F \\ \varphi_{Anfrage} &= \exists\,\overline{y}\;\; (Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n) \end{aligned}$$

Mit aufgelösten Implikationen und negierter Anfrage ist jede Formel eines Prolog-Programms in einer der drei folgenden (Horn-)Formen:

$$\begin{aligned} \varphi_{Regel} &= \forall\,\overline{x}\,\forall\,\overline{y}\;\; (P \vee \neg L_1 \vee \neg L_2 \vee \ldots \neg \vee L_n) \\ \varphi_{Fakt} &= \forall\,\overline{x}\;\; F \\ \neg\varphi_{Anfrage} &= \forall\,\overline{y}\;\; (\neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_n) \end{aligned}$$

[4] Es existiert noch eine allgemeine Variante namens *SL-Resolution*. [Sch87] und [Vol87] zeigen noch weitere effiziente Resolutionsverfahren, von denen einige leider nicht für allgemeine Klauselmengen widerlegungsvollständig sind. Für den eingeschränkten Fall der Hornklauseln sind alle dort vorgestellten Verfahren jedoch gleichwertig.

Hier kommen nur noch geschlossene Skolemformen vor, d. h. die Formeln sind jeweils der ∀-Abschluss ihrer eigenen Matrix. Die Formeln müssen gleichzeitig erfüllt sein, also bildet man die Konjunktion ihrer ∀-Abschlüsse:

$$\varphi := \bigwedge_i \forall\ \overline{x_i} \ \bigvee_j L_{ij}$$

Die Matrizen der Formeln sind alle in KNF und bilden damit eine Klauselmenge. O.B.d.A. lässt sich annehmen, dass alle (nun freien) Variablen in den Klauseln verschiedene Namen haben, da sie vorher nur klausellokal gebunden waren.

$$\Gamma^* := \{\{P, \neg L_1, \neg L_2, \ldots, \neg L_n\}, \{F\}, \{\neg Q_1, \neg Q_2, \ldots, \neg Q_n\}, \ldots\}$$

Definition 2.20. Analog zu bereinigten Formeln sei Γ^* eine *bereinigte prädikatenlogische Klauselmenge* , wenn

- alle $K \in \Gamma^*$ Klauseln prädikatenlogischer Literale (eine atomare Formel oder die Negation einer solchen) sind,
- alle Variablen frei und in höchstens einer Klausel vorkommen.

Die Klauselmenge Γ^* ist nun genau dann mit einer Belegung der freien Variablen erfüllbar, wenn die Konjunktion φ erfüllbar ist. Im Folgenden werden solche Klauselmengen auch kurz mit Γ bezeichnet, wenn aus dem Kontext hervorgeht, dass Γ eine prädikatenlogische Klauselmenge ist.

Hornklauseln, die nun nur aus negativen Literalen bestehen, heißen in der Logikprogrammierung *Zielklauseln*, alle anderen heißen *definite Klauseln*, *Regeln* oder *Programmklauseln.*

Man kann zeigen: Eine geschlossene Skolemform φ mit Matrix in KNF ist genau dann unerfüllbar, wenn sich eine *prädikatenlogische Resolutionswiderlegung* aus Γ^* finden lässt: Dies ist der sogenannte *Resolutionssatz der Prädikatenlogik.*[Sch87]

Eine prädikatenlogische Resolution findet nun statt, indem genau die Klauseln miteinander resolviert werden, die sich durch *Unifikation* in die Form $K_1 = \{L, \ldots\}$ und $K_2 = \{\neg L, \ldots\}$ bringen lassen, das heißt, ein $L \in K_1$ und ein $\neg L \in K_2$ nehmen durch das Ersetzen von Variablen durch Grundterme aus U_H (oder durch andere Variablen) dieselbe Gestalt an.

Auf diese Weise lassen sich genau die für die Resolution *relevanten* Klauseln aus Φ_H nachbilden — oder sogar direkt Literale mit Variablen miteinander resolvieren.

Wir wollen nun zuerst den Begriff der Unifikation formal fassen.

2.4.1 Unifikation

Soll eine Resolution mit der Klauselmenge Γ^* durchgeführt werden, so wählt man in jedem Schritt zwei Literale aus verschiedenen Klauseln aus, die sich *unifizieren* lassen.

Definition 2.21. Ein *Unifikator* θ einer Literalmenge $L = \{L_1, L_2, \ldots, L_n\}$ ist eine Substitution von freien Variablen, für die gilt $L_1\theta = L_2\theta = \ldots = L_n\theta$, die also alle Literale miteinander identifiziert.

$L_i\theta$ ist hierbei das Literal, das aus L_i entsteht, wenn alle Ersetzungen aus θ angewendet werden. $L_i\theta$ heißt auch *Instanz* von L_i. Enthält $L_i\theta$ nur Grundterme und keine Variablen mehr, so spricht man auch von einer *Grundinstanz*.

Definiere analog $K\theta := \{L\theta \mid L \in K\}$ für eine Klausel K.

Ein Unifikator θ heißt *allgemeinster Unifikator* einer Literalmenge, wenn es für jeden Unifikator θ' derselben Literale und eine Substitution σ gibt, so dass $\theta' = \theta \circ \sigma$. Anschaulich ist also θ der „minimale“ Unifikator, der so wenige Variablen wie möglich ersetzt, um die Literale zu unifizieren.

Es gibt auch Mengen von Literalen, die nicht unifizierbar sind; etwa dann, wenn zwei Literale verschieden sind, aber keine Variablen enthalten, so dass nichts ersetzt werden kann.

Wir konstruieren einen konkreten Unifikationsalgorithmus für zwei Literale L_1 und L_2, der sich an eine ähnliche Version in [Par11] anlehnt (siehe Anhang B.1).

Der Algorithmus terminiert entweder mit der Ausgabe 'nicht unifizierbar' oder gibt den allgemeinsten Unifikator zurück. Der sogenannte *occurs check* verhindert, dass eine Variable rekursiv durch einen Term ersetzt wird, der sie selbst enthält.

Satz 2.22 (Robinson, 1965)**.** Jede unifizierbare Menge von Literalen besitzt auch einen allgemeinsten Unifikator.

Beweis. In [Sch87] über die Korrektheit des Unifikationsalgorithmus.

2.4.2 Die SLD-Resolution

Zunächst soll ein einzelner Resolutionsschritt definiert werden.

Definition 2.23 (nach [Sch87])**.** Seien K_1 und K_2 prädikatenlogische Klauseln, das heißt, alle in Literalen vorkommenden Variablen sind frei.

Dann ist jede Klausel R ein *prädikatenlogischer Resolvent* von K_1 und K_2, für die gilt:

- Es gibt Substitutionen σ_1 und σ_2, die lediglich Variablen umbenennen, so dass $K_1\sigma_1$ und $K_2\sigma_2$ keine gleichnamigen Variablen haben. Diese Bedingung entfällt offensichtlich, wenn $\{K_1, K_2\}$ eine bereinigte prädikatenlogische Klauselmenge ist.
- Es gibt nichtleere Mengen von Literalen $A_1 \subseteq K_1\sigma_1$ und $A_2 \subseteq K_2\sigma_2$, so dass $\{\neg L \mid L \in A_1\} \cup A_2$ mit allgemeinstem Unifikator θ unifizierbar ist.[5] Diese Literale werden also miteinander resolviert.
- Dann ist $R = ((K_1\sigma_1 - A_1) \cup (K_2\sigma_2 - A_2))\theta$, d. h. die Menge der übrigen Literale, auf die jedoch auch der Unifikator mit angewendet werden muss.

[5]Die Negation von L ist hier nicht nur als formelsyntaktischer Präfix zu verstehen. Doppelte Negationen sollen aufgelöst werden.

Definition 2.24 (nach [Sch87])**.** Wir schreiben dann für einen Resolutionsschritt in einer Klauselmenge Γ auch:

$$res(\Gamma) = \Gamma \cup \{R \mid R \text{ ist prädikatenlogischer Resolvent von } K_1, K_2 \in \Gamma\}$$

Und für die Menge aller resolvierbaren Klauseln Res^*:

$$\begin{aligned} Res^0(\Gamma) &= \Gamma \\ Res^{k+1}(\Gamma) &= res(Res^k(\Gamma)) \\ Res^*(\Gamma) &= \bigcup_{k=0}^{\infty} Res^k(\Gamma) \end{aligned}$$

Nun kann man den Resolutionssatz der Prädikatenlogik formal definieren.

Satz 2.25 (Resolutionssatz der Prädikatenlogik)**.** Sei φ in geschlossener Skolemform mit Matrix F und sei F in KNF. F ist also eine Klauselmenge. Dann gilt

$$\varphi \text{ ist unerfüllbar } \Leftrightarrow \Box \in Res^*(F)$$

Beweis. Über das sogenannte *Lifting-Lemma*, nachzulesen in [Sch87].

Als nächstes soll das Prinzip des SLD-Algorithmus *(selective linear definite clause resolution)* für eine gegebene Regelmenge Γ erläutert werden. Unerfüllbar soll dabei die Klauselmenge $\Gamma \cup \{\neg Q\}$ sein. Q ist eine Anfrage.

Eine Eigenschaft von manchen Resolutionsverfahren ist die sogenannte *Linearität*. Anschaulich erfolgt eine lineare Resolution auf Basis einer Klausel K als eine Folge $(K_1, K_2, \ldots, K_n)$ von Klauseln, wobei $K_1 = K$ und $K_n = \Box$. Jedes K_i entsteht dann durch Resolution von K_{i-1} mit einem $B \in \Gamma$ oder mit einem K_j, $j < i$.

Manche lineare Resolutionen sind gleichzeitig *Input*-Resolutionen. Diese zeichnen sich dadurch aus, dass bei jedem Resolutionsschritt mindestens eine der Elternklauseln aus der ursprünglichen (Eingabe-)Menge Γ stammen muss. Für allgemeine Klauselmengen sind lineare Verfahren vollständig, Input-Verfahren dagegen nicht mehr. Für Hornklauselmengen sind jedoch beide Verfahren vollständig.[Sch87]

Ist Γ die Regelmenge eines Prolog-Programms, so sind sind alle Klauseln in Γ definite Hornklauseln, enthalten also genau ein positives Literal. Die negierte Anfrage $\neg Q$ ist eine Zielklausel, enthält also nur negative Literale.

Jeder Resolutionsschritt muss dann aus einer definiten Regel $B \in \Gamma$ und einer Zielklausel K eine neue Zielklausel K' erzeugen. K' enthält wieder nur negative Literale, weil bei Resolution einer negativen Klausel mit einer definiten Regel das positive Literal eliminiert werden muss. Diese Art der Resolution ist die SLD-Resolution. Wir erkennen, dass SLD eine Input-Resolution und damit linear ist.

Enthält Q Variablen, so ist man an den Unifikatoren interessiert, die bei der prädikatenlogischen Resolution entstehen.

Wenn für bestimmte Substitutionen θ die Klausel $\neg Q\theta$ widerlegt wurde, so folgt $\Gamma \models Q\theta$ (Satz von Clark).[Sch87]

Die Substitution θ heißt dann *Belegung* oder *Lösung*.

Wir benutzen nun den Ableitungsoperator $\vdash$, um Zielklauseln miteinander in Relation zu setzen. Seien A und B Zielklauseln und Γ eine Menge definiter Hornklauseln. Die Relation

$$(A, \theta) \vdash_\Gamma (B, \theta')$$

gilt dann, falls es ein $D \in \Gamma$ gibt, so dass B prädikatenlogischer Resolvent von A und D mit allgemeinstem Unifikator σ ist, und wenn $\theta \circ \sigma = \theta'$. Nenne Paare (A, θ) auch *Konfiguration*.

Eine Folge von Konfigurationen

$$((K_1, \theta_1), (K_2, \theta_2), \ldots, (K_n, \theta_n))$$

ist dann eine *Widerlegung* von K_1 auf Basis von Γ, wenn $(K_i, \theta_i) \vdash_\Gamma (K_{i+1}, \theta_{i+1})$ für $1 \leq i < n$, $K_n = \square$ sowie $\theta_1 = [\,]$.

Der SLD-Algorithmus durchläuft nun solche Folgen von Konfigurationen und arbeitet dabei wie folgt:

Algorithmus 1 : Allgemeine SLD-Resolution

Daten : Bereinigte prädikatenlogische Klauselmenge $\Gamma = \{B_1, B_2, \ldots, B_n\}$

Prozedur SLD(Q)

Eingabe : Anfrage Q

1 $Z \leftarrow \{(\neg Q, [\,])\}$

2 **while** $Z \neq \emptyset$ **do**

3 Wähle ein $(K, \theta) \in Z$.

4 $Z \leftarrow Z \setminus \{(K, \theta)\}$

5 **if** $K = \square$ **then**

6 **print** „$\Gamma \models Q\theta$“

7 **else**

8 Es ist $K = \{\neg L_1, \neg L_2, \ldots, \neg L_m\}$.

9 **for** $i \leftarrow 1$ **to** n **do**

10 Es ist $B_i = \{L'_1, \neg L'_2, \ldots, \neg L'_k\}$.

11 **if** $\{L_1, L'_1\}$ unifizierbar **then**

`// allgemeinster Unifikator`

12 $\theta^* \leftarrow \text{unify}(L_1, L'_1)$

`// K* ist prädikatenlogischer Resolvent`

13 $K^* \leftarrow ((K \setminus \{\neg L_1\}) \cup (B_i \setminus \{L'_1\}))\,\theta^*$

`// K* ist also abgeleitete Klausel`

14 $Z \leftarrow Z \cup (K^*, \theta \circ \theta^*)$

Man sieht, dass es für ein Literal einer Zielklausel mehrere resolvierbare Regeln geben kann und damit auch mehrere Folgekonfigurationen. Da jede davon zu einer leeren Klausel führen kann, müssen alle verfolgt werden. Die Kette von Konfigurationen wird damit zu einem Baum, dessen Blätter leere oder nicht weiter resolvierbare Klauseln sind. Dieser Konfigurationsgraph ist sogar dann immer ein Baum, also kreisfrei, wenn die Resolution linear ist.

Das konkrete Verhalten des Algorithmus hängt nun davon ab, wie die Operationen auf der Menge Z implementiert werden. Typische Datenstrukturen sind dort *Queues* und *Stacks*.

Wird ersteres verwendet, ergibt sich *SLD-Resolution mit Breitensuche*, da der Konfigurationsbaum in Breitensuche durchlaufen wird.

Das auf Stacks basierende, in der Praxis häufiger verwendete Verfahren ist aber *SLD-Verfahren mit Tiefensuche*. Es generiert im Durchschnitt deutlich weniger zwischengespeicherte Klauseln für eine erfolgreiche Resolution und benötigt bei gleichem Zeitaufwand weniger Speicherplatz, arbeitet also effizienter.

Wird eine Lösung gefunden oder gibt es an einer Stelle keine Möglichkeiten zur Resolution mehr, so setzt bei der Tiefensuche das sogenannte *Backtracking* ein und der Algorithmus springt zurück zur letzten Verzweigungsmöglichkeit. Eine Lösung selbst besteht aus einer Belegung θ der in der Anfrage vorkommenden Variablen.

Beispiel 2.26.
Sei `parent(X, anna), parent(anna, Y).` eine Anfrage. Dann könnte eine mögliche Ausgabe sein:

```
X = bob, Y = alice.
X = adam, Y = eva.
```

Hier gibt es zwei Belegungen θ_1 und θ_2, die den Variablen X und Y jeweils verschiedene Werte zuordnen. Beide Lösungen wurden aus der Axiomenmenge des Prolog-Programms abgeleitet.

Auch eine Grundresolution über der Herbrand-Expansion der Axiomenmenge hätte dann ergeben, dass sich aus der Klauselmenge, die die Klausel

$\neg$`parent(bob, anna)` $\vee$ $\neg$`parent(anna, alice)` oder

$\neg$`parent(adam, anna)` $\vee$ $\neg$`parent(anna, eva)`

enthält, zusammen mit den vorhandenen Regeln ein Widerspruch ableiten lässt.

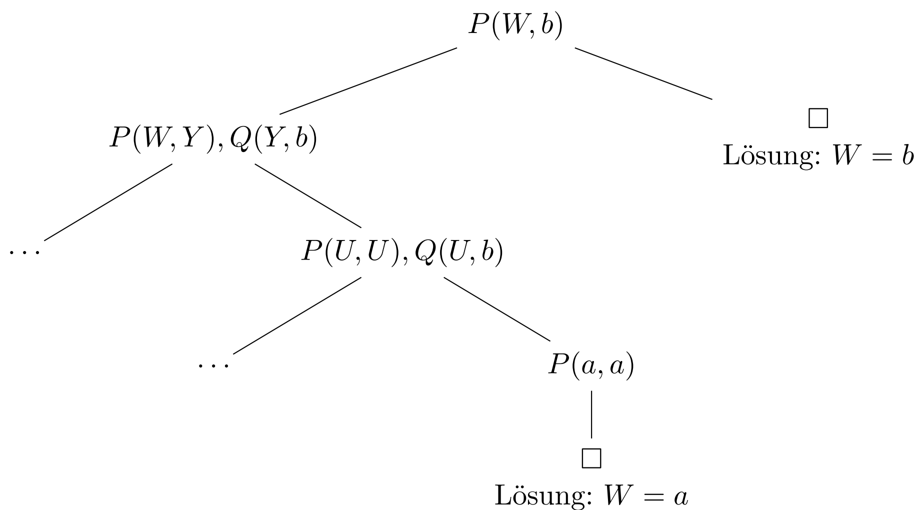
Beim Angeben eines Konfigurationsbaum werden statt der Konfigurationen oft auch einfach die Zielklauseln verwendet, wobei die nicht negierte Form angegeben wird. Es werden dann also direkt die abzuleitenden Aussagen angegeben.

Beispiel 2.27.
Seien folgende Regeln und eine Anfrage gegeben:

```
P(X, Z) :- P(X, Y), Q(Y, Z).
P(U, U).
Q(a, b).

?- P(W, b).
```

In diesem Suchbaum beginnt der Algorithmus mit der Zielklausel $\{\neg P(W, b)\}$. Diese wird mit der ersten Regel zu $\{\neg P(W, Y), \neg Q(Y, Z)\}$ resolviert sowie mit der zweiten Regel direkt zur leeren Klausel $\square$. Der Algorithmus gibt an dieser Stelle die Unifikatoren aus und damit auch die Belegungen der Variablen in der Zielklausel.

$\{\neg P(W, Y), \neg Q(Y, Z)\}$ wiederum wird mit der zweiten Regel zu $\{\neg P(U, U), \neg Q(U, b)\}$, dieses mit der dritten Regel zu $P(a, a)$ und dieses mit der zweiten Regel zu $\square$.



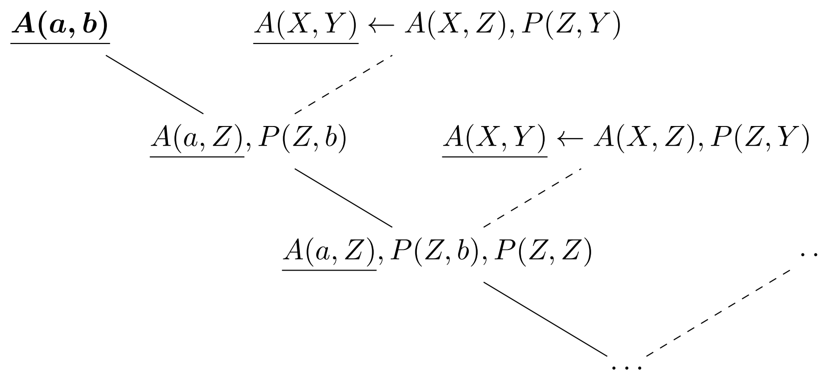
Die Pfade, die links im Baum durch „...“ ersetzt wurden, werden ebenfalls abgesucht. Die erste Regel in der Datenbank verhindert bei ungeschickter Wahl der Reihenfolge jedoch, dass dort jemals eine Lösung gefunden wird.

Allgemein kann es bei Regelmengen mit rekursiven oder zyklischen Regeln sein, dass der Algorithmus in unendlich tiefe Pfade im Suchbaum gerät und somit nicht mehr alle Lösungen ausgibt. Dies gilt nur für die Tiefensuche. Bei der Breitensuche ist offensichtlich, dass alle Lösungen beliebiger Tiefe nach endlicher Zeit ausgegeben werden. Auch dann kann es jedoch passieren, dass der Algorithmus nicht terminiert.

Beispiel 2.28.
Ein weiteres Beispiel, an dem gängige Prolog-Interpreter mit Tiefensuche scheitern:

```
ancestor(X, X).
ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).

?- ancestor(alice, bob).
```

Der dazugehörige Suchbaum, der von der Zielklausel aus nach unten durchlaufen wird:



Abhilfe schafft hier das Vertauschen der Literale im Rumpf der Regel. Dann werden zwar Lösungen ausgegeben, der Algorithmus terminiert jedoch weiterhin im Allgemeinen nicht.

Beispiel 2.29.
Ein häufiges Problem für Tiefensuchen-Algorithmen ist auch eine Definition einer symmetrische Relation:

```
connected(X, Y) :- edge(X, Y).
connected(X, Y) :- connected(Y, X).

?- connected(a, b).
```

Im Fall, dass die `edge`-Relation nicht erfüllt werden kann, wird wieder und wieder die vorhandene Symmetrieregel angewandt und die neue Zielklausel versucht abzuleiten.

Bei der Tiefensuche kommt es also darauf an, in welche Reihenfolge Klauseln für die Resolution ausgewählt werden.[6]

Satz 2.30. Das SLD-Verfahren mit Breitensuche ist für Hornklauseln widerlegungsvollständig und korrekt. Das SLD-Verfahren mit Tiefensuche ist für Hornklauseln korrekt und außerdem widerlegungsvollständig, wenn der Suchbaum keine unendlich tiefen Pfade besitzt.

Beweis. Siehe [Sch87].

Ein Algorithmus muss eine Klausel wählen, wenn mehrere Regeln mit der aktuellen Zielklauseln resolvierbar sind, und er muss innerhalb der Zielklausel das nächste zu resolvierende Literal auswählen. Die Reihenfolge bestimmte eine sogenannte *selection function.*

[6]Ein weiterer Hinweis darauf, dass Prolog keine rein logische, sondern zum Teil auch prozedurale Programmiersprache ist. Ein erfahrener Prolog-Programmierer ist sich der Auswertungsreihenfolge der Klauseln und Literale jedoch bewusst und nutzt diese sogar konsequent aus.

Da diese Resolution linear ist und immer definite Klauseln resolviert werden (es sind in Prolog keine negativen Regelköpfe erlaubt), heißt sie *selective linear definite clause resolution (SLD)*.

Es ist auch eine prozedurale Sicht auf den Resolutionsprozess möglich: Bei Prolog wird immer mit dem Regelkopf resolviert, während der Regelrumpf dadurch Teil der Zielklausel wird. Anschaulich lässt sich sagen, dass Prolog Teilziele (also Literale) nach und nach „abarbeitet“. Es leuchtet ein, dass die leere Zielklausel dann eine erfolgreiche Berechnung bedeutet. Prolog wird jedoch versuchen, alle möglichen Belegungen zu finden.

Daher kann es auch in dem Fall, dass eine Lösung gefunden wurde, zu einer Endlosschleife oder zu einer unendlich duplizierten Ausgabe derselben Lösungen kommen. Dies kann nur mit zusätzlichem Platzaufwand vermieden werden, indem Zwischenergebnisse tabelliert werden — aber auch dies funktioniert nicht in allen Fällen (siehe Kapitel 3).

Damit sind die nötigen theoretischen Grundlagen für Prolog und speziell den SLD-Algorithmus erläutert. Im Anhang B.3 befindet sich eine rekursive Variante des Algorithmus, die den Aufrufstapel für eine Tiefensuche nutzt.

3 Dependence Logic und Teamresolution

3.1 Einführung in Dependence Logic

Die *Dependence Logic (Abhängigkeitslogik)* $\mathcal{D}$ ist eine von Jouko Väänänen geschaffene Erweiterung der Prädikatenlogik erster Stufe, in der sich mehr Abhängigkeiten zwischen auftretenden Variablen ausdrücken lassen. Es zeigt sich, dass $\mathcal{D}$ bereits dadurch ungleich ausdrucksstärker ist als $\mathcal{FO}$.

Dabei wird eine neue Art von Literal bzw. atomarer Formel eingeführt. Neben relationalen Ausdrücken

$$R(t_1, t_2, \dots, t_n)$$

lassen sich sogenannte Abhängigkeitsatome wie folgt schreiben:

$$=(t_1, t_2, \dots, t_n, u)$$
$$dep(t_1, t_2, \dots, t_n, u)$$

Die Semantik ist dabei folgende: Der Term u hängt funktional von den Termen $t_1, t_2, \dots, t_n$ ab, das heißt, es gibt eine Funktion f, die u bestimmt durch

$$f(t_1, t_2, \dots, t_n) = u.$$

Dabei soll im weiteren Verlauf dieser Arbeit die Vereinfachung gelten, dass die t_i und u aus genau einer Variable bestehen.

Wenn sich aus dem Kontext ergibt, dass es sich um eine Abhängigkeit handelt, wird in der Praxis auch $t_1, t_2, \dots, t_n \to u$ geschrieben für „$t_1, t_2, \dots, t_n$ determinieren u“.

Bemerkung: Die obige Aussage ist eigentlich eine Formel in Prädikatenlogik zweiter Stufe — es wird eine Funktion existenzquantifiziert. Genauer gesagt ist diese Formel in $\mathcal{SO}\exists$, einem Fragment der zweitstufigen Logik, in dem Relationen und Funktionen nicht $\forall$-quantifiziert werden dürfen.

Es gilt, dass $\mathcal{D}$ und $\mathcal{SO}\exists$ tatsächlich gleich mächtig in dem Sinne sind, dass es für jede geschlossene Formel der einen Logik eine äquivalente Formel in der anderen gibt und umgekehrt.[Vää07]

Obwohl die Dependence Logic echt ausdrucksstärker ist als die Prädikatenlogik erster Stufe, ist der Endlichkeitssatz hier gültig[Vää07] — im Gegensatz zur vollständigen Prädikatenlogik zweiter Stufe $\mathcal{SO}$. Der Endlichkeitssatz ist fundamentale Voraussetzung für die Anwendung des Resolutionskalküls oder anderer formaler Beweiskalküle.

Beispiel 3.1.
Sei folgende Formel gegeben:

$$\psi_{inf} := \exists z \forall x_0 \exists y_0 \forall x_1 \exists y_1 \; (=(x_1, y_1) \land \neg(y_1 = z) \land (x_0 = x_1 \leftrightarrow y_0 = y_1))$$

Dann besagt ψ_{inf}, dass $y_1 = f(x_1)$ für eine Funktion f (nicht etwa $f(x_0, x_1)$, wie die alleinige Anordnung der Quantoren aussagt) und $y_0 = g(x_0)$ für eine Funktion g. Außerdem gibt es ein z, das verschieden von allen y_1 ist, und zuletzt ist $f = g$ und f bijektiv aufgrund der Teilformel mit Äquivalenz.

ψ_{inf} sagt also aus, dass es eine bijektive Funktion gibt, die von allen Elementen aus der Grundmenge auf eine echte Teilmenge der Grundmenge abbildet. Dies ist jedoch nur für genau die Strukturen mit unendlicher Grundmenge erfüllt. Eine solche Formel ist mit den Mitteln der einfachen Prädikatenlogik erster Stufe nicht ausdrückbar.

3.1.1 Funktionale Abhängigkeiten

Der Begriff der (funktionalen) Abhängigkeit existiert auch in der Datenbanktheorie. Dort wird er im Zusammenhang mit den Attributen einer Relation ($\widehat{=}$ Tabelle) verwendet.

Beispiel 3.2.

Tabelle 3.1: Beispiel

PLZ	Ort	Straße
30167	Hannover	Bachstraße
30167	Hannover	Asternstraße
30625	Hannover	Maneckestraße

In diesem Beispiel existiert die funktionale Abhängigkeit **PLZ → Ort**, das heißt, zu jeder Postleitzahl gibt es genau einen Ort, oder auch: Die Postleitzahl *determiniert* den Ort. Man sagt auch: Die *Ausprägung* der Relation *erfüllt* die funktionale Abhängigkeit **PLZ → Ort**.

Diese Abhängigkeit ist eine, die sich aus Eigenschaften der realen Welt ergibt. Es kann aber auch andere Ausprägungen der Relation, d. h. andere enthaltene Tupel ($\widehat{=}$ Zeilen) statt der vorhandenen geben, die beispielsweise die Abhängigkeit **Ort → Straße** erfüllen, obwohl dies intuitiv sinnlos erscheint.

Eine funktionale Abhängigkeit im Rahmen der Dependence Logic bezieht sich immer auf die Belegung der in einer Formel verwendeten Terme. Die möglichen Belegungen von Variablen stammen dabei aus dem Universum einer bestimmten Struktur.

Beispiel 3.3.
Wir betrachten die Abhängigkeit $=(t_1, t_2, x)$.

Tabelle 3.2: Abhängigkeit erfüllt

t_1	t_2	x
0	1	5
0	1	5
0	2	2
1	2	4
1	2	4

Tabelle 3.3: Abhängigkeit nicht erfüllt

t_1	t_2	x
0	1	5
0	1	5
0	2	2
1	2	**4**
1	2	**2**

Die im Abhängigkeitsausdruck verwendeten Variablen lassen sich als Spalten einer Tabelle auffassen. Eine Zeile einer Tabelle ist dann eine mögliche Belegung freier Variablen einer Formel.

In einer einzelnen Belegung lassen sich jedoch keine Abhängigkeiten oder Unabhängigkeiten feststellen. Alle Abhängigkeiten sind trivialerweise erfüllt. Eine Unterscheidung ist erst möglich, wenn mehrere Belegungen zueinander in den Kontext gesetzt werden.

Eine solche Tabelle heißt auch *Team*, formal eine Menge von (Belegungs-)Funktionen. Im nächsten Abschnitt sollen die für die Abhängigkeitslogik wichtigen Begriffe formal definiert werden.

3.1.2 Teams und Erfüllbarkeit in der Dependence Logic

Vgl. auch [Vää07] für die folgenden Definitionen.

Definition 3.4. Sei $\varphi \in \mathcal{D}$ eine Dependence-Formel, $Fr(\varphi)$ die Menge der freien Variablen in φ und $V \supseteq Fr(\varphi)$. Dann heißt für eine Struktur $\mathcal{A} = (A, L^{\mathcal{A}})$ jede Funktion $s \colon V \to A$ eine *Belegung* für φ über der Struktur $\mathcal{A}$. In dieser Hinsicht besteht kein Unterschied zu einer Belegung in $\mathcal{FO}$.

Nenne den Definitionsbereich V von s auch *Domäne* einer Belegung und den Wertebereich A *Kodomäne*.

Definition 3.5. Eine Menge X heißt *Team* mit Domäne V und Kodomäne A, wenn jedes Element $s \in X$ eine Funktion $s : V \to A$ ist. Ein Team ist also die Erweiterung des Konzeptes der Belegung freier Variablen auf eine Menge von Belegungen, die zusammen ausgewertet werden.

Anschaulich ist jedes Element s eines Teams X also eine Zeile einer Tabelle, während X die ganze Tabelle ist. Jede freie Variable ist eine Spalte, und alle Tabelleneinträge sind Elemente aus der Grundmenge A.

Interpretiert eine Struktur $\mathcal{A}$ einen Term für eine Belegung s, so definiere

$$\begin{aligned} a^{\mathcal{A}}\langle s\rangle &:= a^{\mathcal{A}} \text{ für eine Konstante } a, \\ x^{\mathcal{A}}\langle s\rangle &:= s(x) \text{ für eine freie Variable } x \text{ und} \\ f(t_1,\dots,t_n)^{\mathcal{A}}\langle s\rangle &:= f^{\mathcal{A}}(t_1^{\mathcal{A}}\langle s\rangle,\dots,t_n^{\mathcal{A}}\langle s\rangle) \text{ für eine Funktion } f, \end{aligned}$$

Die freien Variablen eines Terms werden also mit den Werten einer „Zeile" belegt.

Definition 3.6. Sei X ein Team mit Kodomäne A und F eine Funktion mit $F\colon X \to A$. Dann ist das *supplementierende Team* $X(F/x) := \{s(F(s)/x) \mid s \in X\}$, wobei $s(F(s)/x)$ bedeutet, dass s die neue Variable x zusätzlich mit $F(s)$ belegen soll. Anschaulich wird also für alle $s \in X$ eine Spalte x eingefügt, deren Wert zeilenweise von F bestimmt wird.

Definition 3.7. Sei X ein Team und $B \subseteq A$. Dann ist das *duplizierende Team* $X(B/x) := \{s(b/x) \mid s \in X, b \in B\}$, wobei $s(b/x)$ wiederum bedeutet, dass s die neue Variable x mit b belegen soll. Für jedes s und für jedes $b \in B$ wird also s dupliziert, so dass die „Zeile“ s mit allen möglichen Belegungen b für die Spalte x im Team vorhanden ist.

Definition 3.8. Sei $\mathcal{A} = (A, L)$ ein Modell und X ein Team. Dann definiere die Erfüllung einer Formel φ, schreibe auch $\mathcal{A} \models_X \varphi$, wie folgt:

- Falls $\varphi \in \mathcal{FO}$, gilt $\mathcal{A} \models_X \varphi$ genau dann, wenn

$$\forall s \in X : \mathcal{A}[x_1/s(x_1), \ldots, x_n/s(x_n)] \models \varphi$$

  Hier sind $x_1, \ldots, x_n$ freie Variablen in φ. Ein Team erfüllt gewöhnliche $\mathcal{FO}$-Formel also bereits dann, wenn jede Belegung des Teams sie erfüllt.

- $\mathcal{A} \models_X \ \mathrm{dep}(t_1, \ldots t_n, w)$ genau dann, wenn für je zwei Belegungen s, $s' \in X$ gilt:

$$\left(\bigwedge_{i=0}^{n} t_i\langle s\rangle = t_i\langle s'\rangle \right) \to w\langle s\rangle = w\langle s'\rangle$$

- $\mathcal{A} \models_X \neg\,\mathrm{dep}(t_1, \ldots t_n, w)$ genau dann, wenn $X = \emptyset$. Die Wahl dieser Definition kann zunächst verwundern. Die Negation der obigen Definition ergibt:

$$\left(\bigwedge_{i=0}^{n} t_i\langle s\rangle = t_i\langle s'\rangle \right) \wedge w\langle s\rangle \neq w\langle s'\rangle$$

  und ist wegen des Falles $s = s'$ für nichtleere X nicht möglich. Diese Eigenschaft ist wesentlich für die *downward closure* genannte Eigenschaft. (s.u.)

- $\mathcal{A} \models_X \psi_1 \wedge \psi_2$ genau dann, wenn $\mathcal{A} \models_X \psi_1$ und $\mathcal{A} \models_X \psi_2$

- $\mathcal{A} \models_X \psi_1 \vee \psi_2$ genau dann, wenn es Teams Y und Z gibt mit $Y \cup Z = X$ und $\mathcal{A} \models_Y \psi_1$ und $\mathcal{A} \models_Z \psi_2$ — nicht etwa, wenn ein Team ψ_1 oder ψ_2 erfüllt.

- $\mathcal{A} \models_X \exists x\ \psi$ genau dann, wenn es eine Funktion $F\colon X \to A$ gibt, so dass $\mathcal{A} \models_{X(F/x)} \psi$, wenn ψ also von einem beliebigen supplementierenden Team von X erfüllt werden kann.

- $\mathcal{A} \models_X \forall x\ \psi$ genau dann, wenn $\mathcal{A} \models_{X(A/x)} \psi$, wenn ψ also vom duplizierenden Team von X über der Grundmenge A erfüllt werden kann.

Bemerkung: Für eine geschlossene $\mathcal{D}$-Formel φ gilt $\mathcal{A} \models \varphi$ genau dann, wenn $\mathcal{A} \models_{\{\emptyset\}} \varphi$, wenn φ also mit dem Team erfüllt ist, das genau aus der leeren Belegungsfunktion besteht. Anschaulich ist dies möglich, weil nur freie Variablen einer Formel durch ein Team belegt werden.

Achtung: Es gilt $\{\emptyset\} \neq \emptyset$. Das leere Team erfüllt alle Formeln und gleichzeitig ihre Negationen. Dies ist nur ein Beispiel dafür, dass die Erfüllung von Formeln in der Dependence Logic keinem *„tertium non datur“* folgt. Auch erfüllt ein Team nicht unbedingt eine Formel oder ihre Negation. Für einige Formeln φ gibt es unabhängig von der gewählten Struktur $\mathcal{A}$ Teams X derartig, so dass $\mathcal{A} \not\models_X \varphi$ und gleichzeitig $\mathcal{A} \not\models_X \neg\varphi$, selbst wenn φ keine Dependence-Atome enthält.[Vää07]

Satz 3.9 (downward closure)**.** Diese Eigenschaft könnte auch als *Abgeschlossenheit unter Teilmengenbildung* übersetzt werden und besagt, dass

$$\mathcal{M} \models_X \varphi \Rightarrow \mathcal{M} \models_Y \varphi$$

für alle Teams X und alle $Y \subseteq X$.

Beweis. Über obige Definitionen.[Vää07]

3.2 Verträglichkeit mit Prolog

Ziel dieser Arbeit ist das Finden von Ansätzen, Abhängigkeitsausdrücke in Prolog zu definieren und diese algorithmisch auswerten zu lassen, also eine effizient arbeitende Implementierung der Dependence Logic zu schaffen — es sollen Programme nicht mehr nur wie in Prolog aus reinen Hornklauseln bestehen, sondern auch das Beschreiben von funktionalen Abhängigkeiten zulassen.

Die Einbettung von Dependence-Atomen in Hornklauseln ist jedoch nicht trivial. Tatsächlich ist die Abhängigkeitslogik sogar weitestgehend unverträglich mit den Konzepten, auf denen Prolog basiert.

Im Folgenden wird die Logikprogrammierung aus verschiedenen Blickwinkeln betrachtet. Dabei soll aufgezeigt werden, warum sie in der Form von Prolog mit der Abhängigkeitslogik unverträglich ist. Wenn möglich, sollen dennoch einige Lösungsansätze mit eventuellen Einschränkungen festgehalten werden.

3.2.1 Teams in Prolog

Die Aufgabe von Prolog ist letztendlich das mechanische Ableiten des Beweises einer Aussage aus einer vorgegebenen Formelmenge. Dass die Implikation durch eine Formelmenge zumindest semi-entscheidbar ist, folgt aus der Kombination verschiedener Sätze: Denen von Skolem, Löwenheim und Herbrand, dem Endlichkeitssatz sowie dem Gödelschen Vollständigkeitssatz und Resolutionssatz der Prädikatenlogik.

Ein abgeleiteter Beweis hat jeweils die Gestalt einer Resolutionswiderlegung aus Klauselinstanzen, die sich durch Unifikationen aus den zur Verfügung stehenden Klauseln ergeben.

Die Ausgabe des Programms sind schließlich Belegungen von Variablen und damit Unifikatoren. Instanzen der Anfrageklausel, die durch diese Unifikatoren entstehen, sind genau die, die aus der Regel- und Faktenmenge folgen.

Angenommen, die Formelmenge ist nun keine reine prädikatenlogische Klauselmenge mehr, sondern es kommen Abhängigkeitsatome in Formeln vor — wie sollen diese interpretiert werden? Sicherlich können auch Formelmengen aus $\mathcal{D}$ von Belegungen erfüllt oder nicht erfüllt sein. Wenn die Dependence-Logik betrachtet wird, so redet man allerdings automatisch über die Erfüllbarkeit bezogen auf Teams, nicht mehr auf einzelne Belegungen. Jede Dependence-Formel ist aber erfüllbar, denn das leere Team erfüllt alle Formeln.

Die gewöhnlichen Lösungen einer Anfrage, also Belegungen ihrer freien Variablen, entsprechen jeweils einelementigen Teams. Hier bildet jedoch ein Dependence-Atom keine echte Einschränkung, denn für einzelne Belegungen ist jeder (positive) Abhängigkeitsausdruck erfüllt. Demnach müssen mehrere Belegungen, mehrere Lösungen im Kontext betrachtet werden. Erst dann lassen sich funktionale Abhängigkeiten feststellen.

Man sucht also keinen Erfüllbarkeitstest für Dependence-Formeln, sondern einen Algorithmus, der „optimale“ Teams ausgibt, wofür erst definiert werden muss, wann ein Team optimal ist.

Es ist plausibel, dass alle Lösungen, die ein Prolog-Interpreter für eine Anfrage ausgibt, ein Team bilden. Diese Menge von Belegungen lässt sich dann weiter auf Abhängigkeiten untersuchen. Dafür wird jedoch keine Erweiterung der Sprache Prolog benötigt — alle Regeln wären weiterhin Hornformeln, lediglich die Anfrage wird vom Benutzer auf zu untersuchende Dependence-Atome ergänzt.

3.2.2 Erweiterte Hornformeln

Für einen anderen Ansatz als den letzten benötigt man zwangsläufig Abhängigkeitsausdrücke, die sich auf Variablen in den Programmklauseln beziehen und nicht nur auf die Anfrage.

Betrachtet man eine Regel bzw. Implikation $A \leftarrow B$, so könnte man etwa Abhängigkeitsatome in den Teilformeln A oder B platzieren wollen. Hier stört aber, dass die Implikation innerhalb der Dependence Logic undefiniert ist.

Die übliche Äquivalenz

$$A \leftarrow B \equiv A \vee \neg B$$

findet aufgrund der Definition der Disjunktion in $\mathcal{D}$ hier keine Anwendung: Angenommen, $B \equiv B' \wedge \delta$ für ein Dependence-Atom δ. Dann wäre $\neg B \equiv \neg B' \vee \neg\delta$. Die Teilformel $\neg\delta$ wird nun lediglich vom leeren Team erfüllt, während B' sofort von jedem Team X erfüllt wird, dessen Einzelbelegungen $s \in X$ schon B' erfüllen.

Nach Definition der Disjunktion wird $\neg B' \vee \neg\delta$ also von $X \cup \emptyset = X$ erfüllt, wenn jede Einzelbelegung $s \in X$ schon $\neg B'$ erfüllt. Auch ist dann insgesamt $A \vee \neg B' \vee \neg\delta$ von $\emptyset \cup X \cup \emptyset = X$ erfüllt, so dass das Abhängigkeitsatom keine echte Einschränkung ist.

Angenommen, dass stattdessen $A = A' \wedge \delta$, es ist also ein Abhängigkeitsatom Teil der Konklusion. Es muss also $(A' \wedge \delta) \vee \neg B$ erfüllt werden. Auch hier gibt es keine

Verknüpfung der Form „Jedes Team, das B erfüllt, muss auch A erfüllen", denn es reicht für ein Team $X = Y \cup Z$ aus, dass $(A' \wedge \delta)$ vom Teilteam Y erfüllt wird und $\neg B$ vom Teilteam Z.

Wir sehen also, dass wir Regeln mit Abhängigkeitsatomen nicht als klassische Klauseln interpretieren können.

Es existieren Erweiterungen der Dependence Logic, mit denen die Negationen und Implikationen mit der klassischen Bedeutung wieder eingeführt werden, etwa *Team Logic (TLD)* oder Logiken mit *intuitionistischer Implikation (ID)*[1].

- Mit der Wiedereinführung des klassischen Negationsoperators $\sim$ in *TLD* lässt sich zwischen $\mathcal{M} \models_X \neg\varphi$ (alle $s \in X$ erfüllen $\neg\varphi$) und $\mathcal{M} \models_X \sim\varphi$ (nicht alle $s \in X$ erfüllen φ) unterscheiden.

- Die intuitionistische Implikation $\varphi \to \psi$ ist definiert als:

$$\mathcal{M} \models_X \varphi \to \psi \quad \Leftrightarrow \quad \forall Y \subseteq X : \mathcal{M} \models_Y \varphi \to \mathcal{M} \models_Y \psi$$

Man kann jedoch zeigen, dass $\mathcal{D}$ mit jeder Erweiterung um eine klassische Negation genauso ausdrucksstark wird wie $\mathcal{SO}$[2], so dass der (für mechanische Beweisverfahren sehr wichtige) Endlichkeitssatz genau wie der Vollständigkeitssatz für solche Logiken nicht mehr gilt. Dies gilt auch für die Erweiterung um intuitionistische Implikation, sogenannte lineare Implikation oder die klassische Implikation mit Negation.

3.2.3 Resolution in $\mathcal{D}$

Das Resolutionskalkül kann als Verfahren auf Klauseln angewendet werden, in denen Literale positiv und negativ vorkommen. Abhängigkeitsatome sind Literale, aber nicht in diesem Sinne miteinander prädikatenlogisch resolvierbar, da sie keine Relationen darstellen. In Kapitel 2 wurde bereits gezeigt, dass die Erfüllbarkeit einer Formel durch ein Herbrand-Modell nur davon abhängt, welche Interpretationen für dessen Relationen gewählt werden - genau das führt der Resolutionsalgorithmus automatisch durch.

Auch macht es keinen Sinn, Abhängigkeitsatome unifizieren zu wollen, da diese keine Aussagen über Individuen aus dem Universum treffen, sondern über das globale Verhalten der Variablen zueinander. Genauer gesagt: Die Resolution über der Herbrand-Expansion einer Formelmenge macht nur dann Sinn, wenn die einzelnen Grundformeln der Expansion stellvertretend für die ursprüngliche Formel sind. In der Dependence Logic gibt es keine sinnvolle Definition für eine äquivalente Expansion. Während in der gewöhnlichen Prädikatenlogik alle Literale durch Relationssymbol gebildet werden, ist dies in der Dependence Logic nicht mehr der Fall.

Da die Klauselresolution also nicht in der Dependence Logic angewendet werden kann, werden neue Ansätze benötigt.

[1] siehe [Vää07] und [Yan13]

[2] Die Dependence Logic spricht über existenzquantifizierte Funktionen. Man sieht leicht, dass die klassische Negation damit Allquantoren für prädikatenlogische Objekte der zweiten Stufe einführt.

Eine Definition für Dependence-Hornformeln stammt aus [EKMV12]. Das Fragment der Dependence Logic heißt dort *$\mathcal{D}$-Horn* und enthält Formeln der Gestalt

$$\varphi = \forall \overline{x}\ \exists \overline{y}\ (\bigwedge_i \delta_i \wedge \bigwedge_j C_j)$$

wobei δ_i Dependence-Ausdrücke der Form $\delta_i = \mathrm{dep}(\overline{z_i}, w_i)$ sind mit $\overline{z_i} \subseteq \overline{x}$ und $w_i \in \overline{y}$. Außerdem sind die C_j Hornklauseln, besitzen also höchstens ein positives Literal.

Hier sind die Abhängigkeitsatome keine Klauselliterale, sondern beziehen sich nur auf die Variablen vorhandener Literale. Für ein gegebenes Team in Form von Belegungen für Klauselvariablen kann dann effizient überprüft werden, ob es die Hornklauseln und zusätzlich die Abhängigkeiten erfüllt.

Dennoch passt die Gestalt der Formeln nicht zu Prolog-Klauseln; in diesen werden alle Variablen allquantifiziert. Es kann somit gar keine Abhängigkeitsatome geben, in denen existenzquantifizierte Variablen von allquantifizierten Variablen abhängen. Außerdem wurde im letzten Abschnitt bereits gesagt, dass ein Erfüllbarkeitstester nicht direkt eingesetzt werden kann, um optimale Teams zu berechnen. Auch kann die Theorie hinter der prädikatenlogischen Resolution aus den genannten Gründen nicht einfach dazu genutzt werden, die Deduktion von logischen Aussagen auf die zweitstufige Prädikatenlogik zu heben.

3.2.4 Prolog und relationale Algebra

Es ist möglich, die Sprache Prolog so einzuschränken, dass sie auch als Abfragesprache für Datenbanken verwendet werden kann. Die Fakten entsprechen als endliche Relationen dann Datensätzen in Tabellen (*extensionale Datenbanken*). Die Regeln, die die Form echter Implikationen haben, lassen sich dann als *Sichten (Views)* verstehen, die auf vorhandene Tabellen die üblichen relationalen Operatoren wie Vereinigung, Schnitt, horizontaler Verbund, Selektion und Projektion anwenden (*intensionale Datenbanken*). Für all diese Operationen lassen sich äquivalente Prolog-Regeln oder Regelmengen angeben.

Verschiedene eingeschränkte Fragmente von Prolog bilden dann die *Datalog*[CGT89] genannte Sprachfamilie, die Logikprogrammierung verwendet, um *deduktive Datenbanken* zu definieren. Diese zeichnen sich gegenüber der klassischen Relationenalgebra (bzw. SQL) vor allem dadurch aus, dass sie Relationen rekursiv definieren können, bei der Berechnung jedoch immer einen Fixpunkt und damit eine endliche Lösung erreichen.

Da der Begriff der funktionalen Abhängigkeit eigentlich sogar aus der Datenbanktheorie stammt, liegt es nahe, ihn in der relationenalgebraischen Interpretation von Prolog/Datalog zu nutzen und nicht in der Interpretation als Theorembeweiser. Aber auch hier fügen sich funktionale Abhängigkeiten nicht in die Anfragesprache selbst ein: Jede Prolog-Regel definiert eine Relation zeilenweise, d.h. tupelweise. Genauso werden Relationen auch elementweise ausgewertet. Es ist nicht vorgesehen, Ausdrücke über ganze Relationen zu formulieren. Es gibt in ISO-Prolog zwar eingebaute Prädikate wie `bagof/3`, die ganze Mengen von Lösungen aggregieren, jedoch nur für genau eine freie Variable, deren Belegungen als Elemente einer Liste berechnet werden.

Denkbar wäre im Zusammenhang mit Datenbanken das Angeben von funktionalen Abhängigkeiten als *Constraints* für die Ausprägung einer Relation. Diese zu überprüfen ist dann Aufgabe eines Algorithmus, der Tupel nicht berechnet, sondern in eine bestehende Relation einfügt. Einfügeprozeduren fallen jedoch nicht in den Aufgabenbereich einer Anfragesprache wie Datalog.

Mangels globaler Variablen in Prolog kann sich ein Abhängigkeitsatom lediglich auf einzelne Klauseln beziehen. Normalerweise bestimmen die Lösungen eines Regelrumpfes die Interpretation des Prädikats im Regelkopf. Ein Ansatz für eine Implementierung von Dependence-Atomen wäre nun: Werden funktionale Abhängigkeiten als Constraints in einer Klauseldefinition platziert, so könnten sie die Lösungsmengen der Rumpfliterale weiter einschränken und so die Interpretation des Prädikats im Regelkopf beeinflussen.

Wenn die Lösungsmenge einer Anfrage auf funktionale Abhängigkeiten untersucht werden soll, kann die Generierung der Ausgabe nicht Lösung für Lösung wie beim SLD-Algorithmus erfolgen, da für das Auswerten von funktionalen Abhängigkeiten bereits alle Lösungen benötigt werden.

Abhängigkeitsatome in Regeln bewirken sogar, dass für die Berechnung der Lösungen der Klausel erst alle Lösungen der Rumpfliterale bekannt sein müssen, so dass ein Algorithmus, der rekursiv Lösungen berechnet, immer auf kompletten Lösungsmengen operieren muss.

3.3 Implementationsansätze

3.3.1 Einschränkungen der Sprache

Funktionale Abhängigkeiten können zwischen Belegungen von Variablen bestehen. In Prolog kommen Variablen in Klauseln und in der Anfrage vor, so dass etwa die gebildeten Klauselinstanzen oder die Lösungen der Anfrage auf Abhängigkeiten hin untersucht werden können.

Es gibt aber Prolog-Programme und -Anfragen derartig, dass die Menge der Lösungen bzw. der verwendeten Klauselinstanzen unendlich groß ist. Dafür reicht es aus, dass bestimmte rekursive oder zyklische Regeln vorkommen.

Ist eine Menge von Belegungen unendlich groß, so ist offensichtlich nicht allgemein entscheidbar, ob sie eine bestimmte funktionale Abhängigkeit erfüllt. Die erlaubten Prolog-Programme und Anfragen müssen also auf eine Teilmenge so eingeschränkt werden, dass aus ihnen nur endliche Lösungsmengen entstehen können.

Dafür könnten etwa folgende Einschränkungen vereinbart werden:

- Regeln dürfen im Kopfliteral nur Variablen enthalten, die auch in mindestens einem Rumpfliteral vorkommt. Analog dürfen Fakten überhaupt keine Variablen enthalten.
- Es ist nicht möglich, Regeln zyklisch anzuwenden, so dass es keine unendlich tiefen Pfade im Suchbaum gibt.

Die erste Einschränkung sorgt zum einen dafür, dass eine Belegung s immer alle Variablen mit Grundtermen belegen muss — dass also alle Instanzen der Anfrage Grundinstanzen sind, die dadurch auf Abhängigkeiten untersucht werden können — und zum anderen dafür, dass alle Relationen endlich sind.

Die zweite Einschränkung kann etwa über eine totale Ordnungsrelation $<$ zwischen Prädikaten modelliert werden, wobei für ein Prädikat P im Kopf einer Regel und Prädikate $Q_1, \ldots, Q_n$ im Rumpf gelten müsste $P < Q_i$ für $i = 1, \ldots, n$. Da nur endlich viele verschiedene Prädikate vorhanden sind, kann der Suchbaum in diesem Fall nur eine endliche Tiefe haben und die Lösungsmenge damit nur endlich sein.

Diese Begrenzung schränkt die Mächtigkeit der automatischen Beweisführung jedoch ein: Es ist damit nicht mehr möglich, Transitivität und Symmetrie von Relationen zu axiomatisieren. Aus datenbanktheoretischer Sicht lassen sich damit genau die Anfragen der gewöhnlichen Relationenalgebra (ohne Negation) beschreiben, die keine rekursiven Relationen kennt.

Beispiel 3.10.
Seien folgende Regeln gegeben, die einen ungerichteten Graphen beschreiben:

```
edge(a, b).
edge(b, c).
edge(c, d).

% erlaubt
connected(X, Y) :- edge(X, Y).

% Symmetrie - nicht erlaubt
connected(X, Y) :- connected(Y, X).

% Transitivität - nicht erlaubt
connected(X, Y) :- connected(X, Z), connected(Z, Y).
```

In einem Graphen lässt sich berechnen, ob es einen beliebigen Pfad zwischen zwei Knoten gibt. Die Länge eines Pfades in einem beliebigen Graphen ist zwar immer endlich, aber nicht allgemein beschränkt. Während Symmetrieregeln dafür nicht notwendig sind, lässt sich ohne Regeln, die Transitivität ausdrücken, nicht immer ein Pfad zwischen verbundenen Knoten finden.

Ein neuer Ansatz wäre, zu vermeiden, dass eine Regel *mit der gleichen Belegung* der freien Variablen erneut ausgewertet wird, da dieser Fall stets zu einem endlos tiefen Zweig im Suchbaum führen muss.

Wird eine Regel mit dem Kopfprädikat P ausgewertet, so wird überprüft, ob die durch Unifikation gebildete Instanz (die eventuell noch freie Variablen enthält) bereits ausgewertet wurde. Ist dies der Fall, so werden die gespeicherten Teillösungen zurückgegeben.

Dieser Ansatz kann jedoch scheitern, wenn in den Regeln Funktoren vorkommen, die keine Hauptfunktoren sind. Diese entsprechen in der Prädikatenlogik Funktionssymbolen

und führen sofort zu einem unendlichen Herbrand-Universum und damit zu potentiell unendlich vielen Instanzen, die für ein Prädikat gespeichert werden müssten.

Beispiel 3.11.
Nachfolgend eine Regel mit einem Funktionssymbol `f`:

```
p(X) :- p(f(X)).
```

Hier ist $U_H = \{a, f(a), f(f(a)), \ldots\}$ und damit unendlich. Versucht ein Algorithmus, die Anfrage `p(a)` *top-down* zu beantworten, so muss er rekursiv `p(f(a))`, `p(f(f(a))), ...` auswerten und terminiert dabei nicht, obwohl das Prädikat `p` mit jedem möglichen Argument immer nur genau ein mal ausgewertet wird.

Insgesamt muss Prolog also wie folgt eingeschränkt werden:

1. Variablen in Kopfliteralen müssen auch im Rumpf derselben Regel vorkommen.

2. Es gibt keine Strukturen, d.h. zusammengesetzte Terme als Argumente von Prädikaten.

Werden dann rekursive Aufrufe derartig abgefangen, dass jede der endlich vielen Grundinstanzen nur einmal ausgewertet wird, ergeben sich immer endliche Teams sowie eine endliche Laufzeit des Algorithmus.

Vergleicht man die Einschränkungen 1. und 2. mit [CGT89], so fällt auf, dass damit genau die Regelmengen erzeugt werden können, die Datalog entsprechen[3]. Datalog ist mächtiger als etwa die Abfragesprache SQL genau wegen der Fähigkeit, rekursiv genau die transitive Hülle einer Relation zu berechnen.[4]

Aus [CGT89] geht hervor, dass bei Datalog-Implementierungen entweder der *top-down*-Ansatz oder ein *bottom-up*-Ansatz verwendet wird, um eine Zielklausel abzuleiten. Während *top-down* bedeutet, dass die Anfrage vom ursprünglichen Ziel bis hin zur leeren Klausel resolviert wird, erzeugen *bottom-up*-Verfahren rekursiv alle Fakten, die aus der Wissensbasis folgen und verwerfen alle, die keine Instanz der Zielklausel sind.

Es ist offensichtlich, dass beide Verfahren in ihrer naiven Version für die Praxis nicht geeignet sind. Ein Top-Down-Algorithmus wird entweder nicht immer terminieren (Tiefensuche) oder ineffizient sein (Breitensuche). Das Bottom-Up-Verfahren ist noch ineffizienter als die Breitensuche.

Es wurden in der Vergangenheit große Anstrengungen unternommen, um die Effizienz von *top-down* mit der Robustheit von *bottom-up*-Ansätzen zu kombinieren. Hier gibt es sogenannte *Transformationen* der Regelmenge durch beispielsweise *Magic Sets*, um Effizienz und Terminierung zu garantieren, oder verbesserte *top-down*-Verfahren wie *QSQ (query-subquery)*.

[3] Auch hier gibt es syntaktische Erweiterungen, die den Prolog-Strukturen entsprechen und damit Funktionssymbole in die Logik einführen. In Datalog werden solche Programme auch als *unsafe* bezeichnet.

[4] Eigentlich nur, wenn Literale negiert werden können, da sich ohne die Negation von Rumpfliteralen die relationenalgebraische Operation *Mengendifferenz* nicht in Datalog ausdrücken lässt. Steht die Negation zur Verfügung, müssen weitere Vorkehrungen getroffen werden (*Stratifizierung*), damit das Ergebnis einer Anfrage wohldefiniert ist.

3.3.2 Interpretation der Lösungsmenge als Team

Zunächst wollen wir davon ausgehen, dass nur die Menge aller Lösungen einer Anfrage betrachtet werden soll, d. h. die Belegungen der in der Anfrage vorkommenden freien Variablen. Diese bildet dann ein Team, wenn jede Variable mit einem Grundterm belegt wird.

Definition 3.12. Sei Γ eine bereinigte prädikatenlogische Klauselmenge mit der Variablenmenge V.

Dann ist ein Team X eine *Lösungsmenge* für Γ, wenn

$$s \in X \Rightarrow \{K\langle s\rangle \mid K \in \Gamma\} \models \Box$$

für $s\colon V \to U_H$. Schreibe auch $\Gamma\langle s\rangle$ für $\{K\langle s\rangle \mid K \in \Gamma\}$ und schreibe $\Gamma \models_X \Box$ für obige Implikation.

Aus den vorherigen Kapiteln geht hervor, dass eine Belegung freier Variablen in Γ genau dann eine Lösung ist, wenn sich aus den dadurch entstehenden Klauselinstanzen der Widerspruch $\Box$ ableiten lässt.

Das *maximale Team* $X(\Gamma)$ einer Klauselmenge Γ ist dann die Menge *aller* Lösungen s, so dass

$$s \in X \Leftrightarrow \Gamma\langle s\rangle \models \Box$$

gilt. Die Implikation wird also zu einer Äquivalenz. Schreibe $\Gamma \models_X^{max} \Box$, wenn X für Γ maximal ist.

Es ist möglich, dass während der Resolution nicht alle freien Variablen der resolvierten Klauseln mit Termen unifiziert werden müssen, um die leere Klausel zu abzuleiten. Die Belegung dieser Variablen ist dann beliebig. Betrachte eine Lösungsinstanz mit einer einzelnen unbelegten Variable x. Auf Teams bezogen bedeutet dies, dass dann das duplizierende Team $X[U_H/x]$ eine Lösungsmenge für Γ ist, also $\Gamma \models_{X[U_H/x]} \Box$ gilt.

Es gilt für die Kardinalität des duplizierenden Teams, falls x vorher nicht von X belegt wurde:

$$\|X[U_H/x]\| = \|X\| \cdot \|U_H\|$$

Damit kann es aber unendlich große Teams geben, wenn U_H unendlich ist. Abhilfe schafft hier jede der beiden beiden Datalog-Spracheinschränkungen: Durch Einschränkung (1) müssen im Kopf einer Regel vorkommende Variablen auch im Rumpf gebunden sein. Damit bleiben aber während der Resolution eines Literals der Zielklausel mit einem Regelkopf alle Variablen erhalten oder werden durch Grundterme ersetzt. Insgesamt gibt es also nur Lösungen, in denen alle Variablen durch Grundterme ersetzt werden. Einschränkung (2) alleine betrachtet stellt sicher, dass U_H endlich ist.

Bemerkung: Alle Lösungen einer Anfrage zusammengenommen bilden stets ein maximales Team.

Die Definition lässt sich auch über eine Anfrage $\psi = \exists\overline{y}\ L_1 \wedge \dots L_n$ zu einer Axiomenmenge Φ formulieren:

$$s \in X \Leftrightarrow \Phi \models L_1\langle s\rangle \wedge \dots L_n\langle s\rangle$$

Nun können zusammen mit gewöhnlichen Anfragen auch zusätzlich Dependence-Atome ausgewertet werden, die als weitere zu prüfende Formel erst interpretiert werden, wenn alle Lösungen einer Anfrage auf gewohnte Weise ermittelt wurden.

Formal ist ein Team X genau dann die Lösung einer Anfrage mit Dependence-Atomen δ_i, wenn:

$$\Gamma \models^{max}_{X} \Box \;\wedge\; \models_X \bigwedge_i \delta_i$$

X muss also maximales Team für Γ sein und in X müssen die geforderten funktionalen Abhängigkeiten erfüllt sein. Das Modell ist dabei beliebig, da die Abhängigkeitsatome nur freie Variablen enthalten, die durch X belegt werden.

Nun wird der Sinn der Beschränkung auf maximale Teams deutlich: Teilteams könnten die funktionalen Abhängigkeiten vielleicht erfüllen, während ein Benutzer daran interessiert ist, ob die Belegung einer Variable *allgemein* durch eine andere Variable bestimmt wird. Während eine allgemeine Deduktion nicht möglich ist, liefert die Betrachtung möglichst großer Teams einen Nutzen in der Praxis.

Beispiel 3.13.
Ein Beispiel für eine Anfrage mit Abhängigkeit:

```
zone(niedersachsen, norden).
zone(hamburg, norden).
zone(bayern, süden).
wetter(norden, regen).
wetter(süden, sonne).

-? zone(X, Y), wetter(Y, regen) | =(Y).
```

Anschaulich ist hier nach allen Bundesländern X gefragt, die in einer regnerischen Zone liegen. Dabei soll die Zone Y durch eine 0-stellige Funktion bestimmt werden und damit konstant sein, das heißt, es dürfen keine zwei regnerischen Bundesländer gefunden werden, die in verschiedenen Zonen liegen.

Mit einer Änderung der letzten Regel zu

```
wetter(süden, regen).
```

würde sich das Ergebnis der Anfrage von

```
X = niedersachsen, Y = norden;
X = hamburg, Y = norden;
yes.
```

zu

```
no.
```

für „keine Lösung“ ändern, da die funktionale Abhängigkeit nicht mehr erfüllt wäre.

Für eine endliche Belegungsmenge ist ein solches Prüfen von Dependence-Atomen effizient möglich. Sei $\delta = \mathrm{dep}(\overline{x}, w)$ ein Dependence-Atom. w soll also funktional von den Variablen $\overline{x} = x_1, \ldots, x_n$ abhängen.

Dann wird dies von folgendem Algorithmus überprüft:

Algorithmus 2 : Überprüfung eines Abhängigkeitsausdrucks δ

```
Prozedur checkDependency(X, δ)
    Eingabe : Team X, Atom δ = dep(x_1, ..., x_n, w)
1   forall the s ∈ X do
2       forall the s' ∈ X do
3           match ← true;
4           for i ← 1 to n do
5               if s(x_i) ≠ s'(x_i) then
6                   match ← false;
7                   break;
8           if match then
9               if s(w) ≠ s'(w) then
10                  return false ;
11  return true
```

Eine Prolog/Datalog-Anfrage besteht aus einer Zielklausel und beliebig vielen Abhängigkeitsatomen. Der Algorithmus kann dann genutzt werden, um nacheinander alle geforderten Abhängigkeiten zu überprüfen. Ist das Ergebnis $false$, so ist das gesamte Team keine Lösung für die Anfrage. Beim Ergebnis $true$ ist die Menge der Einzellösungen für die Zielklausel gleichzeitig ein Team, das die Abhängigkeitsatome erfüllt.

3.3.3 Binäre Teamresolution

Im letzten Abschnitt wurde die Menge aller Lösungen als Team betrachtet, um die in der Lösungsmenge enthaltenden Variablen auf funktionale Abhängigkeiten zu untersuchen. Dieser Ansatz lässt sich in dem Sinne verallgemeinern, dass das rekursive Ableiten der Zielklausel durch andere Klauseln ebenfalls um Überprüfungen auf funktionale Abhängigkeiten ergänzt werden kann.

Dabei wird nicht nur die Anfrage um eine Menge von Dependence-Atomen erweitert, sondern potenziell auch jede andere in einem Prolog-Programm vorkommende Gruppe von Zielklauseln. Im Rumpf einer Regel würde dies interpretiert als: „Wird mit dem Kopf dieser Regel unifiziert, so versuche alle Lösungen für den Rumpf herzuleiten. Betrachte dann die Unifikation mit dem Regelkopf nur dann als erfolgreich, wenn die gefundene Lösungsmenge zusätzlich die Abhängigkeiten erfüllt.“

Betrachte die Klausel

$$A \leftarrow B_1 \wedge \ldots \wedge B_n$$

mit den Literalen A und B_1 bis B_n, die freie Variablen enthalten können. Dann ist für jede Substitution θ der Ausdruck $A\theta$ eine Instanz von A.

Sei $A\theta$ nun eine Grundinstanz von A, θ soll also alle Variablen in A ersetzen. Da θ aus Unifikationen entsteht, haben alle Substitutionen in θ die Form $[x/t]$ für eine Variable x und einen Grundterm t.

Dadurch ist θ äquivalent zu einer Funktion s_θ im folgenden Sinne: Es gibt ein s_θ mit $s_\theta(x) = t$ für genau die x und t, für die θ die Substitution $[x/t]$ enthält. Dabei ist t ein Grundterm, in Datalog demnach immer eine Konstante.

Man kann nun Mengen von Substitutionen als Teams zusammenfassen und fordern, dass die enthaltenen Belegungen bestimmten funktionalen Abhängigkeiten genügen. Wir wollen diese Abhängigkeiten wie oben angekündigt zusammen mit Regeln notieren.

Dazu fassen wir Regeln K nicht mehr als einzelne Klauseln auf, sondern fügen noch eine Menge von Abhängigkeitsatomen hinzu. Es ist

$$K = (C, \Delta)$$

mit einer prädikatenlogischen Hornklausel C und Dependence-Atomen $\Delta = \{\delta_1, \ldots, \delta_m\}$, $m \geq 0$.

Eine Konfiguration besteht dann aus einer Zielklausel C, relevanten Abhängigkeitsatomen sowie dem aktuellen Unifikator θ:

$$(C, \Delta, \theta)$$

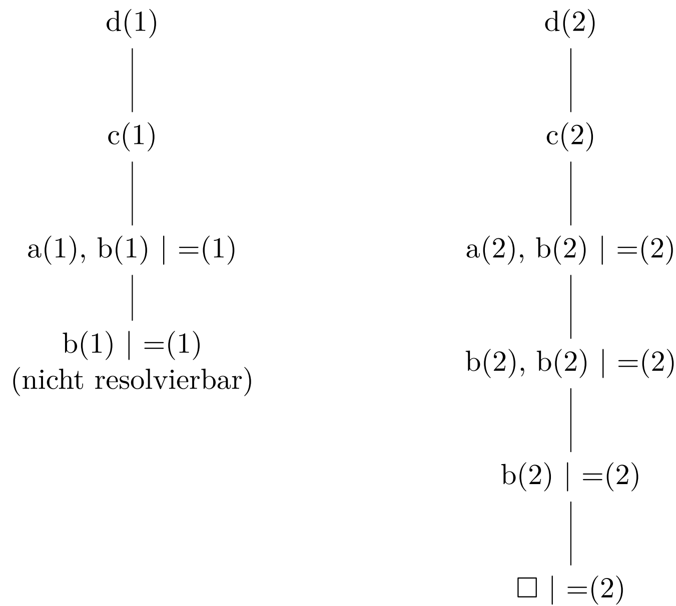
Obige Interpretation könnte man aus prozeduraler Sicht auch als *subtree cutting* bezeichnen: Soll eine Konfiguration (C, Δ, θ) abgeleitet werden, so kann es mehrere Möglichkeiten geben, alle Literale in C zu resolvieren und zur leeren Klausel zu gelangen. Dabei können sich verschiedene Unifikatoren ergeben. Diese sollen dann allen δ-Atomen genügen. Diese möglichen Pfade zur leeren Klausel sind in dem Teil des Suchbaums enthalten, der sich unter der Konfiguration (C, Δ, θ) aufspannt, und der als andere Blätter lediglich unresolvierbare Klauseln enthält.

Ist nun ein δ-Atom aus der Menge Δ unerfüllt, werden alle Lösungen innerhalb des Teilbaums unterhalb der Konfiguration (C, Δ, θ) ungültig, der Teilbaum wird damit abgeschnitten. C wird als „nicht weiter resolvierbar“ gewertet. Damit ähnelt der Mechanismus dem *cut* in Prolog.[ISO96]

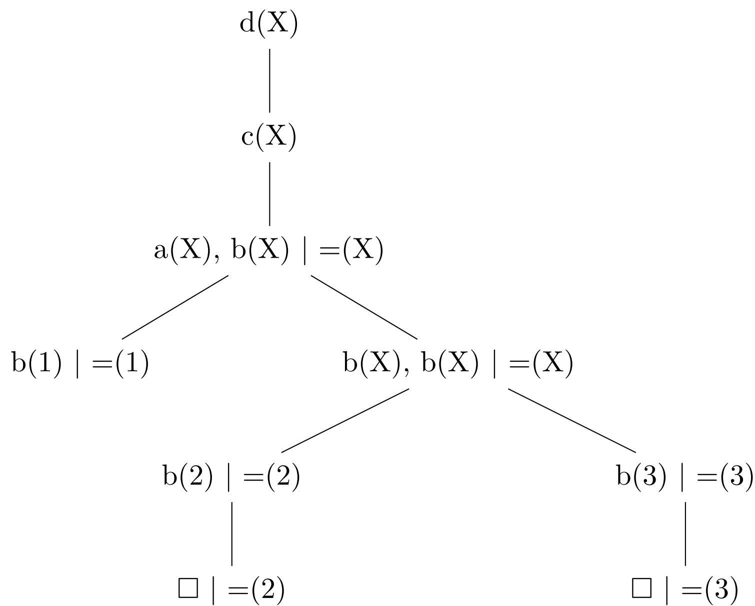
Beispiel 3.14.
Betrachte eine Regelmenge, die wie folgt gegeben ist:

```
a(1).
a(X) :- b(X).
b(2).
b(3).
c(X) :- a(X), b(X) | =(X).
d(Y) :- c(Y).
```

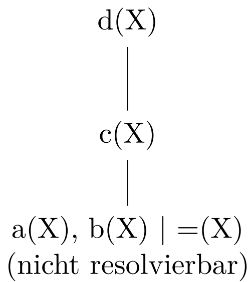
Die zugehörigen Resolutionsbäume für die Anfragen `d(1)` und `d(2)` sind:



Die allgemeine Anfrage `d(X)` führt jedoch wieder zu keiner Lösung:



Die Variable x wird also im Teilbaum unterhalb des Dependence-Atoms mit verschiedenen Variablen belegt. Das Team $\{x \mapsto 2, x \mapsto 3\}$ erfüllt aber $=(x)$ nicht. Damit wird der Teilbaum unter der Klausel `a(X), b(X) | =(X)` nicht weiter betrachtet und der Baum kollabiert, ohne dass eine weitere Lösung für die Anfrage gefunden wird, zu:



Bemerkung: Diese Art der Teamresolution heißt *binär*, da ein Team X eine Dependence-Menge Δ entweder erfüllt oder nicht erfüllt. Im negativen Fall kann es aber trotzdem Teilmengen $Y \subset X$ geben, die Δ erfüllen. Möglich wäre es, die Lösungen im abgeschnittenen Teilbaum nicht zu löschen, sondern das Team auf ein optimales Teilteam so zu reduzieren, dass Δ erfüllt ist. Hierbei wird jedoch ein Maß für die „Güte“ eines Teams benötigt, wie etwa die Anzahl der Belegungen.

Das *subtree cutting* lässt sich mit geringen Modifikationen gut in den Tiefensuchen-SLD-Algorithmus einbetten.
Das Schema ist folgendes:

- Die Ausgabe erfolgt verzögert - Lösungen werden per Tiefensuche gesammelt und erst beim Backtracking ausgegeben, wenn wieder die Baumwurzel erreicht wird.
- Es werden alle δ-Atome einer Regel mit unifiziert, d. h. die Variablen substituiert.
- Beim Erreichen der leeren Klausel enthalten die Variablen eines δ-Atoms bestimmte Terme. Diese werden für dieses δ tabelliert und bilden eine einzelne Belegung eines Teams, dessen Domäne die in δ vorkommenden Variablen sind.
- Beim Backtracking werden gefundene Lösungen beibehalten, so dass an Verzweigungen im Baum mehrelementige Teams entstehen können. Für diese werden die δ-Atome überprüft, ggf. wird der Teilbaum mit allen enthaltenen Lösungen abgeschnitten.
- Wird beim Backtracking die Wurzel erreicht, werden alle übrigen Lösungen ausgegeben.

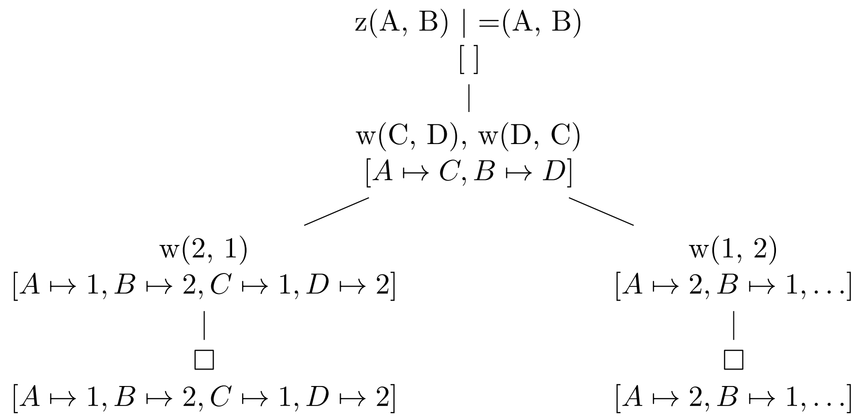
Wir wollen an dieser Stelle die Datalog-Einschränkungen annehmen. Dependence-Atome können dann zu jedem Zeitpunkt der Tiefensuche leicht überprüft werden, da in ihnen enthaltene Variablen Teilmenge der Domäne der bis dahin gesammelten Substitutionen sein müssen und sowohl die Domäne als auch die Menge der Substitutionen endlich ist.

Beispiel 3.15.
Folgendes Beispiel zeigt, dass die Substitutionen, die auf Blattebene gewonnen werden, unverändert übernommen werden können, um die Abhängigkeitsprüfungen durchzuführen.

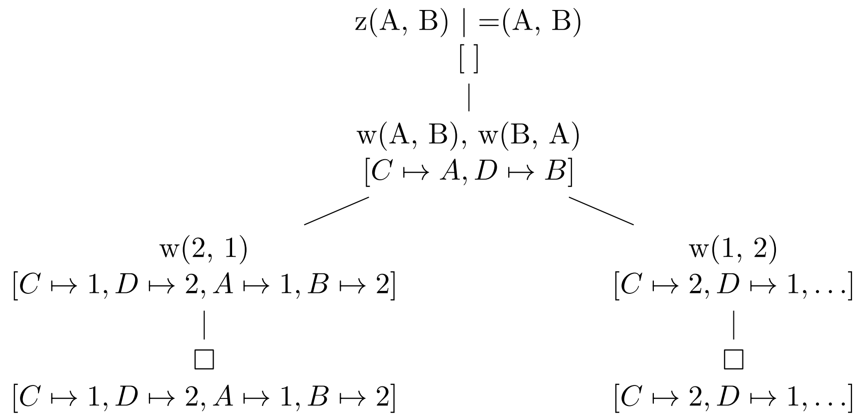
```
w(1, 2).
w(2, 1).
z(C, D) :- w(C, D), w(D, C) | =(D).

?- z(A, B) | =(A, B).
```

Es folgt der zugehörige Suchbaum, in dem auch die Substitutionen sichtbar sind. Bei der Unifikation kann es passieren, dass zwei Variablen miteinander verglichen werden. In diesem Fall ist nicht festgelegt, welche der Variablen durch die andere ersetzt wird. Dies spielt jedoch für das Tabellieren der Belegungen eines Teams keine Rolle.

Fall 1: Es werden stets die Variablen der Zielklausel substituiert.



Fall 2: Es werden stets die Variablen der Programmklausel substituiert.



Es ist klar, dass die Substitutionen in den Blättern des Baums immer auch Zuweisungen für alle Variablen der Zwischenkonfigurationen beinhalten, somit auch für die Variablen in den Dependence-Atomen.

Daraus folgt der SLD-Algorithmus mit Tiefensuche und subtree-cutting, der im Anhang B.5 zu finden ist.

Die Funktion `shrink`, die im Algorithmus verwendet wird, passt ein Team X an, das eine Abhängigkeit δ nicht erfüllt. Dies kann jede Funktion sein, die X so weit wie notwendig verkleinert, dass es das jeweilige δ erfüllt. Im Fall der binären Teamresolution definieren wir:

$$\text{shrink}(X, \delta) = \begin{cases} X & \text{falls } \models_X \delta \\ \emptyset & \text{sonst} \end{cases}$$

Es sind aber auch Definitionen möglich, die selektiv bestimmte Belegungen aus dem Team entfernen, damit δ erfüllt wird.

4 Entwurf eines Interpreters für Dependence-Prolog

Im Rahmen dieser Arbeit soll ein Interpreter für einen Prolog-Dialekt mit Dependence-Atomen implementiert werden. In den Beispielen des letzten Kapitels wurde bereits angedeutet, welche Erweiterung der Syntax dabei vorgenommen wird.

Die Klauseln haben folgende Form:

```
P(...) :- L1(...), L2(...) | =(...), =(...).
```

Die Dependence-Atome werden nicht einfach per `,` mit den anderen Literalen verbunden, da das Komma in Prolog eine Konjunktion darstellt, wir aber keine Formeln der zweitstufigen Logik betrachten wollen. Stattdessen soll das Symbol `|` genutzt werden, das die Bedeutung „wird gefiltert von" hat.

Das *Pipe*-Symbol `|` entspricht in unixoiden Betriebssystemen einer Weiterleitung einer Ausgabe eines Programms in die Eingabe eines anderen Programms, oft tatsächlich zum Zweck der Filterung oder Weiterverarbeitung der Ausgaben.

In der prozeduralen Interpretation von Prolog, bei der Regelanwendungen Aufrufe von Unterprogrammen sind, bewirkt die Pipe mit Dependence-Atomen ein analoges Verhalten.

Ein wesentlicher Teil eines Interpreters ist der enthaltene *Parser (Zerleger)*. Parser werden meist auf Basis einer formalen Grammatik entworfen. Sie haben die Aufgabe, die Konstruktion einer Symbolfolge aus der Grammatik „rückgängig" nachzuvollziehen und auszugeben, etwa als Folge von *Ableitungen*.

Die grundlegenden Definitionen für diese und weitere Begriffe aus den Grundlagen der Compilertheorie sind in [Par11] und [ASU86] zu finden und werden im Folgenden als bekannt vorausgesetzt. Wir werden in folgenden Abschnitten eine kontextfreie Grammatik für Dependence-Prolog sehen und einen einfachen Parser dafür entwerfen.

Oft wird das *Parsen* eines Programms eigentlich von zwei getrennten Komponenten übernommen, die *lexikalische* und die *syntaktische* Analyse.

1. Ein *Lexer* oder *lexikalischer Scanner* erhält eine Zeichenkette in einer plattformabhängigen Codierung. Diese wird in Teilzeichenketten, die *Lexeme* zerlegt, beispielsweise über reguläre Ausdrücke (*regular expressions*), und diese in logische Blöcke, sogenannte *Tokens* transformiert. Auch werden Kommentare und Leerräume entfernt. Aus dem Lexem erhält ein Token auch seinen *Tokenwert*, wie etwa den Wert einer als Zeichenkette codierten Zahl, der unabhängig von der logischen Information „Zahl" (der Token-Klasse) ist.

2. Der eigentliche syntaktische *Parser* wandelt die Folge von Tokens ausgehend von Produktionsregeln einer Grammatik in eine bestimmte Datenstruktur um, bei kontextfreien Grammatiken üblicherweise ein Ableitungsbaum oder Syntaxbaum.

Der getrennte Entwurf dieser Komponenten besitzt mehrere Vorteile:

- Der tendenziell komplexe Parser kann unabhängig von der Eingabecodierung geschrieben werden und muss durch die Reduzierung auf Tokens ein sehr viel kleineres logisches Alphabet verarbeiten; er muss auch nicht mit Leerräumen oder Kommentaren umgehen können.
- Die dazugehörige Grammatik besitzt analog dazu nur noch Tokens als Terminalsymbole und muss deren genaue Darstellung (wie etwa Verkettung von Ziffern oder Buchstaben zu einer Zahl bzw. einem Wort) nicht modellieren.
- Bereits vor dem eigentlichen Parsen kann festgestellt werden, ob ein Zeichen überhaupt zu einem gültigen Token gehört. Die lexikalische Analyse kann somit eine eigene Stufe der Fehlerbehandlung erhalten, während die Fehlerbehandlung der syntaktischen Analyse sich immer auf ganze Tokens beziehen kann.

Nachdem ein Quellprogramm durch einen Parser aufbereitet wurde, etwa in einen Syntaxbaum, kann das Ergebnis weiterverarbeitet werden. Der auf das Parsen folgende Schritt ist die *semantische Analyse*.

4.1 Lexikalische Analyse

Der *Scanner* soll auf Basis von mehreren regulären Ausdrücken eine Zeichenkette in Tokens zerlegen.

Definition 4.1. Das *Rechneralphabet* Σ_R ist die Menge aller Symbole, die ein gegebener Rechner mit fester Wortbreite d darstellen kann. In der Praxis ist $|\Sigma_R| = 2^d$, wobei nicht alle Bitmuster ein sinnvolles Zeichen codieren. Beispiele hierfür sind ASCII, UCS oder UTF-32.[1] Wir wollen Σ_R im Weiteren nur als große, aber endliche Menge von Symbolen betrachten.

Wir schreiben außerdem kurz C_a für die Menge der Kleinbuchstaben in Σ_R und C_A analog für Großbuchstaben, C_1 für Ziffernsymbole und C_S für die Menge der Weißzeichen und Umbrüche.

Sei außerdem $C_\#$ die Menge, die die Sonderzeichen `, ; . | ( ) =` enthält.

Sei $C = C_a \cup C_A \cup C_1 \cup \{\ _\ \}$.

Im Rückblick auf Kapitel 1 können wir nun jede lexikalische Einheit durch einen regulären Ausdruck darstellen.

[1] Codierungen mit variabler Wortbreite wie UTF-8 erfordern bereits eine Gliederung der Symbole in logische Einheiten, die *code points*.

Atome: Jede Zeichenkette w, die mit einem Kleinbuchstaben beginnt, nur aus Ziffern besteht oder in Anführungszeichen (`' '`) gesetzt ist. Der dazugehörige reguläre Ausdruck ist $C_aC^* \mid C_1^+ \mid '(C|(''))^*'$.

Variablen: $w \in (C_A \mid _\,)C^*$

Konnektoren: Die Symbole $w \in C_\#$ entsprechen selbst regulären Ausdrücken und damit jeweils einem eigenen Token, das Wort `:-` für die Implikation ist ein weiterer regulärer Ausdruck.

Redundante Anteile sind leere Zeichenketten: $w \in C_S^+$. Auch alle Kommentare, also alle Zeichen zwischen `%` bzw. `//` und einem Zeilenumbruch werden übersprungen, ebenso Zeichen zwischen `/*` und dem nächsten darauf folgenden `*/`.

Eine Zeichenkette, die einem der regulären Ausdrücke genügt, wird vom Scanner in das entsprechende Token verwandelt.

Definition 4.2. Das *Tokenalphabet* oder *logische Alphabet* Ω enthält die möglichen Tokenklassen der Sprache und ist definiert als

$$\Omega := \{\omega_a \mid a \in C_\#\} \cup \{\omega_{Var}, \omega_{Atom}, \omega_{:-}, \omega_{Leer}\}.$$

Ein Scanner wird nun auf Basis der regulären Ausdrücke der einzelnen Tokenklassen erstellt. Scanner dieser Art simulieren direkt einen endlichen Automaten. Sie lesen Zeichen, solange noch Übergänge gefunden werden, und springen anschließend zum letzten erreichten Endzustand zurück, um ein vollständiges Token zu erkennen. Die Nummer des Endzustands steht dann für eine bestimmte Tokenklasse. Eine Zeichenkette w aus einer Sprache A kann auch Präfixe besitzen, die in A liegen, etwa bei Zahlen oder Variablennamen. Dann ist die Zerlegung in Tokens nicht eindeutig. Durch das Verfolgen von Übergängen, bis kein weiterer Zustand mehr erreichbar ist, wird eine Eindeutigkeit wiederhergestellt. Es sollen also maximale Präfixe als Tokens gesucht werden.

Sei nun $\omega \in \Omega$ eine der obigen Tokenklassen und $M_\omega = (Z_\omega, C_\omega, \delta_\omega, z_{0,\omega}, E_\omega)$ ein minimaler DEA, der den dafür definierten regulären Ausdruck akzeptiert. Sei $C_\omega \subseteq \Sigma_R$ der Teil des Alphabets, der im regulären Ausdruck vorkommt. Da M_ω minimal ist, gibt es genau einen Fehlerzustand, d.h. Zustand ohne erreichbare Endzustände. Dieser Fehlerzustand sei z_{error}. Es soll gelten

$$Z_\omega \cap Z_{\omega'} = \{z_{error}\}$$

für alle verschiedenen $\omega, \omega' \in \Omega$.

Man kann die Sprachen dann zusammenfassen, indem ein neuer Startzustand per ε-Übergang in alle alten Startzustände übergeht. Dieser NEA wird mit einer Potenzmengenkonstruktion determiniert.

Für obige Menge regulärer Sprachen gilt eine besondere Eigenschaft:

Definition 4.3. Nenne eine Menge A von Sprachen *präfixdisjunkt*, wenn für alle $L \in A$ und beliebige Wörter $uv \in L$ gilt $u \notin L'$ für alle $L' \in A, L' \neq L$.

Diese Eigenschaft garantiert, dass während der Umwandlung des NEAs in einen DEA jeder deterministische Zustand bis auf den Startzustand nur aus einem einzelnen nichtdeterministischen Zustand hervorgeht. Anschaulich ist dies der Fall, weil alle Übergänge aus den Startzuständen verschiedene Zeichen lesen oder im Fehlerfall im selben Zustand z_{error} münden.

Wenn der neue Startzustand mit z_s bezeichnet wird, so ist $M = (Z, \Sigma_R, \delta, z_0, E)$ der neue DEA, der sich direkt angeben lässt:

$$
\begin{aligned}
Z &:= (\bigcup_{\omega \in \Omega} Z_\omega) \cup \{z_s\} \\
\delta &:= \bigcup_{\omega \in \Omega} (\delta_\omega \cup \{(z_s, a) \to z \mid (z_{0,\omega}, a) \to z \in \delta_\omega\}) \\
E &:= \bigcup_{\omega \in \Omega} E_\omega
\end{aligned}
$$

Dieser DEA kann nun simuliert werden, um eine Zeichenkette in Tokens zu zerlegen (siehe Anhang B.6).

4.2 Syntaktische Analyse

Nachdem ein Programm in eine Tokenfolge $\omega_1, \ldots, \omega_n$ umgewandelt wurde, kann darauf ein Parser operieren.

Zunächst wird eine kontextfreie Grammatik für die Sprache benötigt, die als terminales Alphabet Ω benutzt. Folgende Produktionsmenge P erzeugt genau Dependence-Prolog-Programme:

$$
\begin{array}{lll}
\text{(Programm)} & S & \to \varepsilon \mid KS \\
\text{(Klausel)} & K & \to LK' \\
 & K' & \to \omega_{.} \mid \omega_{:-}\ R\,\omega_{.} \\
\text{(Rumpf)} & R & \to LR' \\
 & R' & \to \varepsilon \mid \omega_{,}LR' \mid \omega_{;}LR' \mid \omega_{|}\ D \\
\text{(Literal)} & L & \to \omega_{Atom}L' \\
 & L' & \to \varepsilon \mid \omega_{(}\ P\ \omega_{)} \\
\text{(Term)} & T & \to \omega_{Atom}T' \mid \omega_{Var} \\
 & T' & \to \varepsilon \mid \omega_{(}\ P\ \omega_{)} \\
\text{(Parameterliste)} & P & \to TP' \\
 & P' & \to \varepsilon \mid \omega_{,}\ TP' \\
\text{(Abhängigkeiten)} & D & \to \varepsilon \mid \omega_{=}\ \omega_{(}\ X\ \omega_{)}\ D' \\
 & D' & \to \varepsilon \mid \omega_{,}\ \omega_{=}\ \omega_{(}\ X\ \omega_{)}\ D' \\
\text{(Variablen)} & X & \to \omega_{Var}\ X' \\
 & X' & \to \varepsilon \mid \omega_{,}\ \omega_{Var}\ X' \\
\text{(Anfrage)} & A & \to R\,\omega_{.}
\end{array}
$$

Mit $V = \{S, K, K', R, R', L, L', T, T', P, P', D, D', X, X', A\}$ ist $G = (V, \Omega, P, S)$ dann nicht nur eine kontextfreie Grammatik, sondern sogar vom Typ der *LL(1)*-Grammatiken und damit besonders einfach zu parsen.

Dass dies eine LL(1)-Grammatik ist, zeigen wir über die sogenannten *First-* und *Follow-Mengen*, die wir mit Algorithmen aus [Par11] berechnen.

First- und Follow-Mengen

$V_\varepsilon \subseteq V$ ist die Menge aller Nichtterminale, aus denen (transitiv) das leere Wort abgeleitet werden kann. Diese werden für den weiteren Verlauf benötigt. Es ist $V_\varepsilon = \{S, R', L', T', P', D, D', X'\}$.

Es ergeben sich folgende First- und Follow-Mengen aus der Grammatik:

N	**First(N)**	**Follow(N)**
S	$\varepsilon, \omega_{Atom}$	$\$$
K	ω_{Atom}	$\omega_{Atom}, \$$
K'	$\omega_{.}, \omega_{:-}$	$\omega_{Atom}, \$$
R	ω_{Atom}	$\omega_{.}$
R'	$\varepsilon, \omega_{,}, \omega_{;}, \omega_{\vert}$	$\omega_{.}$
L	ω_{Atom}	$\omega_{.}, \omega_{,}, \omega_{;}, \omega_{:-}, \omega_{\vert}$
L'	$\varepsilon, \omega_{(}$	$\omega_{.}, \omega_{,}, \omega_{;}, \omega_{:-}, \omega_{\vert}$
T	$\omega_{Atom}, \omega_{Var}$	$\omega_{)}, \omega_{,}$
T'	$\varepsilon, \omega_{(}$	$\omega_{)}, \omega_{,}$
P	$\omega_{Atom}, \omega_{Var}$	$\omega_{)}$
P'	$\varepsilon, \omega_{,}$	$\omega_{)}$
D	$\varepsilon, \omega_{=}$	$\omega_{.}$
D'	$\varepsilon, \omega_{,}$	$\omega_{.}$
X	ω_{Var}	$\omega_{)}$
X'	$\varepsilon, \omega_{,}$	$\omega_{)}$
A	ω_{Atom}	

N	$\boldsymbol{\alpha}$ mit $N \to \alpha \in P$	**First**($\boldsymbol{\alpha}$)
$N \in V_\varepsilon$	ε	ε
S	KS	ω_{Atom}
K	LK'	ω_{Atom}
K'	$\omega_{.}$	$\omega_{.}$
K'	$\omega_{:-}\ R\ \omega_{.}$	$\omega_{:-}$
R	LR'	ω_{Atom}
R'	$\omega_{,}\ LR'$	$\omega_{,}$
R'	$\omega_{;}\ LR'$	$\omega_{;}$
R'	$\omega_{\vert}\ D$	$\omega_{\vert}$
L	$\omega_{Atom}\ L'$	ω_{Atom}
L', T'	$\omega_{(}\ P\ \omega_{)}$	$\omega_{(}$
T	$\omega_{Atom}\ T'$	ω_{Atom}
T	ω_{Var}	ω_{Var}
P	TP'	$\omega_{Atom}, \omega_{Var}$
P'	$\omega_{,}\ TP'$	$\omega_{,}$
D	$\omega_{=}\ \omega_{(}\ X\ \omega_{)}\ D'$	$\omega_{=}$
D'	$\omega_{,}\ \omega_{=}\ \omega_{(}\ X\ \omega_{)}\ D'$	$\omega_{,}$
X	$\omega_{Var}\ X'$	ω_{Var}
X'	$\omega_{,}\ \omega_{Var}\ X'$	$\omega_{,}$
A	$R\ \omega_{.}$	ω_{Atom}

Dabei ist $\$$ das Symbol für das Wortende.

LL(1)-Eigenschaft

Für alle Nichtterminale $N \in V$ muss nun gelten, dass alle rechten Seiten α_i von Produktionen $N \to \alpha_i$ disjunkte First-Mengen haben, damit ein Parser, der nur das nächste Zeichen sieht, deterministisch bestimmen kann, welche der Produktionen von N angewendet wird.

Dies ist in der obigen Grammatik der Fall.

Ist ein $\alpha_i = \varepsilon$, so kann N „gelöscht“ werden. Dann muss zusätzlich

$$\text{Follow}(N) \cap \text{First}(\alpha_j) = \emptyset$$

gelten für alle anderen rechten Seiten α_j von Produktionen $N \to \alpha_j$. Erst dann ist eindeutig, ob N gelöscht oder durch eine nichtleere rechte Seite ersetzt werden muss.

Auch dieses ist für die obige Grammatik erfüllt.

Die Grammatik (V, Σ, P, S) erzeugt Programme, die Grammatik (V, Σ, P, A) erzeugt Anfragen. Letztere Grammatik ist auch vom Typ LL(1), es wandert lediglich das $-Zeichen in die Follow-Menge von A. Im Weiteren beziehen wir uns nur auf S als Startsymbol, alle Aussagen treffen jedoch auch auf die Grammatik zu, die Anfragen über das Startsymbol A verarbeitet.

Bemerkung: Wenn eine kontextfreie Sprache $A \in \mathcal{L}_{LL(1)}$ eine LL(1)-Sprache ist, also von einer LL(1)-Grammatik erzeugt werden kann, dann gibt es einen deterministischen Kellerautomaten, der sie akzeptiert. Dafür muss sie jedoch nicht notwendigerweise LL(1) sein, es gilt:

$$\mathcal{L}_{LL(1)} \subsetneq \mathcal{L}_{LR(1)} = \mathcal{L}_{DKA} \subsetneq \mathcal{L}_{NKA} = \mathcal{L}_2$$

Sogenannte LR(1)-Parser können alle deterministisch entscheidbaren kontextfreien Sprachen (auch DCFL genannt) verarbeiten, die eine echte Obermenge der LL(1)-Sprachen bilden. Selbst sind sie eine echte Untermenge aller kontextfreien Sprachen (CFL), die die Typ 2-Sprachen der Chomsky-Hierarchie bilden.

Implementierbar sind also bereits Parser für LR(1)-Sprachen, diese sind jedoch deutlich aufwendiger zu konstruieren als LL(1)-Parser, so dass man dafür gerne auf Werkzeuge wie *yacc*[2] zurückgreift.

Tabellengesteuerter Top-Down-LL(1)-Parser

Für einen LL(1)-Parser kann man ausgehend von den First- und Follow-Mengen nun eine Parsing-Tabelle konstruieren. Eine solche Tabelle erlaubt letztendlich, eine gelesene Eingabe Schritt für Schritt *top-down* in den dazugehörigen Ableitungsbaum zurückzuführen. Dafür werden die anwendbaren Produktionsregeln durchnummeriert:

(1) $S \rightarrow \varepsilon$
(2) $S \rightarrow KS$
(3) $K \rightarrow LK'$
(4) $K' \rightarrow \omega_{.}$
(5) $K' \rightarrow \omega_{:-}\ R\ \omega_{.}$
(6) $R \rightarrow LR'$
(7) $R' \rightarrow \varepsilon$
(8) $R' \rightarrow \omega_{,}LR'$
(9) $R' \rightarrow \omega_{;}LR'$
(10) $R' \rightarrow \omega_{|}\ D$
(11) $L \rightarrow \omega_{Atom}L'$
(12) $L' \rightarrow \varepsilon$
(13) $L' \rightarrow \omega_{(}\ P\ \omega_{)}$
(14) $T \rightarrow \omega_{Atom}T'$
(15) $T \rightarrow \omega_{Var}$
(16) $T' \rightarrow \varepsilon$
(17) $T' \rightarrow \omega_{(}\ P\ \omega_{)}$
(18) $P \rightarrow TP'$
(19) $P' \rightarrow \varepsilon$
(20) $P' \rightarrow \omega_{,}\ TP'$
(21) $D \rightarrow \varepsilon$
(22) $D \rightarrow \omega_{=}\ \omega_{(}\ X\ \omega_{)}\ D'$
(23) $D' \rightarrow \varepsilon$
(24) $D' \rightarrow \omega_{,}\ \omega_{=}\ \omega_{(}\ X\ \omega_{)}\ D'$
(25) $X \rightarrow \omega_{Var}\ X'$
(26) $X' \rightarrow \varepsilon$
(27) $X' \rightarrow \omega_{,}\ \omega_{Var}\ X'$
(28) $A \rightarrow R\ \omega_{.}$

[2] Yacc: http://dinosaur.compilertools.net/

Nun kann man schließlich die Parsing-Tabelle angeben:

N	$\omega_.$	$\omega_,$	$\omega_;$	$\omega_{:-}$	$\omega_{\vert}$	$\omega_=$	$\omega_($	$\omega_)$	ω_{Atom}	ω_{Var}	$\$$
S									2		1
K									3		
K'	4			5							
R									6		
R'	7	8	9		10						
L									11		
L'	12	12	12	12	12		13				
T									14	15	
T'		16					17	16			
P									18	18	
P'		20						19			
D	21					22					
D'	23	24									
X										25	
X'		27						26			
A									28		

Ein tabellengesteuerter Algorithmus verwendet nun üblicherweise einen Stack, der anfangs mit dem Startsymbol S gefüllt ist, und kann alle Terminalzeichen der Eingabe nacheinander lesen. Dabei werden Nichtterminale auf dem Stack anhand von Produktionen ersetzt (das aktuelle Eingabezeichen, auch *Lookahead* genannt, determiniert die Produktion anhand der Tabelle) und Terminale schlicht vom Stack entfernt, wenn das gleiche Zeichen zu diesem Zeitpunkt das Lookahead ist.

Im Fehlerfall ist das Terminalsymbol auf dem Stack ungleich dem Lookahead, oder letzteres führt zu keiner anwendbaren Produktion. Der Parser kann dann entweder abbrechen oder Eingabeteile überspringen, um zumindest noch den hinteren Teil der Eingabe zu verarbeiten.

An dieser Stelle halten wir uns grob an den Top-down-Stack-Parser aus [Par11] und [ASU86], werden diesen aber modifizieren, da er zwar die syntaktische Gültigkeit eines Programms entscheidet, aber dabei keinen sogenannten *Ableitungsbaum* berechnet. Jeder innere Knoten in einem Ableitungsbaum enthält ein Nichtterminal als Wert und die es ersetzende rechte Seite als Kindknoten. Blätter sind Terminalsymbole oder ε.

Anstatt Terminale und Nichtterminale auf dem Stack zu speichern, sollen dort Referenzen auf Knoten des Baumes liegen, so dass ein gültiger Ableitungsbaum erstellt wird. Der komplette Algorithmus ist als Pseudocode in Anhang B.7 zu finden.

4.3 Weiteres Vorgehen bei der Implementierung

4.3.1 Semantische Analyse und Übersetzung

Der Ableitungsbaum selbst enthält noch mehr Informationen, als für die Weiterverarbeitung der Programmstruktur notwendig ist: Zum einen, weil einige Tokenklassen nichts zur Semantik des Programms beitragen, weil sie nur als Trennzeichen o.Ä. dienen, zum anderen, weil manche Knoten zusammengefasst werden können. Obige Grammatik besitzt in den rechten Seiten einer Regel etwa höchstens zwei Nichtterminale, obwohl beispielsweise eine Klausel mehr als zwei Literale enthält. Der dadurch eigentlich zu tiefe Syntaxbaum kann somit vereinfacht werden.

Den Ableitungsbaum in einen *abstrakten Syntaxbaum* zu verwandeln, der nur noch wesentliche Informationen enthält, ist Aufgabe der semantischen Analyse, ebenso wie das Überprüfen der semantischen Gültigkeit des Programms. [ASU86] Beispiel: Jede benutzte Variable muss deklariert sein, oder in einer Zuweisung müssen Datentypen kompatibel sein. Viele solcher Zusammenhänge sind nicht in einer kontextfreien Grammatik ausdrückbar, so dass die Trennung zwischen Parser und semantischer Analyse überhaupt erforderlich wird.

Für die Implementierung von Dependence-Prolog soll der Baum minimiert werden, anschließend wird jedes Prädikat im Kopf einer Regel zusammen mit seiner Stelligkeit indiziert, so dass für ein gegebenes Literal effizient alle möglichen Unifikationspartner gefunden werden. Auch werden Variablen und Atome durchnummeriert und die Bezeichner tabelliert, da Operationen auf Ganzzahlen günstiger sind als Operationen auf Zeichenketten. Auf diesen Datenstrukturen kann dann der in Kapitel 3 beschriebene erweiterte SLD-Algorithmus mit Tiefensuche ausgeführt werden.

4.3.2 Hinweise zu Prolog/Datalog

Es wird SLD mit Tiefensuche implementiert, ohne Ergebnisse bei der Suche zwischenzuspeichern. Es sind Lösungsmengen zwar durch die Datalog-Einschränkungen immer endlich, aber der Algorithmus kann in eine Endlosschleife geraten und identische Lösungen immer wieder ausgeben.

Bei Tiefensuche reicht es nicht aus, die gefundenen Grundinstanzen zwischenzuspeichern, da es auch dann zu einer zyklischer Auswertung kommen kann, wenn die beteiligten Literale noch freie Variablen beinhalten. Vielmehr muss jedes zu resolvierende Prädikat (d.h. jedes Teilziel) tabelliert werden. Ein *SLG* genanntes Deduktionsverfahren von Chen und Warren[CW95] führt diese Tabellierung durch und arbeitet dabei effizient und top-down. Gespeicherte Teilziele werden dabei in Abhängigkeitsgraphen eingepflegt, um Zyklen erkennen zu können, und korrekt aufgelöst.

In dieser Arbeit beschränkt sich die Deduktion aus Gründen der Einfachheit jedoch auf die in Kapitel 2 beschriebene SLD-Resolution. Die Grammatik, die für die erweiterte Sprache entworfen wurde, lässt zusammengesetzte Terme wie in Prolog zu, da die Terminierung bei SLD zum einen ohnehin nicht sichergestellt ist, und da die Implementierung so zum anderen leichter erweitert werden kann.

4.4 Bedienung des Programms

Built-in-Prädikate

Der Interpreter kennt einige vordefinierte Prädikate, die nicht mit derselben Stelligkeit als Kopf einer Regel verwendet werden dürfen. Dies zu überprüfen gehört ebenfalls zur semantischen Analyse.

Diese Prädikate sind: `consult/1, quit/0` und `write/1`.

Das `consult`-Prädikat kann Teil von Benutzeranfragen sein und den Namen einer Prolog-Datei (die Endung `.pl` wird automatisch ergänzt) enthalten. Die genannte Datei wird dann eingelesen und enthaltene Regeln der Wissensbasis hinzugefügt.

Das Prädikat `write` dient der Textausgabe auf der Standardausgabe des Systems, `quit` beendet das Programm.

ISO-Prolog und bekannte Implementierungen enthalten etwa weitere 100 eingebaute Prädikate[ISO96], viele für arithmetische Ausdrücke, Datei- und Streammanipulation, dynamisches Ändern von Regeln, Listenverarbeitung, Überprüfungen auf Datentypen von Variablenbelegungen und die Negation von Literalen.

Kommandozeile

Die Bedienung des Interpreters erfolgt über die Kommandozeile. Das Programm kann mit verschiedenen Befehlszeilenoptionen gestartet werden, nämlich:

- `-v` - „version“ — Ausgabe der Versionsnummer und des Erstellungsdatums
- `-h` - „help“ — Ausgabe der Hilfe.
- `-c <datei>` - „consult“ — falls eine Regelbasis zum Programmstart aus einer Datei geladen werden soll.
- `-d` - „debug“ — aktiviere Debug-Ausgaben.

Hinweis zu Kommentaren

Der lexikalische Scanner des Interpreters ist so implementiert, dass für Blockkommentare `/* ... */` die Schachtelungstiefe d mitgezählt wird. Dadurch sind Kommentare nicht mehr durch reguläre Ausdrücke darstellbar. Dieses Problem wird so gelöst, dass der entworfene Automat für die Tiefe $d = 0$ genutzt wird, für alle $d > 0$ stattdessen aber nur noch die Kommentar-Tokens gezählt werden.

Diese Vorgehensweise weicht von der beschriebenen Theorie ab. Der Vorteil verschachtelbarer Blockkommentare ist aber, dass ganze Abschnitte des Programms auskommentiert werden können, die selbst noch Blockkommentare enthalten, etwa Dokumentation für Regeln.

Hinweis zur Ausgabe

Ursprünglich erfolgt die Ausgabe in Prolog Lösung für Lösung, per Eingabe von `;` fordert der Benutzer das Programm auf, die nächste Lösung auszugeben. Tatsächlich wird die Tiefensuche dabei angehalten, bis eine weitere Lösung angefordert wird. Für die Teamresolution werden Lösungen mengenweise berechnet, so dass am Ende alle Lösungen auf einmal vorliegen. Gerät die Resolution in eine Endlosschleife, so wird keine Lösung ausgegeben.

Hinweise zu Disjunktionen

Während in ISO-Prolog sowie in der erstellten LL(1)-Grammatik Disjunktionen von Literalen existieren, werden diese im Teamresolutionsalgorithmus nicht weiter behandelt. In Standard-Prolog werden Disjunktionen aufgelöst, indem dasselbe Prädikat mit mehreren Rümpfen, einmal je Disjunktionsglied, in der Datenbank gespeichert wird. Sind jedoch auch Abhängigkeitsausdrücke vorhanden, wird unklar, auf welches Disjunktionsglied diese sich beziehen. Sinnvoll ist die Vereinigung der Lösungsmengen der Disjunktionsglieder mit anschließender Abhängigkeitenprüfung auf der Vereinigung. Diese Abweichung vom vereinfachten Pseudocode wurde auch in der Implementierung gewählt.

5 Abgrenzung und Fazit

Der in dieser Arbeit entwickelte Interpreter erfüllt die Aufgabe, Abhängigkeitsausdrücke in Hornformeln effizient zu verarbeiten — in dem Sinne, dass der Algorithmus trotz mehr Platzbedarf nicht wesentlich langsamer als Original-Prolog ist.

Die Wahl der Semantik für Abhängigkeitsausdrücke entspricht keinen echten Formeln aus $\mathcal{D}$. Es wurde jedoch gezeigt, warum eine echte Erweiterung der Ausdrucksstärke der beteiligten logischen Formeln in Prolog zu vermeiden war.

Klauseln in Prolog können aus verschiedenen Gründen nicht auf das Fragment $\mathcal{D}^*$-Horn aus [EKMV12] zurückgeführt werden, das ursprünglich der Anlass dafür war, nach einer effizienten Implementierung zu suchen. Auch wird in [EKMV12] nur gezeigt, dass das *model checking*, also das Überprüfen von $(\mathcal{A} \models \varphi)$ für feste Modelle $\mathcal{A}$ und Formeln φ in polynomieller Zeit läuft. Damit ist jedoch noch kein Ansatz gefunden, um das Implikationsproblem für Dependence-Formeln zu (semi-)entscheiden.

Die theoretischen Grundlagen für prädikatenlogische Resolution in Prolog wurden in dieser Arbeit im Detail hergeleitet. Während sich die meisten Dokumente zu Prolog auf die Programmierung konzentrieren, war [Sch87] eine hervorragende Quelle, die die Hintergründe ausführlicher darlegt.

Sehr interessant ist auch die Verwendung der Logikprogrammierung für Datenbanksprachen wie Datalog. Obwohl Datalog in der Öffentlichkeit bisher keine Bekanntheit erlangt hat, ist es immer noch aktuelles Forschungsfeld, zuletzt für die Verwendung in der parallelen Datenverarbeitung in großen *Cloud*-Netzwerken.[1]

Die Algorithmen, die erarbeitet wurden, um Datalog-Anfragen effizient, aber trotzdem immer terminierend auszuwerten, hätten im Rahmen dieser Arbeit ebenfalls großen Nutzen, da Lösungen hier verzögert ausgegeben werden. Deren Implementierung ist aber komplex und erfordert wie in Kapitel 4 beschrieben noch zusätzliche theoretische Grundlagen, die über die einfache Logikprogrammierung hinausgehen.

Ein ausgelassenes Detail ist also die Unvollständigkeit der naiven Tiefensuche, selbst mit Datalog-Einschränkungen der Sprache. Auch ist noch zu zeigen, wie nützlich die *subtree-cutting*-Semantik in der Praxis ist. Während funktionale Abhängigkeiten in der Datenbanktheorie ein bekannter Begriff sind, bleibt zu hinterfragen, welche Anwendungsmöglichkeiten sie bei automatischer Beweisführung wie in Prolog bieten.

Interessant wäre ein vollständiger Kalkül, der das Implikationsproblem für die Dependence Logic $(\Phi \models \varphi\ ?)$ löst. Die Resolution ist dafür offensichtlich nicht geeignet, nach [Vää07] ist das Unerfüllbarkeitsproblem für $\mathcal{D}$ aber theoretisch semi-entscheidbar.

[1] Boom: http://boom.cs.berkeley.edu/

A Beweise

A.1 Erfüllbarkeitsäquivalenz des Herbrand-Modells

Satz (Löwenheim, Skolem)**.** Eine geschlossene Formel $\varphi \in \mathcal{FO}$ hat genau dann ein Modell, wenn φ ein Herbrand-Modell hat. Damit hat jede erfüllbare geschlossene Formel insbesondere ein abzählbares Modell.

Beweis (sinngemäß nach [Sch87])**.** Nur „$\Rightarrow$“ ist zu zeigen.

Wenn φ ein Modell hat, dann hat auch die Skolemform ψ von φ ein Modell nach Satz 2.11. Nenne dieses $\mathcal{A}$ und dessen Grundmenge A. Konstruiere nun eine Herbrand-Struktur $\mathcal{H}$, wähle jedoch ein Element aus A, falls eine neue Konstante gewählt werden muss. Gesucht ist nun eine Interpretation der Relationen aus $\mathcal{A}$ in $\mathcal{H}$, so dass $\mathcal{H} \models \psi$.

Sei R eine n-stellige Relation aus $\mathcal{A}$ mit Interpretation $R^{\mathcal{A}}$ und seien die t_i Terme aus U_H. Dann wähle $R^{\mathcal{H}}$ als die Menge mit

$$R^{\mathcal{H}} = \left\{(t_1, t_2, \ldots, t_n) \in (U_H)^n \mid (t_1^{\mathcal{A}}, t_2^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}}) \in R^{\mathcal{A}}\right\}$$

Die Interpretation $R^{\mathcal{H}}$ beschreibt also einfach, dass alle n-Tupel von Termen aus U_H (diese bestehen aus beliebig tief verschachtelten Funktionen und Konstanten, sind aber immer variablenfrei) die Relation R erfüllen, wenn die Formel $R(t_1, t_2, \ldots, t_n)$ auch im Modell $\mathcal{A}$ interpretiert erfüllt ist. Dies ist für ein gegebenes $R(t_1, t_2, \ldots, t_n)$ eindeutig.

Nun ist noch zu zeigen, dass $\mathcal{H} \models \psi$.

Dafür führen wir eine Induktion über die Anzahl n der in ψ enthaltenen Quantoren durch.

Ist ψ quantorenfrei, also $n = 0$, enthält es keine Variablen und ist damit eine rein aussagenlogische Verknüpfung von Relationsformeln, deren Argumentterme nur aus Konstanten und Funktionstermen bestehen. Man kann ψ in eine logisch äquivalente konjunktive Normalform bringen. Dann gilt

$$\begin{aligned} \psi &\equiv \bigwedge_i \bigvee_j R_k(t_1, t_2, \ldots, t_l) \\ \Leftrightarrow \mathcal{A} &\models \bigwedge_i \bigvee_j R_k(t_1, t_2, \ldots, t_l) \\ \Leftrightarrow &\quad \bigwedge_i \bigvee_j R_k^{\mathcal{A}}(t_1^{\mathcal{A}}, t_2^{\mathcal{A}}, \ldots, t_l^{\mathcal{A}}) \end{aligned}$$

Nach Definition von $R^{\mathcal{H}}$, da die Terme variablenfrei und damit $\in U_H$ sind:

$$\begin{aligned} &\Leftrightarrow \quad \bigwedge_i \bigvee_j R_k^{\mathcal{H}}(t_1^{\mathcal{H}}, t_2^{\mathcal{H}}, \ldots, t_l^{\mathcal{H}}) \\ &\Leftrightarrow \mathcal{H} \models \bigwedge_i \bigvee_j R_k(t_1, t_2, \ldots, t_l) \end{aligned}$$

Nun habe ψ insgesamt $n > 0$ Quantoren. Da ψ Skolemform ist, gibt es einen quantorenfreien Suffix F von ψ und es gilt

$$\mathcal{A} \models \forall x_1 \ldots \forall x_n F$$

Nach der Erfüllungsdefinition für Allquantoren:

$$\Leftrightarrow \mathcal{A}[x_n/a] \models \forall x_1 \ldots \forall x_{n-1} F \quad \text{für alle } a \in A$$

x_n ist also freie Variable in F und wird durch $[x_n/a]$ von $\mathcal{A}$ mit a belegt. Man kann die Aussage auch abschwächen und statt $a \in A$ Terme $t^{\mathcal{A}}$ einsetzen, da $t^{\mathcal{A}} \in A$ für alle $t \in U_H$ (wir haben die eventuell nötige künstliche Konstante für U_H aus A gewählt).

$$\Rightarrow \mathcal{A}[x_n/t^{\mathcal{A}}] \models \forall x_1 \ldots \forall x_{n-1} F \quad \text{für alle } t \in U_H$$

Die Belegung von x_n mit $t^{\mathcal{A}}$ hat jedoch den gleichen Effekt wie die Ersetzung von x_n durch t direkt in F, solange im Modell $\mathcal{A}$ ausgewertet wird.

$$\Leftrightarrow \mathcal{A} \models \forall x_1 \ldots \forall x_{n-1} F[x_n/t] \quad \text{für alle } t \in U_H$$

Da nur noch $n-1$ Quantoren vorhanden sind und die Formel für jedes t geschlossene Skolemform ist, kann vorausgesetzt werden:

$$\begin{aligned} &\Leftrightarrow \mathcal{H} \models \forall x_1 \ldots \forall x_{n-1} F[x_n/t] \quad \text{für alle } t \in U_H \\ &\Leftrightarrow \mathcal{H}[x_n/t^{\mathcal{H}}] \models \forall x_1 \ldots \forall x_{n-1} F \quad \text{für alle } t \in U_H \\ &\Leftrightarrow \mathcal{H} \models \forall x_1 \ldots \forall x_n F \end{aligned}$$

□

B Algorithmen

B.1 Unifikation zweier Literale

Algorithmus 3 : Unifikation zweier Literale

```
Prozedur unify(L, L')
    Eingabe : Prädikatenlogische Literale L, L' mit disjunkten Variablen
1   Es gilt L = P(t_1, ..., t_n) und L' = P'(t'_1, ..., t'_m).
2   if P ≠ P' oder n ≠ m then
3       error('nicht unifizierbar')
4   θ := [ ].
5   for i ← 1 to n do
6       θ := unifyTerm(t_i θ, t'_i θ) ∘ θ
7   return θ

Prozedur unifyTerm(t, t')
    Eingabe : Prädikatenlogische Terme t, t' mit disjunkten Variablen
1   if t oder t' ist eine Variable x then
2       Es sei t = x und t' der andere Term.
        // occurs check
3       if x kommt in t' vor then
4           error('nicht unifizierbar')
5       return [x/t']
6   Es ist t = f(u_1, ..., u_n) und t' = f'(u'_1, ..., u'_m) mit n, m ≥ 0.
7   Konstanten sind 0-stellige Funktionen.
8   if f ≠ f' oder n ≠ m then
9       error('nicht unifizierbar')
10  θ := [ ]
11  for i ← 1 to n do
12      θ := unifyTerm(u_i θ, u'_i θ) ∘ θ
13  return θ
```

B.2 Allgemeine SLD-Resolution

Algorithmus 4 : Allgemeine SLD-Resolution

```
Daten : Bereinigte prädikatenlogische Klauselmenge Γ = {B_1, B_2, ..., B_n}
Prozedur SLD(Q)
    Eingabe : Anfrage Q
1   Z ← {(¬Q, [ ])}
2   while Z ≠ ∅ do
3       Wähle ein (K, θ) ∈ Z.
4       Z ← Z \ {(K, θ)}
5       if K = □ then
6           print „Γ ⊨ Qθ“
7       else
8           Es ist K = {¬L_1, ¬L_2, ..., ¬L_m}.
9           for i ← 1 to n do
10              Es ist B_i = {L'_1, ¬L'_2, ..., ¬L'_k}.
11              if {L_1, L'_1} unifizierbar then
                    // allgemeinster Unifikator
12                  θ* ← unify(L_1, L'_1)
                    // K* ist prädikatenlogischer Resolvent
13                  K* ← ((K \ {¬L_1}) ∪ (B_i \ {L'_1})) θ*
                    // K* ist also abgeleitete Klausel
14                  Z ← Z ∪ (K*, θ ∘ θ*)
```

B.3 Rekursive SLD-Resolution mit Tiefensuche

Algorithmus 5 : Rekursive SLD-Resolution mit Tiefensuche

Daten : Bereinigte prädikatenlogische Klauselmenge $\Gamma = \{B_1, \ldots, B_n\}$

Prozedur sldRec(K, θ, Q)

```
    Eingabe : Zielklausel K, Substitution θ, Anfrage Q
1   if K = □ then
2       print Γ ⊨ Qθ
3       return

4   Es ist K = {¬L_1, ¬L_2, ..., ¬L_m}.
5   for i ← 1 to n do
6       Es ist B_i = {L'_1, ¬L'_2, ..., ¬L'_k}.
7       if {L_1, L'_1} unifizierbar then
            // allgemeinster Unifikator
8           θ* ← unify(L_1, L'_1)
            // K* ist prädikatenlogischer Resolvent
9           K* ← ((K \ {¬L_1}) ∪ (B_i \ {L'_1})) θ*
            // versuche, K* abzuleiten
10          sldRec(K*, θ ∘ θ*, Q)
```

Prozedur sld(Q)

```
    Eingabe : Anfrage Q
1   sldRec(¬Q, [ ], Q)
```

B.4 Überprüfung eines Abhängigkeitsausdrucks δ

Algorithmus 6 : Überprüfung eines Abhängigkeitsausdrucks δ

```
Prozedur checkDependency(X, δ)
    Eingabe : Team X, Atom δ = dep(x_1, ..., x_n, w)
 1  forall the s ∈ X do
 2      forall the s' ∈ X do
 3          match ← true;
 4          for i ← 1 to n do
 5              if s(x_i) ≠ s'(x_i) then
 6                  match ← false;
 7                  break;
 8          if match then
 9              if s(w) ≠ s'(w) then
10                  return false ;
11  return true
```

B.5 SLD mit Tiefensuche und subtree-cutting

Algorithmus 7 : SLD mit Tiefensuche und subtree-cutting

Daten : Bereinigte prädikatenlogische Klauselmenge mit Abhängigkeiten $\Gamma = \{(B_1, \Delta_1), \ldots, (B_n, \Delta_n)\}$

Prozedur sldDepRec(K, θ, Δ)

Eingabe : Zielklausel K, Substitution θ, Abhängigkeitsmenge Δ

1. **if** $K = \square$ **then**
   - `// eine gefundene Lösung`
2.   **return** $\{s_\theta\}$
3. Es ist $K = \{\neg L_1, \neg L_2, \ldots, \neg L_m\}$.
4. $X \leftarrow \emptyset$
5. **for** $i \leftarrow 1$ **to** n **do**
6.   Es ist $B_i = \{L'_1, \neg L'_2, \ldots, \neg L'_k\}$.
7.   **if** $\{L_1, L'_1\}$ unifizierbar **then**
   - `// allgemeinster Unifikator`
8. $\theta^* \leftarrow$ unify(L_1, L'_1)
   - `// K* ist prädikatenlogischer Resolvent`
9. $K^* \leftarrow ((K \setminus \{\neg L_1\}) \cup (B_i \setminus \{L'_1\}))\,\theta^*$
   - `// leite rekursiv K* mit den Abhängigkeiten der angewendeten Regel ab, füge Lösungen dem Team hinzu`
10. $X \leftarrow X \cup$ sldDepRec$(K^*, \theta \circ \theta^*, \Delta_i)$
   - `// Überprüfe, ob Abhängigkeiten erfüllt`
11. **forall the** $\delta \in \Delta$ **do**
12.   **if** not(checkDependency(X, δ)) **then**
13. $X \leftarrow$ shrink(X, δ)
14. **return** X

Prozedur sldDep(Q, Δ)

Eingabe : Anfrage (Q, Δ)

1. **print** sldDepRec$(\neg Q, [\,], \Delta)$

B.6 Lexikalischer Scanner (Tokenizer)

Algorithmus 8 : Lexikalischer Scanner (Tokenizer)

```
Prozedur scan(w)
    Eingabe : Zeichenkette w ∈ Σ_R, w = a_1 a_2 ... a_n
    // aktueller Zustand
1   z ← z_0.
    // letzter Endzustand
2   z_e ← z_error
    // Index des zu lesenden Zeichens
3   i ← 1
    // Token-Start
4   s ← 1
    // Token-Ende
5   t ← 0
6   while i ≤ n do
7       z ← δ(z, a_i)
8       if z ∈ E then
            // Token a_s ... a_t erkannt
9           t ← i
10          i ← i + 1
11      else if z = z_error then
            // kein längeres Token möglich
12          if t < s then
13              error('ungültige Zeichen: ' a_s ... a_i)
14          print Token: a_s ... a_t Typ: z_e
15          s ← t + 1
16          z = z_0
17      else
18          i ← i + 1
```

B.7 Syntaktische Analyse (Parser)

Algorithmus 9 : Syntaktische Analyse (Parser)

```
Prozedur parseTree(w, P)
    Eingabe : Zeichenkette w ∈ (Ω ∪ {$})*, w = ω1ω2 ... ωn mit ωn = $,
              Parsing-Tabelle P
    // Wurzelknoten des Baums
1   R ← new Node('S')
    // Stack des Kellerparsers
2   S ← new Stack()
3   S.push(R)
    // Index des zu lesenden Zeichens in der Eingabe
4   i ← 1
5   while i ≤ n do
        // N ist Node, X ist Terminal oder Nichtterminal
6       N ← S.pop()
7       X ← N.content()
8       if X ∈ Ω then
            // X ist Terminalzeichen
9           if X = ωi then
10              i ← i + 1
11          else
12              error('unerwartetes Token: ' ωi)
13      else
            // X ist Nichtterminal
14          if P[X, ωi] ist leer then
15              error('unerwartetes Token: ' ωi)
16          else
17              (X → β) ← P[X, ωi]
18              for i ← |β| to 1 do
19                  B ← new Node(βi)
20                  N.getChildren().addFirst(B)
21                  S.push(B)
22  return R
```

Index

Abhängigkeit, 33, 34
Abhängigkeitsatom, *siehe* Dependence-Atom
Abhängigkeitslogik, *siehe* Dependence Logic
Anfrage, 8, 10
Atom, 9
Aussagenlogik, 15
Axiom, 11
Axiomenmenge, 11

Backtracking, 28
Belegung, 27, 35

Closed World Assumption, 9

Datalog, 40
Dependence Logic, 33
Dependence-Atom, 33
Domäne, 35
downward closure, 37

Erfüllbarkeit, 15

Fakt, 8
Formel
    bereinigte, 17
    geschlossene, 17
Funktor, 9

Grundmenge, 16

Hauptfunktor, 9
Herbrand-Expansion, 21
Hornklausel, 11

Instanz, 25
    Grund-, 25
Interpretation, 15

Klausel, 10
    definite, 24
    prädikatenlogische, 24
    Programm-, 24
    Ziel-, 24
Klauselmenge
    prädikatenlogische, 24
Konfiguration, 27
Konstante, 11
Kopf
    Regel-, 10

Lösung, 27
Lexem, 53
Lexer, 53
Literal, 10

Matrix, 17
Modell, 15
    Herbrand-, 17

occurs check, 25

Parser, 53
Prädikat, 10
Prädikatenlogik
    erster Stufe, 11
Präfix, 17
Pränexform, 17
Prolog, 7
    Pure Prolog, 16

Regel, 7

Relation, 11
Resolution
    Grund-, 22
    SLD-, 26
Resolvent
    prädikatenlogischer, 25
Rumpf
    Regel-, 10

Scanner, *siehe* Lexer
Signatur, 16
Skolemform, 18
Skolemisierung, 18
Struktur, 9, 15
    Herbrand-, 17
subtree cutting, 47

Team, 35
    duplizierendes, 36
    maximales, 44
    supplementierendes, 36
Teamresolution
    binäre, 46
Term, 11
    Grund-, 22
Token, 53
Tokenizer, *siehe* Lexer

Unifikation, 24
Unifikator, 25
    allgemeinster, 25
Universum, 16
    Herbrand-, 19

Variable, 9, 11

Literaturverzeichnis

[ASU86] Aho, Alfred V. & Sethi, Ravi & Ullman, Jeffrey D. *Compilers: Principles, Techniques and Tools*, 1986. 2nd edition (2003). Pearson International Edition, Prentice Hall, Inc.

[CGT89] Ceri, Stefano & Gottlob, Georg & Tanca, Letizia. *What You Always Wanted to Know About Datalog (And Never Dared to Ask)*, 1989.

[CW95] Chen, Weidong & Warren, David S. *Tabled Evaluation with Delaying for General Logic Programs*, 1995.

[EKMV12] Ebbing, Johannes & Kontinen, Juha & Müller, Julian-Steffen & Vollmer, Heribert. *A Fragment of Dependence Logic Capturing Polynomial Time*, 2012.

[Hof11] Hoffman, Dirk. *Grenzen der Mathematik: Eine Reise durch die Kerngebiete der mathematischen Logik*, 2011. 2. Auflage (2013). Springer Spektrum.

[ISO96] ISO/IEC CD 13211-1, Programming language Prolog, 1996.

[MP92] Marcinkowski, Jerzy & Pacholski, Leszek. *Undecidability of the Horn-clause implication problem*, 1992.

[Par11] Parchmann, Rainer. *Programmiersprachen und Übersetzer*, 2011. Institut für praktische Informatik, Leibniz Universität Hannover.

[Sch87] Schöning, Uwe. *Logik für Informatiker*, 1987. 5. Auflage (2000). Spektrum Akademischer Verlag.

[ShSt94] Shapiro, Ehud & Sterling, Leon. *The Art of Prolog, Second Edition: Advanced Programming Techniques*, 1994. Massachusetts Institute of Technology Press.

[Vää07] Väänänen, Jouko. *Dependence Logic: A New Approach to Independence Friendly Logic*, 2007. London Mathematical Society student texts, no. 70, Cambridge University Press.

[Vol12] Vollmer, Heribert. *Logik (und formale Systeme)*, 2012. Institut für theoretische Informatik, Leibniz Universität Hannover.

[Vol87] Vollmer, Heribert. *Resolutionsverfeinerungen und ihre Vollständigkeitssätze*, 1987. Studienarbeit, EWH Koblenz.

[Yan13] Yang, Fan. *Expressing Second-order Sentences in Intuitionistic Dependence Logic*, 2013.