Programmieren III: Logische Programmierung

Elmar Eder

19. November 1999

1 Einführung

Verschiedene Arten der Programmierung:

- Prozedurale (imperative) Programmierung
- Funktionale Programmierung
- Objektorientierte Programmierung
- Logische Programmierung

Logische Programmierung:

- *Logisches Programm* (logische Beschreibung eines Teils der Welt)
- *Anfragen* an das System

Das System verwendet einen automatischen Theorembeweiser. Das logische Programm enthält typischerweise eine logische Datenbank.

1.1 Einfache Prolog-Programme und Anfragen

Beispiel 1

```
vater(franz,max).
vater(franz,christine).
mutter(anna,max).
mutter(anna,christine).
mutter(christine,hans).
```

Dies ist ein *Prolog-Programm.* Jede Zeile dieses Programms ist ein *Faktum* (englisch: *fact*). Im Beispiel könnte das Faktum `vater(franz,max).` stehen für die Aussage „Franz ist der Vater von Max“. Dabei nennt man das Wort `vater` ein *Prolog-Prädikat* und die Wörter `franz` und `max` *Konstanten.* Prädikate und Konstanten sind *Prolog-Atome.* In Prolog müssen Prolog-Atome immer klein geschrieben werden. Mit einem Prolog-Programm stellt man dem Prolog-System Wissen in Form von logischen Aussagen zur Verfügung. Man kann jetzt eine *Anfrage* (englisch: *query* oder *goal clause*) stellen:

```
?- vater(franz,christine).
```

Darauf antwortet Prolog mit

```
Yes
```

Dies bedeutet, daß Prolog beweisen konnte, daß die in der Anfrage formulierte logische Aussage (wir nennen sie das *Ziel* der Anfrage) logisch aus dem Wissen, das im Prolog-Programm ausgedrückt ist, folgt. Wir sagen auch, diese Aussage folge logisch oder semantisch aus dem Programm. Man kann nun weitere Anfragen stellen:

```
?- vater(max,franz).
No
?- bruder(max,christine).
[WARNING: Undefined predicate: 'bruder/2']
No
```

Prolog gibt für jede dieser Anfragen die Antwort `Yes` oder `No` je nachdem, ob das Ziel der Anfrage logisch aus dem Prologprogramm folgt oder nicht. Da im Programm nichts über das Prädikat `bruder` ausgesagt ist, folgt `bruder(max,christine)` nicht rein logisch aus dem Programm und Prolog gibt die Antwort `No` aus[1]. Man kann aber Prolog auch danach fragen etwa, wer der Vater von Christine ist.

```
?- vater(X,christine).
```

Hier ist `X` eine Variable und Prolog versucht einen Wert für die Variable einzusetzen so, daß es beweisen kann, daß das Ziel der Anfrage für diesen Wert der Variablen gilt. Im Beispiel antwortet Prolog mit

```
X = franz
```

Variablen müssen in Prolog mit einem großen Buchstaben anfangen[2]. Zur Anfrage

```
?- mutter(anna,Y).
```

gibt es zwei Lösungen. Gibt man nach der ersten Antwort von Prolog `Y = max` einen Strichpunkt ein, so liefert Prolog in einer eigenen Zeile die zweite Lösung nach.

```
Y = max ;
Y = christine
```

So kann man Prolog solange auffordern weitere Antworten zu liefern, bis es mit `No` signalisiert, daß es keine weiteren Lösungen mehr gibt. In einer Anfrage dürfen auch zwei Variablen vorkommen.

```
?- mutter(X,Y).
X = anna
Y = max ;
X = anna
```

[1] Da das Prädikat `bruder` nicht im Programm definiert ist (also nichts darüber ausgesagt ist, wann es auf zwei Objekte zutrifft), gibt Prolog eine entsprechende Warnung (aber keine Fehlermeldung!) aus. Der Grund ist der, daß, obwohl eine solche Anfrage legitim ist, eine solche Situation meist dann eintritt, wenn entweder der Benutzer vergessen hat das Prädikat im Prologprogramm zu definieren oder er sich beim Eintippen des Prädikatnamens vertippt hat. In beiden Fällen bedeutet dies, daß der Benutzer tatsächlich einen Fehler gemacht hat.

[2] Der Unterstreichungsstrich (underscore) _ gilt für Prolog als Großbuchstabe.

```
Y = christine ;
X = christine
Y = hans ;
No
```

Hier hat Prolog alle drei Lösungen für X und Y gefunden.

Beispiel 2
Jetzt fügen wir dem Programm von Beispiel 1 noch die folgenden drei Regeln hinzu.

```
elternteil(X,Y) :- vater(X,Y).
elternteil(X,Y) :- mutter(X,Y).
grossvater(X,Z) :- vater(X,Y), elternteil(Y,Z).
```

Das Symbol `:-` ist zu lesen als „wenn" und das Komma in der dritten Regel zwischen `vater(X,Y)` und `elternteil(Y,Z)` ist zu lesen als „und". Die erste Regel könnte also etwa stehen für die Aussage „X ist Elternteil von Y, wenn X Vater von Y ist.", und die dritte Regel für die Aussage „X ist Großvater von Z, wenn X Vater von Y ist und Y Elternteil von Z ist.". Die drei Regeln *definieren* die Prädikate `elternteil` und `grossvater`. Ein Programm besteht (im wesentlichen) aus Fakten und Regeln. Die Fakten und Regeln eines Programms nennt man auch die *Klauseln* des Programms. Man kann jetzt etwa die Anfrage stellen

```
?- grossvater(X,Z).
```

Der Teil einer Regel, der vor dem Symbol `:-` steht, wird der *Kopf* der Regel genannt, der Teil rechts vom Symbol `:-` (mit Ausnahme des Punktes am Ende) wird der *Rumpf* der Regel genannt. Der Kopf einer Prolog-Regel besteht immer aus einem einzigen Prolog-Literal, während der Rumpf typischerweise aus ein oder mehr durch Beistriche voneinander getrennten Prolog-Literalen besteht. Eine Regel bedeutet: Wenn alle Prolog-Literale des Rumpfes wahr sind, dann ist auch der Kopf wahr. Ein Faktum kann man auffassen als eine Regel, deren Rumpf leer ist.

Selbstverständlich dürfen Variablen auch in einem Faktum auftreten. Ein Beispiel ist Gleichheitsprädikat, das man in Prolog durch ein Faktum `gleich(X,X).` definieren kann, das aber in Prolog auch schon vordefiniert unter dem Namen `=` zur Verfügung steht. Es ist sogar möglich mit dem Faktum `ist_irgendwas(X).` ein einstelliges Prädikat `ist_irgendwas` zu definieren, das auf jedes Objekt zutrifft[3]. Meist wird das Programm lesbarer, wenn man den Variablen aussagekräftige Namen gibt, z.B.

[3]Wenn eine Variable in einer Programmklausel nur einmal vorkommt wie in diesem Fall, dann gibt Prolog eine Warnung (keine Fehlermeldung!) über „Singleton variables" aus. Der Grund liegt in der in Prolog üblichen Praxis Variablen, die in einer Klausel nur einmal vorkommen, keinen Namen zu geben. Anstattdessen wird üblicherweise das Symbol `_` ('anonyme Variable') benutzt. Jedes Vorkommen von `_` (auch in ein und derselben Klausel) steht für eine neue Variable. So ist etwa `p(_,_)` eine andere Schreibweise für `p(X,Y)`, wobei die Variablen X und Y sonst nicht in der betreffenden Klausel vorkommen. Hält man sich an diese Vereinbarung, kann Prolog einen falsch geschriebenen Variablennamen meist daran erkennen, daß er in einer Klausel nur einmal vorkommt. Übrigens kann man einer Variablen, die in einer Klausel nur einmal vorkommt, auch einen Namen geben, der mit `_` beginnt. Dann gibt Prolog keine Warnung aus. Z.B. kann man in das Programm ein Faktum `faengt_mit_f_an(f(_Arg)).` schreiben.

```
...
grossvater(Grossvater,Person) :-
     vater(Grossvater,Elternteil),
     elternteil(Elternteil,Person).
?- grossvater(Grossvater,Enkel).
```

Stellen wir im Beispiel 1 eine weitere Anfrage

```
?- mutter(X,Y), mutter(X,Z).
X = anna
Y = max
Z = max ;
X = anna
Y = max
Z = christine ;
X = anna
Y = christine
Z = max ;
X = anna
Y = christine
Z = christine ;
X = christine
Y = hans
Z = hans ;
No
```

Während die bisherigen Anfragen jeweils nur aus einem Ziel bestanden, muß Prolog hier zwei Ziele (`mutter(X,Y)` und `mutter(X,Z)`) lösen, d.h. für geeignete Werte der Variablen beweisen. Beide Ziele müssen gelten. Der Beistrich ist also auch hier als ein 'und' zu lesen.

1.2 Debugging eines einfachen Prologprogramms mittels `trace`

Mit der Anfrage

```
?- trace.
```

kann man Prolog dazu veranlassen einen Trace durchzuführen. Dies bedeutet, daß Prolog während der Bearbeitung einer Anfrage dem Benutzer Informationen über den Fortgang dieser Bearbeitung gibt. Als einfaches Beispiel betrachten wir das folgende Prolog-Programm.

```
p(a).
p(b).
```

Wenn das Tracing eingeschaltet ist, dann gibt Prolog auf die Anfrage

```
?- p(X).
```

etwa den folgenden Trace aus (was genau ausgegeben wird, ist abhängig von dem jeweiligen Prolog-System).

```
Call: p(X) ?
Exit: p(a) ?
```

Die erste Zeile bedeutet, daß Prolog versucht das Ziel `p(X)` für einen geeigneten Wert der Variablen `X` zu beweisen. Wir sagen, Prolog rufe das Ziel `p(X)` auf (english: *call*). Die zweite Zeile bedeutet, daß Prolog das Ziel erfolgreich bewiesen hat für einen geeigneten Wert der Variablen, im Beispiel hat Prolog für die Variable `X` den Wert `a` eingesetzt. Prolog ist also aus der Anfrage mit dem Ziel `p(X)` erfolgreich ausgestiegen (englisch: *exit*) und hat `p(a)` bewiesen. Im Beispiel hat Prolog dazu die erste Klausel '`p(a).`' des Programms verwendet. Schließlich liefert Prolog die Antwort

```
X = a
```

genauso, wie wenn das Tracing nicht eingeschaltet wäre. Man kann nun durch Eingabe eines Strichpunkts Prolog nach weiteren Lösungen suchen lassen. Prolog gibt dann den folgenden Trace aus.

```
Redo: p(X) ?
Exit: p(b) ?
```

Die erste Zeile bedeutet, daß Prolog noch einmal versucht das Ziel `p(X)` für einen geeigneten Wert der Variablen `X` zu beweisen (englisch: *redo*). Prolog hat sich gemerkt, daß es schon versucht hat das Ziel `p(X)` mittels der ersten Klausel des Programms zu beweisen. Um nun eine neue Lösung zu finden, geht es daher zur nächsten passenden Programmklausel, das ist im Beispiel die zweite. Hiermit kann Prolog `p(b)` beweisen und steigt damit auf die Anfrage nach dem Ziel `p(X)` erfolgreich aus (exit) mit der nun bewiesenen Aussage `p(b)`. Schließlich liefert Prolog die Antwort

```
X = b
```

Betrachten wir nun dasselbe Programm und stellen wir die Anfrage

```
?- p(X), X=b.
```

Prolog gibt in etwa den folgenden Trace aus.

```
Call: p(X) ?
Exit: p(a) ?
Call: a=b ?
Fail: a=b ?
Redo: p(X) ?
Exit: p(b) ?
Call: b=b ?
Exit: b=b ?
```

Hier ruft Prolog das Ziel `p(X)` auf ('`Call: p(X)`') und steigt aus diesem Aufruf erfolgreich aus mit `p(a)`, wobei es für die Variable `X` den Wert `a` eingesetzt hat ('`Exit: p(a)`'). Um die Anfrage erfolgreich zu beantworten, muß Prolog noch das zweite Ziel `X=b` der Anfrage beweisen, wobei für die Variable `X` der Wert `a` eingesetzt werden muß. Prolog ruft also das Ziel `a=b` auf ('`Call: a=b`'). Prolog kann dieses Ziel nicht beweisen. Wir sagen, das Ziel schlage fehl (englisch: *fail*). Im Trace wird dies dem Benutzer durch die Zeile '`Fail: a=b`' mitgeteilt. Da sich also das Ziel `X=b` mit dem Wert `a` für die Variable `X` nicht beweisen läßt, versucht Prolog das Ziel `p(X)` noch einmal mit einem möglicherweise anderen Wert für die Variable `X` zu beweisen (‘`Redo: p(X)`’). Dies gelingt mittels der zweiten Programmklausel mit dem Wert `b` für die Variable `X` (‘`Exit: p(b)`’). Schließlich wird mit diesem Wert `b` für die Variable `X` versucht das Ziel `X=b` zu beweisen (‘`Call: b=b`’). Dies gelingt, da Prolog das Reflexivitätsgesetz der Gleichheit bekannt ist (‘`Exit: b=b`’).

Bei Programmen, die neben Fakten auch Regeln enthalten, kann jedes Call oder Redo selbst wieder einen komplexen Trace auslösen. Wir werden später darauf zurückkommen.

1.3 Funktionszeichen

In Prolog kann man auch Strukturen (oder Records) verwenden. So kann etwa in einer logische Datenbank von Studenten jeder Student durch seinen Vornamen, seinen Familiennamen und seine Matrikelnummer identifiziert sein in der Form `student(Vorname,Familienname,Matrikelnummer)`. Die Datenbank könnte dann etwa Angaben zu den Noten als Fakten enthalten:

```
note(student(max,schneider,53872),2).
note(student(karl,meyer,27613),4).
note(student(hans,huber,48761),2).
```

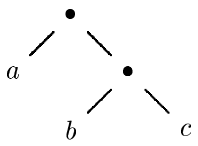
Das könnte im Beispiel etwa bedeuten:

Der Student Max Schneider mit der Matrikelnummer 53872 hat die Note 2.
Der Student Karl Meyer mit der Matrikelnummer 27613 hat die Note 4.
Der Student Hans Huber mit der Matrikelnummer 48761 hat die Note 2.

Das Prolog-Atom `student` ist hier ein *Funktionszeichen*. Der Begriff stammt aus der Logik und kommt daher, daß dieses Prolog-Atom sich auffassen läßt als ein Symbol für eine dreistellige mathematische Funktion, die z.B. dem Vornamen ‘Max’, dem Familiennamen ‘Schneider’ und der Matrikelnummer ‘53872’ einen konkreten Studenten zuordnet. Mit Funktionszeichen kann man komplexe *Terme* bilden. Als *Terme* bezeichnet man Konstanten und Variablen, aber auch das, was man durch Anwendung von Funktionszeichen auf Terme bekommt. So ist z.B. auch `student(max,schneider,53872)` ein Term. Das Prolog-Atom `note` ist hier ein Prädikatszeichen oder Prolog-Prädikat. Man kann jetzt Anfragen stellen und bekommt in den Antworten komplexe Terme geliefert.

```
?- note(Student,Note).
Student = student(max,schneider,53872)
Note = 2 ;
Student = student(karl,meyer,27613)
Note = 4 ;
Student = student(hans,huber,48761)
Note = 2 ;
No
?- note(Student,2).
Student = student(max,schneider,53872) ;
Student = student(hans,huber,48761) ;
No
```

Mit Funktionszeichen kann man auch Bäume darstellen, z.B. den Binärbaum



mittels eines zweistelligen Funktionszeichens `baum` als Term

```
baum(a,baum(b,c))
```

Dabei sind `a` und `b` Konstanten. Man könnte den Baum etwa in eine Sammlung von Bäumen aufnehmen, die in einer logischen Datenbank das einstellige Prädikatszeichen (Prolog-Prädikat) `vorhanden` definiert. Hierzu muß man das folgende Faktum in das Prologprogramm schreiben.

```
vorhanden(baum(a,baum(b,c))).
```

Auf eine Anfrage

```
?- vorhanden(baum(a,baum(B,c))).
```

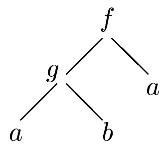
erhielte man dann etwa die Antwort

```
B = b
```

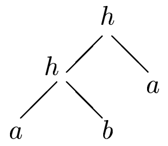
In der obigen Anfrage sind `a` und `c` Konstanten und damit auch Terme. Weiter ist `B` eine Variable und damit ebenfalls ein Term. Auch `baum(B,c)` und `baum(a,baum(B,c))` sind Terme, während `vorhanden(baum(a,baum(B,c)))` ein Prolog-Literal ist.

Im nächsten Beispiel werden wir geordnete knotenmarkierte Binärbäume betrachten. Zur Erinnerung: Ein Baum ist ein Tripel (V, E, w) bestehend aus einer Menge V von Knoten (englisch: vertices), einer Menge E von Kanten (englisch: edges) und einer Wurzel w mit $E \subseteq V \times V$ und $w \in E$ so, daß es von w aus zu jedem $x \in V$ genau einen Weg gibt. Bei einem binären Baum wird zusätzlich verlangt, daß es zu jedem $x \in V$ höstens zwei Nachfolger gibt, also höchstens zwei $y \in V$ mit $(x, y) \in E$. Ein geordneter knotenmarkierter Baum ist ein Tripel $((V, E, w), \phi, \mu)$, wobei ϕ eine Abbildung ist, die jedem $x \in V$ eine Ordnungsrelation $\leq_x$ auf der Menge $\{y \in V \mid (x, y) \in E\}$ der Nachfolger von x zuordnet und $\mu\colon V \to M$ eine Abbildung von der Menge V der Knoten in eine Menge M, genannt die Menge der Markierungen, ist.

Beispiel 3
Wir betrachten diejenigen geordneten knotenmarkierten Binärbäume, für die jeder Knoten entweder genau 2 Nachfolger (einen linken und einen rechten Nachfolger) hat (innerer Knoten) oder keinen Nachfolger hat (Blatt), und wobei die inneren Knoten mit f oder g markiert sind und die Blätter mit a oder b. Ein Beispiel für einen solchen Baum ist



Aufgabe: Transformiere einen solchen Baum in einen neuen Baum durch Ersetzung der Markierungen aller innerer Knoten durch h. Im Beispiel schaut der transformierte Baum so aus:



Darstellung in Prolog: Ein Baum wird als Term dargestellt, im Beispiel:

```
f(g(a,b),a)
```

Dies soll transformiert werden in

```
h(h(a,b),a)
```

Wir führen hierzu ein Prolog-Prädikat `transf` ein, wobei `transf(Baum1,Baum2)` bedeutet „Die Transformierte von `Baum1` ist `Baum2`.“. Dann kann man etwa die Anfrage stellen

```
?- transf(f(g(a,b),a),Baum).
```

und erhält als Antwort

```
Baum = h(h(a,b),a)
```

Das Programm, das dieses Prädikat definiert, lautet

```
transf(a,a).
transf(b,b).
transf(f(Baum1,Baum2),h(TBaum1,TBaum2)) :-
     transf(Baum1,TBaum1),
     transf(Baum2,TBaum2).
transf(g(Baum1,Baum2),h(TBaum1,TBaum2)) :-
     transf(Baum1,TBaum1),
     transf(Baum2,TBaum2).
```

1.4 Operatoren in Prolog

Dieser Abschnitt der Vorlesung kann vorläufig übersprungen werden, da es sich dabei nicht um einen notwendigen Teil der Programmiersprache Prolog, sondern lediglich um die Möglichkeit der Einführung einer bequemeren Schreibweise handelt.

Ein algebraischer Ausdruck wie $3 \cdot 4 + 7$ wird in Prolog behandelt als Term `+(*(3,4),7)`. Dabei sind `3`, `4` und `7` Konstanten und `+` und `*` zweistellige Funktionszeichen. Für Prolog ist die Infixschreibweise `3*4+7` nichts anderes als eine Abkürzung für `+(*(3,4),7)`. Ein arithmetischer Term darf in Infixschreibweise eingegeben werden und Prolog gibt arithmetische Terme auch in Infixschreibweise aus. Mit

```
?- display(3*4+7).
```

kann man sich den Term in der Form

```
+(*(3,4),7)
```

anzeigen lassen. In der Sprechweise von Prolog nennt man ein zweistelliges Funktionszeichen wie `+` oder `*`, für das die Infixschreibweise vereinbart worden ist, einen zweistelligen *Operator*. Die Funktionszeichen `+` und `*` sind in Prolog bereits standardmäßig als zweistellige Operatoren deklariert. Man kann aber auch selbst ein Prologatom als Infixoperator deklarieren. Wären etwa `+` und `*` nicht bereits standardmäßig als Operatoren deklariert, so könnte man sie selber deklarieren, indem man die folgenden zwei Zeilen an den Anfang des Prologprogramms schreibt.

```
?- op(500,yfx,+).
?- op(400,yfx,*).
```

oder

```
:- op(500,yfx,+).
:- op(400,yfx,*).
```

Allgemein wird mit `op(Prec,Spec,Name)` ein Operator mit dem Namen `Name` spezifiziert. Die ersten beiden Argumente `Prec` und `Spec` dienen lediglich dazu das Weglassen von Klammern etwa in Ausdrücken wie `(3*4)+6` zu gestatten.

Das erste Argument `Prec` ist die Präzedenz des zu definierenden Operators. Je niedriger die Präzedenz eines Operators ist, desto stärker bindet er. Im Beispiel hat der Operator `+` die Präzedenz 500 und der Operator `*` die Präzedenz 400. Der Operator `*` bindet also stärker als der Operator `+` („Punkt vor Strich"). So interpretiert Prolog etwa den Ausdruck `3*4+7` wie `(3*4)+7` und nicht wie `3*(4+7)`.

Das zweite Argument ist ein Spezifikator, der die Art der Verwendung des Operators symbolisiert. Das `f` symbolisiert den Operator und `x` und `y` symbolisieren die Argumente des Operators. Ein `x` bedeutet, daß, sofern das betreffende Argument selbst mit einem Operator gebildet und nicht geklammert ist, dieser Operator eine kleinere Präzedenz haben muß als der Operator `Name`. Bei einem `y` muß der betreffende Operator ein Präzedenz kleiner oder gleich der Präzedenz von `Name` haben.

So interpretiert Prolog zum Beispiel den Ausdruck `3*4+7` wie `(3*4)+7`, da die Präzedenz 400 von `*` kleiner oder gleich der Präzedenz 500 von `+` ist. Ebenso interpretiert Prolog den Ausdruck `3+4+5` wie `(3+4)+5` und nicht wie `3+(4+5)`, da der Ausdruck `3+4` als linkes Argument von `+` zulässig ist (die Präzedenz von `+` ist kleinergleich der Präzedenz von `+`), aber der Ausdruck `4+5` nicht ohne Klammerung als rechtes Argument von `+` zulässig ist (die Präzedenz von `+` ist nicht kleiner als die Präzedenz von +). Entsprechend interpretiert Prolog den Ausdruck `3+4+5*6` wie `(3+4)+(5*6)`. Generell wird mit dem Spezifikator `yfx` ein linksassoziativer Operator deklariert und mit dem Spezifikator `xfy` ein rechtsassoziativer Operator. Mit dem Spezifikator `xfx` wird bei Ausdrücken, die durch mehrfache Anwendung des betreffenden Operators gebildet sind, vollständige Klammerung vorgeschrieben.

Man kann auch einstellige Präfix- oder Postfixoperatoren deklarieren. Zum Beispiel könnte man in einem Programm, mit dem man logische Formeln manipulieren kann, einen einstelligen Präfixoperator `nicht` deklarieren durch

```
?- op(800,fy,nicht).
```

Dann betrachtet Prolog den Ausdruck '`nicht a`' als Abkürzung für '`nicht(a)`' und den Ausdruck '`nicht nicht a`' als Abkürzung für '`nicht(nicht(a))`'. Bei der Deklaration

```
?- op(800,fx,nicht).
```

hingegen ist ein Ausdruck '`nicht nicht a`' nicht erlaubt. Mit

```
?- current_op(Prec,Spec,Op).
```

kann man Prolog nach den deklarierten Operatoren fragen.

2 Aussagenlogik

2.1 Die Sprache der Aussagenlogik

Grundzeichen:

1. Aussagenvariablen (P, Q, R, ... auch mit Indizes, ...)

2. Junktoren:
   $\neg$ 'nicht'
   $\wedge$ 'und'
   $\vee$ 'oder'

3. Runde Klammern ( )

Die Menge der Aussagenvariablen bezeichnen wir mit $\mathbb{A}$.

Definition 1 (induktive Definition der *Formeln*) Formeln sind spezielle Zeichenreihen aus Grundzeichen, und zwar

1. Jede Aussagenvariable ist eine Formel.[4]

2. Wenn F eine Formel ist, dann ist auch $\neg F$ eine Formel.

3. Wenn F und G Formeln sind, dann sind auch $(F \wedge G)$ und $(F \vee G)$ Formeln.

[4] Wir identifizieren hier eine Zeichenreihe, die aus nur einem Zeichen besteht, mit diesem einen Zeichen.

Wenn F und G Formeln sind, dann nennen wir die Formel $\neg F$ die *Negation* von F, die Formel $(F \wedge G)$ die *Konjunktion* von F und G und die Formel $(F \vee G)$ die *Disjunktion* der Formeln F und G.

Wenn wir über Formeln reden, lassen wir meist Klammern weg, wo dies nicht zu Mißverständnissen führt. Für Formeln verwenden wir Zeichen F, G, H, ... , auch mit Indizes, ...

Definition 2 (induktive Definition der *Teilformeln* einer Formel)

1. F ist eine Teilformel von F.
2. Wenn $\neg G$ eine Teilformel von F ist, dann ist auch G eine Teilformel von F.
3. Wenn $(G \wedge H)$ eine Teilformel von F ist, dann sind auch G und H Teilformeln von F.
4. Wenn $(G \vee H)$ eine Teilformel von F ist, dann sind auch G und H Teilformeln von F.

2.2 Semantik

Die Semantik ist die Lehre von der Bedeutung. In der mathematischen Logik interessiert man sich insbesondere dafür, wann eine Formel wahr ist und wann sie falsch ist. Hierzu betrachten wir die Menge der Wahrheitswerte (auch Boole'sche Werte genannt)

$$\mathbb{B} = \{\text{wahr}, \text{falsch}\}.$$

Definition 3 Eine *Variablenbelegung* (oder *Interpretation*) ist eine Funktion $i\colon \mathbb{A} \to \mathbb{B}$.

Definition 4 des *Wahrheitswertes* iF einer Formel F bei einer Variablenbelegung i.

1. $iP := i(P)$ für jede Aussagenvariable P.
2. $i\neg F := \begin{cases} \text{wahr}, & \text{wenn } iF = \text{falsch} \\ \text{falsch} & \text{sonst} \end{cases}$
3. $i(F \wedge G) := \begin{cases} \text{wahr}, & \text{wenn } iF = \text{wahr und } iG = \text{wahr} \\ \text{falsch} & \text{sonst} \end{cases}$
4. $i(F \vee G) := \begin{cases} \text{wahr}, & \text{wenn } iF = \text{wahr oder } iG = \text{wahr} \\ \text{falsch} & \text{sonst} \end{cases}$

Das 'oder' ist ein nicht ausschließendes 'oder', d.h., wenn iF und iG beide $=$ wahr sind, dann ist auch $i(F \vee G) =$ wahr. Für jede Variablenbelegung i und jede Formel F gilt $iF \in \mathbb{B}$.

Definition 5 Ein *Modell* einer Formel F ist eine Variablenbelegung i so, daß $iF =$ wahr ist.

Definition 6 Eine Formel F heißt *allgemeingültig*, wenn für jede Variablenbelegung i gilt $iF =$ wahr. Eine Formel F heißt *erfüllbar*, wenn es eine Variablenbelegung i gibt mit $iF =$ wahr (also genau dann, wenn sie ein Modell hat). Andernfalls heißt sie *unerfüllbar*.

Definition 7 Wir sagen, eine Formel G *folge semantisch* aus einer Formel F (in Zeichen: $F \models G$), wenn jedes Modell von F auch Modell von G ist. Zwei Formeln F und G heißen *semantisch äquivalent* (in Zeichen: $F \sim G$), wenn $iF = iG$ ist für alle Variablenbelegungen i (also genau dann, wenn sie die gleichen Modelle haben).

Die Relation $\sim$ ist eine Äquivalenzrelation auf der Menge der Formeln.

Definition 8 Sei S eine Menge von Formeln. Dann heißt i ein *Modell* von S, wenn i Modell jeder Formel $F \in S$ ist. Wenn S eine Formelmenge und G eine Formel ist, dann sagen wir, G *folge semantisch* aus S (in Zeichen: $S \models G$), wenn jedes Modell von S ein Modell von G ist. Wir sagen, eine Formelmenge S sei *erfüllbar*, wenn sie ein Modell hat. Andernfalls heißt sie *unerfüllbar*.

Beispiel 4
Sei $\mathbb{A} := \{P, Q\}$ und $F :\equiv P \wedge \neg Q$. Diese Formel F hat genau ein Modell i, das gegeben ist durch

$$\begin{array}{rcl} i(P) & = & \text{wahr} \\ i(Q) & = & \text{falsch.} \end{array}$$

Denn wenn eine Variablenbelegung i ein Modell von F sein soll, dann muß $iF =$ wahr sein. Also $i(P \wedge \neg Q) =$ wahr. Also gilt nach Definition 4 $iP =$ wahr und $i\neg Q =$ wahr. Also gilt weiter nach Definition 4 $iQ =$ falsch und damit $i(P) =$ wahr und $i(Q) =$ falsch.

Umgekehrt sei $i(P) =$ wahr und $i(Q) =$ falsch. Dann ist nach Definition 4 $iP =$ wahr und $iQ =$ falsch und damit $i\neg Q =$ wahr und folglich $i(P \wedge \neg Q) =$ wahr.

Beispiel 5
Sei $\mathbb{A} := \{P, Q\}$ und $F :\equiv P \vee \neg Q$. Dann hat F genau 3 Modelle i_1, i_2 und i_3, die gegeben sind durch

$$\begin{array}{rcl} i_1(P) & = & \text{wahr} \\ i_1(Q) & = & \text{wahr} \end{array}$$

$$\begin{array}{rcl} i_2(P) & = & \text{wahr} \\ i_2(Q) & = & \text{falsch} \end{array}$$

$$\begin{array}{rcl} i_3(P) & = & \text{falsch} \\ i_3(Q) & = & \text{falsch} \end{array}$$

In jedem der beiden Beispiele ist die Formel F erfüllbar, aber nicht allgemeingültig.

Beispiel 6
Sei $\mathbb{A} := \{P\}$ und $F :\equiv P \wedge \neg P$. Dann hat F kein Modell und ist daher unerfüllbar.

Beispiel 7
Sei $\mathbb{A} := \{P\}$ und $F :\equiv P \vee \neg P$. Dann ist jede Variablenbelegung über der Menge $\mathbb{A}$ der Aussagenvariablen ein Modell von F. Damit ist F allgemeingültig.

Satz 1 *Für alle Formeln F, G und H gelten die folgenden semantischen Äquivalenzen.*

$$
\begin{array}{rll}
F \wedge G \sim G \wedge F & & \textit{Kommutativität von } \wedge \\
F \vee G \sim G \vee F & & \textit{Kommutativität von } \vee \\
\neg\neg F \sim F & & \\
(F \wedge G) \wedge H \sim F \wedge (G \wedge H) & & \textit{Assoziativität von } \wedge \\
(F \vee G) \vee H \sim F \vee (G \vee H) & & \textit{Assoziativität von } \vee \\
F \wedge (G \vee H) \sim (F \wedge G) \vee (F \wedge H) & & \textit{Distributivgesetz} \\
F \vee (G \wedge H) \sim (F \vee G) \wedge (F \vee H) & & \textit{Distributivgesetz} \\
F \wedge F \sim F & & \\
F \vee F \sim F & & \\
F \wedge G \sim \neg(\neg F \vee \neg G) & & \textit{DeMorgan'sche Regel} \\
F \vee G \sim \neg(\neg F \wedge \neg G) & & \textit{DeMorgan'sche Regel}
\end{array}
$$

Satz 2 *Sei F eine Formel und sei G eine Teilformel von F. Weiter sei G' eine Formel mit $G \sim G'$ und F' sei aus F entstanden durch Ersetzung eines Vorkommens der Teilformel G durch die Formel G'. Dann ist $F \sim F'$.*

Beispiel 8
Seien G und H zwei Formeln und sei $F :\equiv G \vee H$. Da $G \sim \neg\neg G$ gilt, läßt sich der Satz anwenden, wenn man $G' :\equiv \neg\neg G$ und $F' :\equiv \neg\neg G \vee H$ setzt. Es ist also $G \vee H \sim \neg\neg G \vee H$.

Satz 3 *Sei F eine Formel. Dann gilt:*

$$
\begin{array}{rcl}
F \textit{ ist allgemeingültig} & \iff & \neg F \textit{ ist unerfüllbar} \\
F \textit{ ist unerfüllbar} & \iff & \neg F \textit{ ist allgemeingültig}
\end{array}
$$

Satz 4 *Sei S eine Formelmenge und F eine Formel. Dann gilt:*

$$S \models F \iff S \cup \{\neg F\} \textit{ ist unerfüllbar.}$$

Dieser Satz, der auch in der später in der Vorlesung einzuführenden Erweiterung der Aussagenlogik, nämlich in der Prädikatenlogik, gilt, ist ein für die logische Programmierung grundlegender Satz. Denn eine Klausel eines Prologprogramms kann man als Formel auffassen, das Prologprogramm selbst als Menge S von Formeln. Die Anfrage kann man als Formel F auffassen. Die Aufgabe von Prolog ist es dann zu beweisen, daß $S \models F$ gilt. Prolog tut dies, indem es aus $S \cup \{\neg F\}$ einen Widerspruch ableitet, also indem es zeigt, daß diese Menge unerfüllbar ist. Nach dem Satz muß daher $S \models F$ gelten.

2.3 Normalformen

Definition 9 Eine Formel heißt ein *Literal*, wenn sie eine Aussagenvariable P oder die Negation $\neg P$ einer Aussagenvariablen P ist.

Definition 10 der Formeln in *Negations-Normalform* (induktive Definition)

1. Jedes Literal ist in Negations-Normalform.
2. Wenn F und G Formeln in Negations-Normalform sind, dann sind auch die Formeln $F \wedge G$ und $F \vee G$ in Negations-Normalform.

Eine derartige induktive Definition ist so zu verstehen, daß eine Formel genau dann in Negations-Normalform ist, wenn sich durch endlich häufige Anwendung der beiden in der Definition formulierten Regeln ableiten läßt, daß sie in Negations-Normalform ist. Man kann das auch mengentheoretisch folgendermaßen formulieren.

Definition 11 der Formeln in *Negations-Normalform* (alternative Definition). Die Menge der Formeln in Negations-Normalform ist die kleinste Menge S mit

1. Jedes Literal ist Element von S.
2. $F, G \in S \Longrightarrow F \wedge G, F \vee G \in S$.

Für 'Negations-Normalform' schreiben wir auch kurz 'NNF'.

Satz 5 *Zu jeder Formel F gibt es eine Formel F' in NNF mit $F \sim F'$.*

Beweis: durch Induktion nach der Länge von F.
Induktionsvoraussetzung: Zu jeder Formel G, die kürzer als F ist, existiere eine Formel G' in NNF mit $G \sim G'$.
Induktionsbehauptung: Es gibt eine Formel F' in NNF mit $F \sim F'$.
Beweis (Induktionsschritt): Wir machen eine Fallunterscheidung.

1. F ist eine Aussagenvariable P. Dann sei $F' :\equiv P$.
2. F ist die Negation einer Formel. In diesem Fall müssen wir wieder vier Unterfälle unterscheiden.
   (a) $F \equiv \neg P$. Dann sei $F' :\equiv \neg P$.
   (b) $F \equiv \neg\neg G$. Dann ist G kürzer als F. Daher existiert nach Induktionsvoraussetzung ein G' in NNF mit $G \sim G'$. Nun ist $F \sim G$, also $F \sim G'$. Sei nun $F' :\equiv G'$.
   (c) $F \equiv \neg(G \wedge H)$. Dann sind $\neg G$ und $\neg H$ kürzer als F. Nach Induktionsvoraussetzung gibt es also Formeln I und J in NNF mit $\neg G \sim I$ und $\neg H \sim J$. Daher ist
   $$F \sim \neg G \vee \neg H \sim I \vee \neg H \sim I \vee J.$$
   Sei nun $F' :\equiv I \vee J$. Dann ist $F \sim F'$, und F' ist in NNF.
   (d) $F \equiv \neg(G \vee H)$. Für diesen Fall geht der Beweis analog.

3. $F \equiv G \wedge H$. Dann sind G und H kürzer als F. Nach Induktionsvoraussetzung existieren Formeln G' und H' in NNF mit $G \sim G'$ und $H \sim H'$. Also

$$F \equiv G \wedge H \sim G' \wedge H \sim G' \wedge H'.$$

   Hiermit ist $F \sim G' \wedge H'$. Sei nun $F' :\equiv G' \wedge H'$. Dann ist F' in NNF, da G' und H' in NNF sind. Außerdem ist $F \sim F'$.

4. $F \equiv G \vee H$. In diesem Fall erfolgt der Beweis analog.

q.e.d.

Definition 12 der *Konjunktion* $F_1 \wedge \cdots \wedge F_n$ von Formeln $F_1, \ldots, F_n$.

1. Die Konjunktion von F_1 ist F_1.
2. $F_1 \wedge \cdots \wedge F_{n+1} :\equiv (F_1 \wedge \cdots \wedge F_n) \wedge F_{n+1}$.

Definition 13 der *Disjunktion* $F_1 \vee \cdots \vee F_n$ von Formeln $F_1, \ldots, F_n$.

1. Die Disjunktion von F_1 ist F_1.
2. $F_1 \vee \cdots \vee F_{n+1} :\equiv (F_1 \vee \cdots \vee F_n) \vee F_{n+1}$.

Definition 14 Eine Formel F ist in *konjunktiver Normalform* (*KNF*), wenn sie eine Konjunktion von Disjunktionen von Literalen ist.

Definition 15 Eine Formel F ist in *disjunktiver Normalform* (*DNF*), wenn sie eine Disjunktion von Konjunktionen von Literalen ist.

Beispiel 9

$(P \vee \neg Q) \wedge (\neg P \vee Q) \wedge \neg R$	ist in KNF.
$(P \wedge \neg Q \wedge \neg P) \vee Q$	ist in DNF.
$(P \vee Q) \wedge (\neg P \vee Q)$	ist in KNF.
$P \wedge Q \wedge \neg R$	ist in KNF und in DNF.
$P \vee Q$	ist in KNF und in DNF.
$\neg P$	ist in KNF und in DNF.

Jede Formel in KNF oder DNF ist in NNF.

Lemma 1 *Zu jeder Formel F in NNF gibt es eine Formel F' in KNF mit $F \sim F'$.*

Beweis: durch Induktion nach der Länge von F.

INDUKTIONSVORAUSSETZUNG: Die Behauptung gelte für alle Formeln in NNF, die kürzer als F sind.

INDUKTIONSBEHAUPTUNG: Die Behauptung gilt für F.

BEWEIS (INDUKTIONSSCHRITT): Wir machen eine Fallunterscheidung.

1. F ist ein Literal. Dann sei $F' :\equiv F$.

2. $F \equiv G \wedge H$. Dann sind G und H kürzer als F und ebenfalls in NNF. Nach Induktionsvoraussetzung gibt es ein $G' \equiv G_1 \wedge \cdots \wedge G_m$ in KNF, wobei jedes G_i eine Disjunktion von Literalen ist, mit $G \sim G'$. Analog gibt es ein $H' \equiv H_1 \wedge \cdots \wedge H_n$ in KNF, wobei jedes H_j eine Disjunktion von Literalen ist, mit $H \sim H'$. Nun gilt

$$F \equiv G \wedge H \sim G' \wedge H' \sim G_1 \wedge \cdots \wedge G_m \wedge H_1 \wedge \cdots \wedge H_n.$$

   Sei nun $F' :\equiv G_1 \wedge \cdots \wedge G_m \wedge H_1 \wedge \cdots \wedge H_n$. Dann ist F' in KNF und $F \sim F'$.

3. $F \equiv G \vee H$. Wie oben bekommen wir wieder

$$\begin{array}{ll} G' \equiv G_1 \wedge \cdots \wedge G_m & \quad G \sim G' \\ H' \equiv H_1 \wedge \cdots \wedge H_n & \quad H \sim H' \end{array}$$

   Hieraus ergibt sich

$$\begin{array}{rcl} F & \equiv & G \vee H \\ & \sim & G' \vee H' \\ & \equiv & (G_1 \wedge \cdots \wedge G_m) \vee (H_1 \wedge \cdots \wedge H_n) \\ & \sim & (G_1 \vee H_1) \wedge (G_1 \vee H_2) \wedge \cdots \wedge (G_1 \vee H_n) \wedge \\ & & (G_2 \vee H_1) \wedge (G_2 \vee H_2) \wedge \cdots \wedge (G_2 \vee H_n) \wedge \\ & & \vdots \\ & & (G_m \vee H_1) \wedge (G_m \vee H_2) \wedge \cdots \wedge (G_m \vee H_n) \end{array}$$

   Letztere semantische Äquivalenz ergibt sich durch die wiederholte Anwendung eines der beiden Distributivgesetze sowie des Assoziativgesetzes für $\wedge$. Jedes G_i und jedes H_j ist eine Disjunktion von Literalen. Durch wiederholte Anwendung des Assoziativgesetzes für $\vee$ kann man also $G_i \vee H_j$ in eine Disjunktion I_{ij} von Literalen umwandeln, d.h. $G_i \vee H_j \sim I_{ij}$. Es folgt, daß F semantisch äquivalent ist zu der folgenden Formel F'.

$$\begin{array}{c} I_{11} \wedge I_{12} \wedge \cdots \wedge I_{1n} \wedge \\ I_{21} \wedge I_{22} \wedge \cdots \wedge I_{2n} \wedge \\ \vdots \\ I_{m1} \wedge I_{m2} \wedge \cdots \wedge I_{mn} \end{array}$$

   Außerdem ist die so definierte Formel F' ist in KNF.

q.e.d.

Satz 6 *Zu jeder Formel F gibt es eine Formel F' in KNF mit $F \sim F'$.*

Beweis: Nach Satz 5 gibt es zu F eine semantisch äquivalente Formel G in NNF. Nach dem Lemma gibt es zu G eine semantisch äquivalente Formel F' in KNF. Also $F \sim G$ und $G \sim F'$ und somit $F \sim F'$. q.e.d.

Satz 7 *Zu jeder Formel F gibt es eine Formel F' in DNF mit $F \sim F'$.*

Beweis: analog. q.e.d.

Man kann aber diesen Satz auch anders beweisen. Die Idee ist die folgende. Man bestimmt zunächst alle Modelle der Formel F, z.B. durch eine Wahrheitswerttabelle. Als Beispiel hierzu betrachten wir die folgende Formel F.

$$\neg(\neg P \lor Q) \lor \neg P.$$

Dazu stellt man eine Tabelle der Wahrheitswerte aller ihrer Teilformeln für alle Variablenbelegungen auf. Jede Zeile der folgenden Tabelle entspricht einer Variablenbelegung.

P	Q	$\neg P$	$\neg P \lor Q$	$\neg(\neg P \lor Q)$	$\neg(\neg P \lor Q) \lor \neg P$
wahr	wahr	falsch	wahr	falsch	falsch
wahr	falsch	falsch	falsch	wahr	wahr
falsch	wahr	wahr	wahr	falsch	wahr
falsch	falsch	wahr	wahr	falsch	wahr

Nun ist eine Variablenbelegung i genau dann ein Modell der Formel F, wenn in dieser Wahrheitswerttabelle am Schnittpunkt der zu i gehörigen Zeile mit der letzten Spalte der Eintrag 'wahr' steht. Da im Beispiel in der letzten Spalte genau dreimal der Eintrag 'wahr' steht, hat F genau drei Modelle. Das erste dieser Modelle ist dadurch definiert, daß bei ihm P wahr ist und Q falsch ist, oder anders ausgedrückt, daß bei ihm $P \land \neg Q$ wahr ist. Das zweite Modell ist dadurch gekennzeichnet, daß bei ihm $\neg P \land Q$ wahr ist und das dritte dadurch, daß bei ihm $\neg P \land \neg Q$ wahr ist. Insgesamt ist also eine Variablenbelegung i genau dann ein Modell von F, wenn bei i die folgende Formel F'

$$(P \land \neg Q) \lor (\neg P \land Q) \lor (\neg P \land \neg Q)$$

wahr wird. Also ist $F \sim F'$ und offensichtlich ist F' in DNF.

Formeln in KNF spielen in der logischen Programmierung eine zentrale Rolle. Betrachten wir etwa das Prologprogramm

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

und dazu die Anfrage

```
?- p.
```

Jede der Klauseln des Programms kann man als Formel schreiben. Die erste Programmklausel entspricht einer logischen Implikation $\mathtt{q} \land \mathtt{r} \to \mathtt{p}$. Diese Implikation kann man als Abkürzung auffassen für die Formel $\neg(\mathtt{q} \land \mathtt{r}) \lor \mathtt{p}$, die nach einer der DeMorgan'schen Regeln semantisch äquivalent ist zu $\neg\mathtt{q} \lor \neg\mathtt{r} \lor \mathtt{p}$, also auch zu $\mathtt{p} \lor \neg\mathtt{q} \lor \neg\mathtt{r}$. Insgesamt entsprechen die drei Programmklauseln den folgenden drei Formeln.

$$\mathtt{p} \lor \neg\mathtt{q} \lor \neg\mathtt{r}$$
$$\mathtt{q}$$
$$\mathtt{r}$$

Das Programm entspricht also der Formelmenge S, die aus diesen drei Formeln besteht. Die Anfrage entspricht einer Formel F. Im Beispiel ist $F \equiv \mathtt{p}$. Prolog muß versuchen nachzuweisen, daß $S \models F$ gilt. Nach Satz 4 gilt dies genau dann, wenn $S \cup \{\neg F\}$ unerfüllbar ist. Genau die Unerfüllbarkeit dieser Formelmenge bestehend aus den Formeln

$$\begin{gathered} \mathtt{p} \lor \neg \mathtt{q} \lor \neg \mathtt{r} \\ \mathtt{q} \\ \mathtt{r} \\ \neg \mathtt{p} \end{gathered}$$

wird von Prolog versucht nachzuweisen. Jede dieser Formeln ist eine Disjunktion von Literalen. Die Formelmenge ist genau dann unerfüllbar, wenn die Konjunktion

$$(\mathtt{p} \lor \neg \mathtt{q} \lor \neg \mathtt{r}) \land \mathtt{q} \land \mathtt{r} \land \neg \mathtt{p}$$

dieser Formeln unerfüllbar ist. Diese Formel ist aber in konjunktiver Normalform. Prolog ist ein automatisches Beweissystem, das gewisse Formeln in KNF auf Unerfüllbarkeit testet.

2.4 Klauseln in der Logik

In der Logik wird eine Disjunktion von Literalen dargestellt als Menge dieser Literale. So wird die Formel $P \lor \neg Q \lor \neg R$ dargestellt als die Menge $\{P, \neg Q, \neg R\}$. So eine Menge wird in der Logik *Klausel* genannt.

Definition 16 Eine *Klausel* in der Logik ist eine endliche Menge von Literalen.

Definition 17 Eine Klausel ist *wahr* bei einer Variablenbelegung i genau dann, wenn mindestens eines ihrer Literale wahr ist bei i. Andernfalls ist sie *falsch* bei i.

Eine Disjunktion D von Literalen ist bei einer Variablenbelegung i genau dann wahr, wenn die zu D gehörige Klausel bei der Variablenbelegung i wahr ist.

Analog wie für Formeln oder Formelmengen sind für Klauseln die semantischen Begriffe *Modell*, *Erfüllbarkeit*, *Unerfüllbarkeit*, *semantische Folgerung*, etc. definiert.

Nach unserer Definition ist auch die leere Menge als Klausel erlaubt. Sie wird mit bezeichnet. Die leere Klausel ist bei jeder Variablenbelegung falsch.

2.5 Resolution

2.5.1 Resolution allgemein

Der Resolutionskalkül ist ein Kalkül zum automatischen Nachweis der Unerfüllbarkeit einer Menge S von Klauseln. Hierzu werden, ausgehend von S solange weitere Klauseln, die semantisch aus S folgen, mit der sogenannten *Resolutionsregel* abgeleitet, bis die leere Klausel und damit ein Widerspruch abgeleitet ist.

Definition 18 Zwei Literale heißen zueinander *komplementär*, wenn eines der beiden Literale die Negation des anderen ist. Das zu einem Literal L komplementäre Literal wird mit $\overline{L}$ bezeichnet. Es gilt also $\overline{L} \equiv \neg L$ oder $L \equiv \neg\overline{L}$.

Definition 19 (*Resolutionsregel*):
Seien c und d zwei Klauseln und sei L ein Literal mit $L \in c$ und $\overline{L} \in d$. Dann heißt die Klausel

$$(c \setminus \{L\}) \cup (d \setminus \{\overline{L}\})$$

eine *Resolvente* von c und d. Die Klauseln c und d heißen *Elternklauseln*. Die Literale L und $\overline{L}$ in den Klauseln c bzw. d heißen die *wegresolvierten* Literale. Eine Menge S' von Klauseln ist durch einen *Resolutionsschritt* aus einer Menge S von Klauseln entstanden, wenn $S' = S \cup \{e\}$ ist, wobei e eine Resolvente zweier Klauseln $c, d \in S$ ist. Eine *Resolutionsableitung* aus einer Klauselmenge S ist eine Folge $(c_1, c_2, c_3, \dots)$ von Klauseln so, daß jedes c_k ein Element von S ist oder eine Resolvente von c_i und c_j mit $i, j < k$. Eine *Resolutionsableitung* einer Klausel c aus S ist eine Resolutionsableitung $(c_1, \dots, c_n)$ aus S, die mit c endet (also mit $c_n = c$). Eine *Resolutionswiderlegung* einer Klauselmenge S ist eine Resolutionsableitung von aus S. Eine Klauselmenge S heißt mit Resolution *widerlegbar* oder *resolutionswiderlegbar*, wenn es eine Resolutionswiderlegung von S gibt. Die Resolutionsregel schreiben wir auch so:

$$\frac{c \qquad\qquad d}{(c \setminus \{L\}) \cup (d \setminus \{\overline{L}\})} \quad , \qquad \text{wenn } L \in c \text{ und } \overline{L} \in d.$$

Die Resolutionsregel erlaubt aus zwei Klauseln eine neue abzuleiten. Bei Regeln allgemein spricht man davon, daß man aus den *Prämissen* die *Konklusion* ableitet. Speziell bei der Resolutionsregel werden die Prämissen *Elternklauseln* genannt und die Konklusion wird *Resolvente* genannt.

Lemma 2 *Seien c und d zwei Klauseln und sei e eine Resolvente von c und d. Dann gilt $\{c, d\} \models e$.*

Beweis: Sei i ein Modell von $\{c, d\}$. Dann sind c und d bei i wahr. Es gibt also Literale $J \in c$ und $K \in d$ so, daß J und K wahr sind bei der Variablenbelegung i. Nach Voraussetzung ist e eine Resolvente von c und d. Es gibt also ein Literal L so, daß $L \in c$ und $\overline{L} \in d$ ist und daß $e = (c \setminus \{L\}) \cup (d \setminus \{\overline{L}\})$ gilt. Wir unterscheiden nun drei (sich nicht gegenseitig ausschließende) Fälle.

1. $J \not\equiv L$. Dann ist $J \in c \setminus \{L\}$, also $J \in e$. Damit ist e wahr bei der Variablenbelegung i, also i ein Modell von e.
2. $K \not\equiv \overline{L}$. Dann ist $K \in c \setminus \{\overline{L}\}$, also $K \in e$. Damit ist e wahr bei der Variablenbelegung i, also i ein Modell von e.
3. $J \equiv L$ und $K \equiv \overline{L}$. Dieser Fall kann nicht auftreten, da J und K beide wahr sind bei der Variablenbelegung i.

Also ist in jedem Fall i ein Modell von e. Damit gilt $\{c, d\} \models e$. q.e.d.

Satz 8 *(Korrektheitssatz): Sei S eine Klauselmenge, die mit Resolution widerlegbar ist. Dann ist S unerfüllbar.*

Beweis: Sei $(c_1, \ldots, c_n)$ mit $c_n =$ eine Resolutionswiderlegung von S. Angenommen, S wäre erfüllbar. Dann würde S ein Modell i haben. Aus dem Lemma folgt durch Induktion nach j, daß i ein Modell von jedem c_j ist $(j = 1, \ldots, n)$. Für $j := n$ würde dies bedeuten, daß i ein Modell der leeren Klausel ist. Da wir wissen, daß die leere Klausel kein Modell hat, folgt ein Widerspruch. q.e.d.

Satz 9 *(Vollständigkeitssatz): Sei S eine unerfüllbare Klauselmenge. Dann besitzt S eine Resolutionswiderlegung.*

Eine Resolutionsableitung aus einer Formelmenge S läßt sich auch als knotenmarkierter Binärbaum darstellen. Dabei sind die Blätter mit Elementen aus S markiert und die Markierung eines jeden inneren Knotens ist die Resolvente der Markierungen seiner beiden Nachfolgerknoten. Der Baum wird üblicherweise so gezeichnet, daß die Wurzel unten ist und die Blätter oben sind.

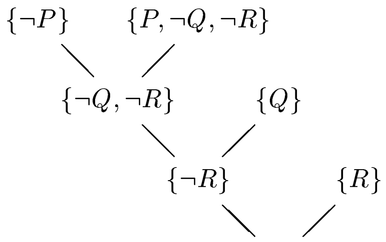
Beispiel 10
Sei

$$S = \{\{P, \neg Q, \neg R\}, \{Q\}, \{R\}, \{\neg P\}\}.$$

Dann ist

$$(\{\neg P\}, \{P, \neg Q\, \neg R\}, \{\neg Q, \neg R\}, \{Q\}, \{\neg R\}, \{R\},)$$

eine Resolutionswiderlegung von S. Diese Resolutionswiderlegung wird durch den folgenden Baum dargestellt.



Schreibt man die Klauseln in Prolog-Notation, so schaut die Klauslemenge folgendermaßen aus

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

und der Widerlegungsbaum folgendermaßen.

```
?- p.     p :- q, r.
    \     /
    ?- q, r.      q.
          \     /
          ?- r.       r.
              \     /
```

2.5.2 Beispiel für Prologs Suche nach einer Resolutionswiderlegung

Wenn man eine Anfrage an Prolog stellt, so wählt sich Prolog zunächst das erste Ziel der Anfrage aus und versucht eine Resolution mit einer Programmklausel, die dieses Ziel als Kopf hat. Prolog nimmt die erste Programmklausel, die paßt. Dadurch erzeugt Prolog eine neue Anfrage (die Resolvente), die es zu lösen versucht. Findet Prolog zu dieser neuen Anfrage keine Lösung (oder keine Lösung mehr, wenn es mit Strichpunkt aufgefordert wird mehrere Lösungen zu suchen), dann macht Prolog die Beweisschritte, die es seit der Resolution der ursprünglichen Anfrage mit der ersten passenden Programmklausel gemacht hat, wieder rückgängig (backtracking) und versucht eine Resolution mit der nächsten passenden Programmklausel. Wenn es keine passende Programmklausel mehr gibt, schlägt die Anfrage fehl und Prolog antwortet mit `No`.

Ein einfaches Beispiel soll dies demonstrieren. Gegeben sei das folgende Prologprogramm und die folgende Anfrage.

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

Prolog macht nun die folgenden Schritte

1. Die Anfrage (Zielklausel) '`?- p.`' wird resolviert mit der ersten Programmklausel '`p:- q, r.`' und als Resolvente ergibt sich die neue Zielklausel '`?- q, r.`'. Prolog merkt sich, mit welcher Programmklausel es resolviert hat.

2. Die Zielklausel '`?- q, r.`' wird resolviert mit der dritten Programmklausel '`q.`' und als Resolvente ergibt sich die neue Zielklausel '`?- r.`' Prolog merkt sich wieder, mit welcher Programmklausel es resolviert hat.

3. Die Zielklausel '`?- r.`' wird versucht mit einer der Programmklauseln zu resolvieren, was scheitert.

4. Prolog macht die Beweisschritte bis zu Schritt 2[5] rückgängig und sucht eine Alternative zu Schritt 2, also eine andere Möglichkeit die Zielklausel '`?- q, r.`' zu lösen (backtracking). Da Prolog in einer Zielklausel grundsätzlich das erste Literal zum Wegresolvieren auswählt, versucht es

[5] bis zum letzten Schritt, bei dem es möglicherweise eine alternative Wahlmöglichkeit gegeben hätte

eine alternative Klausel mit dem Kopf `q` zu finden. Im Programm paßt da aber nur die dritte Programmklausel, von der Prolog sich gemerkt hat, daß es sie schon im Schritt 2 ausgewählt hat. Daher scheitert der Versuch die Zielklausel '`?- q, r.`' zu lösen.

5. Prolog macht die Beweisschritte bis zu Schritt 1 rückgängig und versucht zu Schritt 1 eine Alternative, also eine andere Möglichkeit die Zielklausel '`?- p.`' zu lösen (backtracking). Prolog hat sich in Schritt 1 gemerkt, daß es schon versucht hat diese Zielklausel mit der ersten Programmklausel zu resolvieren. Daher resolviert es die Zielklausel '`?- p.`' diesmal mit der zweiten Programmklausel '`p :- s.`' und als Resolvente ergibt sich die neue Zielklausel '`?- s.`'. Prolog merkt sich wieder, mit welcher Programmklausel es resolviert hat.

6. Prolog versucht die Zielklausel '`?- s.`' mit einer der Programmklauseln zu resolvieren, was scheitert.

7. Prolog macht die Beweisschritte bis zu Schritt 5 rückgängig und sucht eine Alternative zu Schritt 5, also eine andere Möglichkeit die Zielklausel '`?- p.`' zu lösen (backtracking). Prolog hat sich in Schritt 5 gemerkt, daß es dort versucht hat mit der zweiten Programmklausel zu resolvieren und da es keine weitere Programmklausel mit dem Kopf `p` mehr gibt, scheitert der Versuch die Zielklausel '`?- p.`' zu lösen.

8. Damit ist die ursprüngliche Anfrage gescheitert und Prolog antwortet mit `No`.

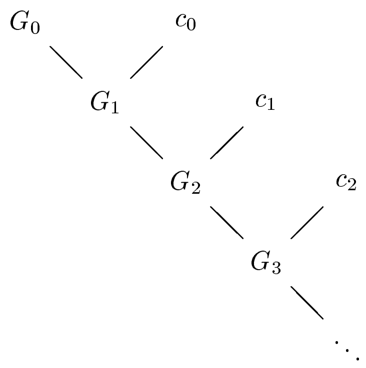
2.5.3 Hornklauseln und SLD-Resolution

Definition 20 Ein *positives Literal* ist eine Aussagenvariable (also ein Literal, das kein Negationszeichen $\neg$ enthält). Ein *negatives Literal* ist die Negation einer Aussagenvariablen (also ein Literal, das das Negationszeichen $\neg$ enthält).

Definition 21 Eine *Hornklausel* ist eine Klausel, die höchstens ein positives Literal enthält. Eine *Programmklausel* ist eine Klausel, die genau ein positives Literal enthält. Eine *Anfrageklausel* oder *Zielklausel* (englisch *query clause* oder *goal clause*) ist eine Klausel, die kein positives Literal enthält.

Jede Hornklausel ist entweder eine Programmklausel oder eine Zielklausel. Prolog arbeitet mit Hornklauseln. Wir werden für Hornklauseln oft Prolognotation verwenden.

Definition 22 *SLD-Resolution* ist Resolution für Hornklauseln mit folgenden Einschränkungen: $S = P \cup \{G\}$. P ist eine Menge von Programmklauseln. G ist eine Zielklausel. Eine SLD-Resolutionsableitung hat die Form



wobei $G_0 \equiv G$ ist und $c_0, c_1, c_2, \dots \in P$ sind. Dabei muß jeder Resolutionsschritt die Form haben.

$$\frac{\texttt{?-}\ A_1, \dots, A_j, \dots, A_m. \qquad\qquad A_j\ \texttt{:-}\ B_1, \dots, B_n.}{\texttt{?-}\ A_1, \dots, A_{j-1}, B_1, \dots, B_n, A_{j+1}, \dots, A_m.}$$

Das Literal A_j heißt dabei das *ausgewählte Literal*. Es wird dabei angenommen, daß eine *Auswahlfunktion* (englisch: *selection funktion*) gegeben sei, die zu einer Zielklausel das ausgewählte Literal findet.

Prolog wählt eimmer das jeweils erste Literal einer Zielklausel aus (es ist also $j = 1$). Den Resolutionsschritt kann man dann auch so schreiben.

$$\frac{\texttt{?-}\ A_1, \dots, A_m. \qquad\qquad A_1\ \texttt{:-}\ B_1, \dots, B_n.}{\texttt{?-}\ B_1, \dots, B_n, A_2, \dots, A_m.}$$

Andere Systeme der logischen Programmierung können sich auch ein anderes Literal als ausgewähltes Literal aussuchen.

Der Name 'SLD-Resolution' kommt aus dem Englischen:
Selection function
Linear resolution
Definite clauses.

Unter *linearer* Resolution versteht man Resolution, die auf solche Resolutionsableitungen beschränkt ist, für die der rechte Nachfolger eines jeden inneren Knotens des Ableitungsbaumes ein Blatt ist. Unter einer *definiten* Klausel versteht man eine Hornklausel.

3 Prädikatenlogik erster Stufe

3.1 Die Sprache der Prädikatenlogik erster Stufe

Im Gegensatz zur Aussagenlogik erlaubt die Prädikatenlogik auch über Objekte und darüber zu sprechen, ob eine Eigenschaft auf alle Objekte zutrifft und ob es ein Objekt gibt, das die Eigenschaft erfüllt. Ein Beispiel für eine prädikatenlogische Formel in der Mathematik ist die Formel

$$\forall x \exists y\ x < y.$$

oder, in Präfixschreibweise, $\forall x \exists y <(x,y)$. Wie in der Aussagenlogik ist eine derartige Formel aus Grundzeichen aufgebaut.

Grundzeichen:

1. Abzählbar unendlich viele *Variablen* (auch Individuenvariablen oder Objektvariablen genannt) (u, v, w, x, y, z, ...).

2. n-stellige *Funktionszeichen* für $n = 0, 1, 2, \ldots$ (f, g, h, ...). Nullstellige Funktionszeichen heißen *Konstanten* (a, b, c, d, ...).

3. n-stellige Prädikatszeichen für $n = 0, 1, 2, \ldots$ (P, Q, R, ...).

4. *Junktoren* $\neg$, $\wedge$, $\vee$.

5. *Quantoren* $\forall$ ('für alle', *Allquantor*), $\exists$ ('existiert', *Existenzquantor*).

6. Runde Klammern ( ) und Beistrich ,.

Definition 23 (induktive Definition der *Terme*) Terme sind spezielle Zeichenreihen aus Grundzeichen, und zwar

1. Jede Variable ist ein Term.

2. Wenn f ein n-stelliges Funktionszeichen ist und $t_1, \ldots, t_n$ Terme sind, dann ist $f(t_1, \ldots, t_n)$ ein Term.

Im Fall $n = 0$ ist mit $f(t_1, \ldots, t_n)$ einfach die Konstante f gemeint.

Für Terme verwenden wir Zeichen r, s, t, ... , auch mit Indizes, ...

Definition 24 (induktive Definition der *Formeln*) Formeln sind spezielle Zeichenreihen aus Grundzeichen, und zwar

1. Wenn P ein n-stelliges Prädikatszeichen ist und $t_1, \ldots, t_n$ Terme sind, dann ist $P(t_1, \ldots, t_n)$ ein Formel. Eine solche Formel nennen wir eine *Atomformel*.

2. Wenn F eine Formel ist, dann ist auch $\neg F$ eine Formel.

3. Wenn F und G Formeln sind, dann sind auch $(F \wedge G)$ und $(F \vee G)$ Formeln.

4. Wenn F eine Formel und x eine Variable ist, dann sind auch $\forall x F$ und $\exists x F$ Formeln.

Wenn F und G Formeln sind, dann nennen wir die Formel $\neg F$ die *Negation* von F, die Formel $(F \wedge G)$ die *Konjunktion* von F und G und die Formel $(F \vee G)$ die *Disjunktion* der Formeln F und G.

Wenn wir über Formeln reden, lassen wir meist Klammern weg, wo dies nicht zu Mißverständnissen führt. Für Formeln verwenden wir Zeichen F, G, H, ... , auch mit Indizes, ...

Definition 25 (induktive Definition der *Teilformeln* einer Formel)

1. F ist eine Teilformel von F.

2. Wenn $\neg G$ eine Teilformel von F ist, dann ist auch G eine Teilformel von F.
3. Wenn $(G \wedge H)$ eine Teilformel von F ist, dann sind auch G und H Teilformeln von F.
4. Wenn $(G \vee H)$ eine Teilformel von F ist, dann sind auch G und H Teilformeln von F.
5. Wenn $\forall x G$ eine Teilformel von F ist, dann ist auch G eine Teilformel von F.
6. Wenn $\exists x G$ eine Teilformel von F ist, dann ist auch G eine Teilformel von F.

Definition 26 Sei F eine Formel und sei $\forall x G$ oder $\exists x G$ eine Teilformel von F. Dann heißt jedes Vorkommen der Variablen x in F, das in dieser Teilformel liegt, ein *gebundenes* Vorkommen von x in F. Ein nicht gebundenes Vorkommen einer Variablen in einer Formel heißt ein *freies* Vorkommen dieser Variablen in der Formel. Eine Formel, in der keine Variablen frei vorkommen, heißt eine *geschlossene* Formel.

Beispiel 11
In der Formel

$$P(x) \vee \neg \forall y\ Q(x, f(y))$$

sind beide Vorkommen der Variablen x frei und beide Vorkommen der Variablen y gebunden. Da in dieser Formel x frei vorkommt, ist die Formel nicht geschlossen. In der Formel

$$P(x, y) \vee \neg \forall y\ Q(x, f(y))$$

sind beide Vorkommen der Variablen x sowie das erste Vorkommen der Variablen y frei, aber das zweite und das dritte Vorkommen der Variablen y gebunden.

Definition 27 Sei F eine Formel und seien $x_1, \ldots, x_n$ diejenigen Variablen, die in F frei vorkommen. Dann heißt die Formel $\forall x_1 \ldots \forall x_n F$ ein *Allabschluß* (auch *$\forall$-Abschluß*) von F und die Formel $\exists x_1 \ldots \exists x_n F$ heißt ein *Existenzabschluß* (auch *$\exists$-Abschluß*) von F. Einen Allabschluß von F bezeichnen wir auch mit $\forall[F]$, einen Existenzabschluß von F mit $\exists[F]$.

3.2 Semantik

Definition 28 Eine *Interpretation* ist ein Tripel $i = (D, i_f, i_p)$, wobei gilt

1. D ist eine nichtleere Menge, genannt der *Individuenbereich* (englisch: domain).
2. i_f ist eine Abbildung, die jedem n-stelligen Funktionszeichen f eine n-stellige Funktion $i_f(f) \colon D^n \to D$ zuordnet.
3. i_p ist eine Abbildung, die jedem n-stelligen Prädikatszeichen P eine n-stellige Relation (= Prädikat) $i_p(P) \subseteq D^n$ zuordnet.

Definition 29 Eine *Variablenbelegung* ist eine Abbildung $\mathcal{V}$, die jeder Variablen x einen Wert $\mathcal{V}(x) \in D$ in einem Individuenbereich D zuordnet.

Definition 30 des *Wertes* $i_{\mathcal{V}}t$ eines Terms t bei einer Interpretation i und einer Variablenbelegung $\mathcal{V}$. Dieser Wert ist ein Element des Individuenbereichs.

1. $i_{\mathcal{V}}x := \mathcal{V}(x)$.
2. $i_{\mathcal{V}}f(t_1, \ldots, t_n) := i_f(f)(i_{\mathcal{V}}t_1, \ldots, i_{\mathcal{V}}t_n)$.

Definition 31 Sei $\mathcal{V}$ eine Variablenbelegung über dem Individuenbereich D, sei x eine Variable und sei $\xi \in D$. Dann bezeichnen wir mit $\mathcal{V}^x_\xi$ die Variablenbelegung, die gegeben ist durch

$$\mathcal{V}^x_\xi(y) := \begin{cases} \xi, & \text{wenn } y \equiv x \\ \mathcal{V}(y) & \text{sonst} \end{cases}$$

Definition 32 des *Wahrheitswertes* $i_{\mathcal{V}}F$ einer Formel F bei einer Interpretation i und einer Variablenbelegung $\mathcal{V}$.

1. $i_{\mathcal{V}}P(t_1, \ldots, t_n) := \begin{cases} \text{wahr}, & \text{wenn } (i_{\mathcal{V}}t_1, \ldots, i_{\mathcal{V}}t_n) \in i_p(P) \\ \text{falsch} & \text{sonst} \end{cases}$
2. $i_{\mathcal{V}}\neg F := \begin{cases} \text{wahr}, & \text{wenn } i_{\mathcal{V}}F = \text{falsch} \\ \text{falsch} & \text{sonst} \end{cases}$
3. $i_{\mathcal{V}}(F \wedge G) := \begin{cases} \text{wahr}, & \text{wenn } i_{\mathcal{V}}F = \text{wahr und } i_{\mathcal{V}}G = \text{wahr} \\ \text{falsch} & \text{sonst} \end{cases}$
4. $i_{\mathcal{V}}(F \vee G) := \begin{cases} \text{wahr}, & \text{wenn } i_{\mathcal{V}}F = \text{wahr oder } i_{\mathcal{V}}G = \text{wahr} \\ \text{falsch} & \text{sonst} \end{cases}$
5. $i_{\mathcal{V}}\forall xF := \begin{cases} \text{wahr}, & \text{wenn } i_{\mathcal{V}^x_\xi}F = \text{wahr für alle } \xi \in D \\ \text{falsch} & \text{sonst} \end{cases}$
6. $i_{\mathcal{V}}\exists xF := \begin{cases} \text{wahr}, & \text{wenn es ein } \xi \in D \text{ gibt so, daß } i_{\mathcal{V}^x_\xi}F = \text{wahr ist} \\ \text{falsch} & \text{sonst} \end{cases}$

Der Wahrheitswert einer geschlossenen Formel F hängt nicht von der Variablenbelegung, sondern nur von der Interpretation i ab. Wir bezeichnen ihn mit iF.

Definition 33 Ein *Modell* einer geschlossenen Formel ist eine Interpretation i so, daß $iF = \text{wahr}$ ist.

Die Begriffe *semantische Folgerung*, *semantische Äquivalenz*, *Allgemeingültigkeit* sowie *Erfüllbarkeit* von Formeln und von Formelmengen sind entsprechend wie in der Aussagenlogik definiert.

Die Sätze, die wir in der Aussagenlogik kennengelernt haben, gelten auch in der Prädikatenlogik. Für die logische Programmierung sind insbesondere die folgenden beiden Sätze von Bedeutung.

Satz 10 *Sei F eine Formel. Dann gilt:*

$$\begin{array}{rcl} F \textit{ ist allgemeingültig} & \iff & \neg F \textit{ ist unerfüllbar} \\ F \textit{ ist unerfüllbar} & \iff & \neg F \textit{ ist allgemeingültig} \end{array}$$

Satz 11 *Sei S eine Formelmenge und F eine Formel. Dann gilt:*

$$S \models F \iff S \cup \{\neg F\} \textit{ ist unerfüllbar.}$$

3.3 Prologklauseln als Formeln

Wir definieren $F \to G$ als Abkürzung für $\neg F \vee G$.

Beispiel 12

```
grossvater(X,Z) :-
      vater(X,Y),
      elternteil(Y,Z).
```

entspricht der Formel

$$\forall x \forall y \forall z \big(V(x,y) \wedge E(y,z) \to G(x,z)\big).$$

„Für alle x, y und z gilt: wenn x Vater von y ist und y Elternteil von z ist, dann ist x Großvater von z."

Allgemein entspricht eine Prologklausel

$$A\texttt{:-}B_1, \ldots, B_n.$$

der Formel

$$\forall[B_1 \wedge \cdots \wedge B_n \to A].$$

Dabei ist $B_1 \wedge \cdots \wedge B_n \to A$ eine Abkürzung für $\neg(B_1 \wedge \cdots \wedge B_n) \vee A$, was wiederum semantisch äquivalent ist zur Formel $A \vee \neg B_1 \vee \cdots \vee \neg B_n$. Die Prologklausel entspricht also auch der Formel

$$\forall[A \vee \neg B_1 \vee \cdots \vee \neg B_n].$$

Ein Faktum

$$A.$$

entspricht der Formel

$$\forall[A].$$

Beispiel 13

```
gleich(X,X).
```

entspricht der Formel

$$\forall x = (x,x).$$

Eine Anfrage

$$\texttt{?-}B_1, \ldots, B_n.$$

entspricht der Formel

$$\forall[\neg B_1 \vee \cdots \vee \neg B_n].$$

Beispiel 14

```
grossvater(X,gabi).
```

entspricht der Formel

$$\forall x \neg G(x, g).$$

Prolog versucht aus dem Programm und aus dieser Formel einen Widerspruch abzuleiten. Es zeigt also indirekt (durch Widerspruchsbeweis) die Existenz eines x mit $G(x, g)$, also eines Großvaters von Gabi. Dieser Beweis geschieht konstruktiv, d.h. Prolog konstruiert einen Wert für x so, daß dann $G(x, g)$ gilt.

Hier ist ein Beispiel für ein komplettes Prologprogramm und eine dazugehörige Anfrage.

Beispiel 15

```
elternteil(hans,peter).
elternteil(peter,gabi).
maennlich(hans).
vater(X,Y) :- elternteil(X,Y), maennlich(X).
grossvater(X,Z) :- vater(X,Y), elternteil(Y,Z).
?- grossvater(X,gabi).
```

Diese sechs Klauseln entsprechen den folgenden sechs Formeln.

$$\begin{gathered}
E(h, p) \\
E(p, g) \\
M(h) \\
\forall x \forall y \big(V(x, y) \vee \neg E(x, y) \vee \neg M(x)\big) \\
\forall x \forall y \forall z \big(G(x, z) \vee \neg V(x, y) \vee \neg E(y, z)\big) \\
\forall x \neg G(x, g).
\end{gathered}$$

Prolog startet mit der letzten dieser sechs Formeln (der Formel, die der Anfrage entspricht) und verwendet die übrigen fünf Formeln, um daraus einen Widerspruch abzuleiten. Wenn die ersten fünf Formeln gelten, kann also die letzte nicht gelten, d.h. es muß einen Wert für x geben so, daß $G(x, g)$ gilt. Im Beispiel gilt dies für $x = h$, also Hans. Prolog findet diesen Wert und gibt aus

```
X = hans.
```

Wir können die sechs Formeln zu einer zusammenfassen entweder so

$$\begin{gathered}
E(h,p) \land \\
E(p,g) \land \\
M(h) \land \\
\forall x \forall y \big(V(x,y) \lor \neg E(x,y) \lor \neg M(x)\big) \land \\
\forall x \forall y \forall z \big(G(x,z) \lor \neg V(x,y) \lor \neg E(y,z)\big) \land \\
\forall x \neg G(x,g).
\end{gathered}$$

oder, was semantisch äquivalent dazu ist, so

$$\begin{gathered}
\forall x \forall y \forall z \Big(\\
E(h,p) \land \\
E(p,g) \land \\
M(h) \land \\
\big(V(x,y) \lor \neg E(x,y) \lor \neg M(x)\big) \land \\
\big(G(x,z) \lor \neg V(x,y) \lor \neg E(y,z)\big) \land \\
\neg G(x,g)\Big).
\end{gathered}$$

Das Ergebnis ist ein Allabschluß einer Konjunktion von Disjunktionen von Literalen, wobei wir unter einem Literal eine Atomformel oder die Negation einer Atomformel verstehen. Von einer solchen Formel sagen wir, sie sei in *konjunktiver Normalform* (KNF).

3.4 Entwurfsprinzipien für Prologprogramme

3.4.1 Allgemeiner Programmentwurf

1. Grobe Analyse des Problems:
   (a) Welche Aufgaben soll das Programm lösen können? Z.B. was ist gegeben, was ist gesucht (input/output)?
   (b) Welches Wissen muß man dazu zur Verfügung stellen?
   (c) Welche Begriffe in den Aufgabenstellungen müssen definiert werden?
   (d) Welche Grundbegriffe eignen sich zur Darstellung des zur Verfügung zu stellenden Wissens? Aus diesen Grundbegriffen müssen sich die in den Aufgabenstellungen verwendeten Begriffe definieren lassen.
2. Umgangssprachliche Formulierung
   (a) Darstellung des Wissens mit Hilfe der ausgewählten Grundbegriffe.
   (b) Definition der in den Aufgabenstellungen verwendeten Begriffe mit Hilfe der Grundbegriffe.
3. Implementierung
   (a) Welche Datenstrukturen eignen sich zur Darstellung?

(b) Formulierung des Wissens als Prologklauseln (meist Fakten).

(c) Formulierung der Begriffsdefinitionen in Form von Prologklauseln (Regeln und Fakten).

(d) Formulierung der Aufgabenstellungen als Zielklauseln.

Beispiel 16
Es ist eine logische Datenbank für Verwandtschaftsbeziehungen zu implementieren.

1. Grobe Analyse des Problems:

   (a) Aufgaben: Gegeben eine Person. Gesucht sind alle Personen, die in einem bestimmten Verwandtschaftsverhältnis zu dieser Person stehen.

   (b) Hierzu muß das Wissen zur Verfügung gestellt werden, wie die Verwandtschaftsbeziehungen der Personen untereinander ausschauen.

   (c) Für die Aufgabenstellungen müssen die Begriffe 'Vater', 'Mutter', 'Sohn', 'Tochter', 'Großvater' definiert werden.

   (d) Als Grundbegriffe eignen sich 'Elternteil', 'männlich' und 'weiblich'.

2. Umgangssprachliche Formulierung

   (a) Hans ist ein Elternteil von Peter.
   Karin ist ein Elternteil von Peter.
   ...
   Gabi ist ein Elternteil von Klaus.
   Karin ist weiblich.
   ...
   Hans ist männlich.
   ...

   (b) x ist Vater von y, wenn x Elternteil von y ist und x männlich ist.
   x ist Mutter von y, wenn x Elternteil von y ist und x weiblich ist.
   x ist Sohn von y, wenn y Elternteil von x ist und x männlich ist.
   x ist Tochter von y, wenn y Elternteil von x ist und x weiblich ist.
   x ist Großvater von y, wenn x Vater eines Elternteils von y ist.

3. Implementierung

   (a) Wir müssen die Personen darstellen. Einfachste Datenstruktur: eine Konstante (klein geschriebenenr Vorname der Person). Begriffe als Prädikatszeichen, z.B. `vater(X,Z)` für „`X` ist Vater von `Y`".

   (b)
   ```
   elternteil(hans,peter).
   ...
   ```
   (s. Fragenblock 7)

   (c)
   ```
   vater(X,Y) :- elternteil(X,Y), maennlich(X).
   ...
   grossvater(X,Y) :- vater(X,E), elternteil(E,Y).
   ```

   (d)
   ```
   ?- grossvater(G,anne).
   ```

Nachdem man nun das Programm entwickelt und getestet hat, kann man auch daran gehen die Effizienz zu verbessern. Wenn man z.B. weiß, daß man das Prädikat `grossvater` vorwiegend mit einer vorgegebenen Konstanten als zweitem Argument und mit einer Variablen als erstem Argument aufgerufen wird, so kann man die Effizienz verbessern, indem man auf der rechten Seite der `grossvater`-Klausel in 3.(c) die beiden Rumpfliterale vertauscht:

```
grossvater(X,Y) :- elternteil(E,Y), vater(X,E).
```

Oft reicht eine einfache logische Spezifikation des Problems in Prolog-Syntax aus als Prolog-Programm, obwohl ein solches Programm dann oft noch ineffizient ist. So kann man das Problem eine Liste zu ordnen etwa folgendermaßen spezifizieren: „Finde zur Liste `L` eine permutierte Liste `O` so, daß `O` geordnet ist. In Prolog hätte man dann die Programmklausel

```
ordne(L,O) :- perm(L,O), geordnet(O).
```

zusammen mit Klauseln, die die Prädikate `perm` und `geordnet` so definieren, daß `perm(L,O)` zu einer Liste `L` alle permutierten Listen `O` findet und daß `geordnet(O)` für eine Liste `O` testet, ob sie geordnet ist. Das Ideal der logischen Programmierung wäre, daß das Programm aus der logischen Spezifikation einersetis und Steuerinformationen andererseits besteht. Man sagt dazu auf englisch

logic programming = logic + control

Die logische Spezifikation garantiert die Korrektheit, die Steuerinformation die Terminierung und die Effizienz. Prolog verwirklicht dieses Ideal nur zu einem gewissen Grade. Es gibt Erweiterungen von Prolog und andere Systeme der logischen Programmierung, die oft eine flexiblere Steuerung der Berechnung ohne gleichzeitige Änderung des Logikteils erlauben.

3.4.2 Zerlegung des Problems in Teilprobleme

Ein zentrales Prinzip bei der logischen Programmierung ist, wie bei allen höheren Programmiersprachen, das Prinzip der Zerlegung des zu lösenden Problems in Teilprobleme.

Beispiel 17
Man kann das Problem der Verwandlung einer aussagenlogischen Formel in KNF zerlegen in das Problem der Verwandlung der Formel in NNF und in das Problem der Verwandlung einer in NNF gegebenen Formel in KNF. In Prolog könnte man etwa ein Prologprädikat `verwandle_in_KNF` definieren durch

```
verwandle_in_KNF(F,KNF) :-
     verwandle_in_NNF(F,NNF),
     verwandle_NNF_in_KNF(NNF,KNF).
```

Man muß dann nur noch die Prologprädikate `verwandle_in_NNF` und `verwandle_NNF_in_KNF` definieren. Man hat also die ursprüngliche Programmieraufgabe auf zwei einfachere Programmieraufgaben zurückgeführt.

3.4.3 Rekursion

Bei der Rekursion führt man ein Problem auf eine einfachere Instanz desselben Problems zurück.

Beispiel 18
Um die Zahl $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$ zu berechnen, genügt es die Zahl $4! = 1 \cdot 2 \cdot 3 \cdot 4$ zu berechen, was einfacher ist, und dann das Resultat mit 5 zu multiplizieren. Allgemein $n! = (n-1)! \cdot n$, wenn $n > 0$ ist. In Prolog kann man ein zweistelliges Prädikat `fak` definieren, wobei `fak(N,F)` bedeutet $N! = F$. Dann schaut das entsprechende rekursive Programm so aus.

```
fak(0,1).
fak(N,F) :- N>0, N1 is N-1, fak(N1,F1), F is F1*N.
```

Man kann jetzt an Prolog eine Anfrage stellen.

```
?- fak(5,F).
F = 120
```

Die erste Klausel des Programms besagt, daß $0! = 1$ ist. In der zweiten Klausel darf man auf der rechten Seite nicht `fak(N-1,F1)` schreiben, da `N-1` für Prolog ein komplexer Term ist, der normalerweise nicht ausgewertet wird. Wenn der arithmetische Ausdruck `N-1` als rechtes Argument von `is` verwendet wird, wie im Beispiel, dann wird dieser Ausdruck ausgewertet. Aus dem gleichen Grund darf der Kopf der Klausel nicht lauten `fak(N,F1*N)`. Auswertung von arithmetischen Ausdrücken muß in Prolog immer explizit verlangt werden.

3.4.4 Das Prinzip der Verallgemeinerung

In einigen Fällen ist das folgende Vorgehen sinnvoll. Um ein Problem zu lösen, verallgemeinert man das Problem und versucht das verallgemeinerte Problem durch Rekursion zu lösen.

Ein Beispiel dafür ist das 8-Damen-Problem. Acht Damen sind so auf einem Schachbrett zu positionieren, daß keine eine der anderen bedroht (s. Aufgabenblock 11, Aufgabe 2). Ein mögliches Vorgehen besteht darin zunächst eine der Damen in der ersten horizontalen Reihe zu positionieren. Dann muß man nur noch das einfachere Problem lösen die restlichen sieben Damen so positionieren, daß sie sich nicht gegenseitig, aber auch nicht die erste Dame bedrohen. Hier läßt sich aber Rekursion nicht in der Weise anwenden wie beim Fakultätsproblem, da es sich hier beim Teilproblem nicht einfach um das 7-Damen-Problem handelt. Der Sachverhalt wird noch deutlicher, wenn man noch einen Schritt weiter geht. Um die restlichen sieben Damen zu positionieren, kann man etwa wiederum zunächst eine zweite Dame in der zweiten horizontalen Reihe positionieren und muß dann die restlichen sechs Damen so positionieren, daß sie sich nicht gegenseitig, aber auch nicht die ersten beiden Damen bedrohen. Allgemein hat man das Problem zu lösen, daß man eine Anzahl (null oder mehr) Damen bereits in den ersten horizontalen Reihen positioniert hat und nun noch die restlichen positionieren möchte. Dies ist eine Verallgemeinerung des ursprünglichen Problems.

3.4.5 Nichtdeterminismus

Systeme der logischen Programmierung sind nichtdeterministisch in dem Sinne, daß eine Anfrage mehrere Antworten haben kann. Das System findet, sofern die Anfrage terminiert, alle korrekten Antworten. Im Beispiel des 8-Damen-Problems kann man das System zunächst die erste Dame nichtdeterministisch in der ersten horizontalen Reihe positionieren laßen. Man kann also Prolog eine Anweisung geben der Art

Wähle irgendein Feld der ersten horizontalen Reihe.

Findet Prolog für diese Position der ersten Dame keine Lösung, sucht es automatisch eine Lösung für die zweite mögliche Position der ersten Dame. Prolog sucht alle Wahlmöglichkeiten automatisch durch. Die Suche muß nicht explizit implementiert werden. Sie ist bereits im Prologsystem eingebaut.

3.4.6 Zwei Sichtweisen für ein Prolog-Programm

1. Logische Sichtweise
2. Prozedurale Sichtweise

Man kann ein Prolog-Programm von der logischen oder von der prozeduralen Sichtweise aus betrachten. Als Beispiel betrachten wir das Fakultätsprogramm

```
fak(0,1).
fak(N,F) :- N>0, N1 is N-1, fak(N1,F1), F is F1*N.
```

Logische Sichtweise:

- Es gilt $0! = 1$.
- Wenn $n > 0$ ist und $n_1 = n - 1$ ist und $n_1! = f_1$ ist und $f = f_1 \cdot n$ ist, dann ist $n! = f$.
  Anders ausgedrückt bedeutet dies:
  Wenn $n > 0$ ist, dann ist $n! = (n-1)! \cdot n$.

Prozedurale Sichtweise:
Im `fak`-Beispiel soll `fak` so aufgerufen werden, daß das erste Argument zum Zeitpunkt des Aufrufs voll instantiiert ist, d.h. keine Variablen enthält. Bei `fak(N,F)` ist also `N` als natürliche Zahl gegeben.

- Wenn $n = 0$ ist, dann ist 1 eine korrekte Antwort für $n!$.
- Wenn $n > 0$ ist, dann berechnet sich eine korrekte Antwort f für $n!$ folgendermaßen:
  Sei $n_1 = n - 1$, $f_1 = n_1!$, $f = f_1 \cdot n$.

Nachteile der prozeduralen Sichtweise:

- Logische Korrektheit des Programms schwerer nachzuweisen.
- Programm oft schwerer lesbar, als wenn es ausgehend von der logischen Sichtweise entwickelt wurde.

- Prädikate häufig nur eingeschränkt (z.B. nur 'in einer Richtung') anwendbar.

Vorteile der prozeduralen Sichtweise:

- Terminierung leichter erkennbar.
- Effizienzverbesserungen leichter erkennbar.

Es ist wichtig beides, die logische und die prozedurale Sichtweise zugleich anzuwenden. Im Anfangsstadium der Programmentwicklung hauptsächlich logische Sichtweise, später zunehmend auch prozedurale Sichtweise, aber ohne die logische Sichtweise aus den Augen zu verlieren!!!

3.5 Normalformen

Ähnlich wie in der Aussagenlogik gibt es auch in der Prädikatenlogik den Begriff der Normalformen. Auch hier gibt es Negationsnormalform, konjunktive Normalform, disjunktive Normalform und eine Reihe weiterer Normalformen. Wir interessieren uns besonders für die konjunktive Normalform.

Definition 34 Eine Formel heißt ein *Literal*, wenn sie eine Atomformel oder die Negation einer Atomformel ist. Im ersteren Fall heißt sie ein *positives Literal*, im letzteren Fall ein *negatives Literal*.

Beispiel 19
Seien x eine Variable, a und b zwei Konstanten, P ein zweistelliges Prädikatszeichen und f ein zweistelliges Funktionszeichen. Dann ist $P(a, f(x, b))$ ist ein positives Literal und $\neg P(f(x, x), f(b, x))$ ein negatives Literal.

Definition 35 Eine Formel F ist in *konjunktiver Normalform* (*KNF*), wenn sie ein Allabschluß einer Konjunktion von Disjunktionen von Literalen ist.

Beispiel 20
Die Formel

$$\begin{gathered}\forall x \forall y \forall z \Big(\\ E(h,p) \wedge \\ E(p,g) \wedge \\ M(h) \wedge \\ \big(V(x,y) \vee \neg E(x,y) \vee \neg M(x)\big) \wedge \\ \big(G(x,z) \vee \neg V(x,y) \vee \neg E(y,z)\big) \wedge \\ \neg G(x,g)\Big).\end{gathered}$$

ist in konjunktiver Normalform.

Zu einer Formel der Prädikatenlogik gibt es im allgemeinen keine semantisch äquivalente Formel in KNF. Zum Beispiel gibt es keine Formel in KNF, die semantisch äquivalent ist zu $\exists x\, P(x)$. Es gilt aber der folgende Satz.

Satz 12 *Zu jeder Formel F kann man effektiv eine Formel F' in KNF konstruieren derart, daß F genau dann erfüllbar ist, wenn F' erfüllbar ist.*[6]

Wenn für zwei Formeln F und F' gilt, daß F genau dann erfüllbar ist, wenn F' erfüllbar ist, so sagen wir auch, F und F' seien *erfüllbarkeitsäquivalent*. In der Prädikatenlogik gilt also im Gegensatz zur Aussagenlogik nicht die semantische Äquivalenz einer Formel mit ihrer KNF, sondern lediglich die Erfüllbarkeitsäquivalenz. Da ein Prologsystem aber ein Theorembeweiser ist, der die Unerfüllbarkeit einer Formel nachzuweisen versucht, genügt die Erfüllbarkeitsäquivalenz um zu sehen, daß bei Prolog die Beschränkung auf KNF keine so gravierende Einschränkung ist.

3.6 Klauseln

Definition 36 Eine *Klausel* ist eine endliche Menge von Literalen.

Eine Formel in KNF läßt sich durch eine endliche Menge von Klauseln darstellen. Und zwar hat ja nach Definition die Formel die Form eines Allabschlusses einer Konjunktion von Disjunktionen von Literalen. Jeder dieser Disjunktionen von Literalen ordnen wir eine Klausel zu, nämlich die Menge aller Literale der Disjunktion.

Beispiel 21
Die obige Formel in KNF entspricht der Klauselmenge, die aus den folgenden Klauseln besteht.

$$\begin{gathered}
\{E(h,p)\} \\
\{E(p,g)\} \\
\{M(h)\} \\
\{V(x,y), \neg E(x,y), \neg M(x)\} \\
\{G(x,z), \neg V(x,y), \neg E(y,z)\} \\
\{\neg G(x,g)\}.
\end{gathered}$$

Definition 37 Eine *Hornklausel* ist eine Klausel, die höchstens ein positives Literal enthält. Eine *Programmklausel* ist eine Klausel, die genau ein positives Literal enthält. Eine *Zielklausel* ist eine Klausel, die kein positives Literal enthält.

Im Beispiel sind alle Klauseln Hornklauseln, und zwar sind die ersten fünf Klauseln Programmklauseln und die sechste Klausel ist eine Zielklausel. Prolog arbeitet stets mit Hornklauseln.

[6]Auf diesen Satz soll hier nicht näher eingegangen werden. Die Idee ist die, daß man die Existenzquantoren in F irgendwie eliminieren muß, da eine Formel in KNF keine Existenzquantoren enthalten darf. Wenn nun etwa $\forall x \exists y\, P(x,y)$ gilt, so muß es eine Funktion geben, die jedem Wert für x einen entsprechenden Wert für y zuordnet, es muß also die Formel $\forall x\, P(x, f(x))$ erfüllbar sein. Diese Einführung von Funktionszeichen nennt man *Skolemisierung*. Das neu eingeführte Funktionszeichen f nennt man *Skolemfunktionszeichen*, oft auch einfach *Skolemfunktion*.

3.7 Resolution

Betrachten wir nochmal das Großvater-Beispiel.

```
...
grossvater(X,Z) :- vater(X,Y), elternteil(Y,Z).
?- grossvater(X,gabi).
```

Es soll grob skizziert werden, wie eine solche Anfrage von Prolog gelöst wird. Prolog startet mit der Anfrage. Das Ziel `grossvater(X,gabi)` ist für einen geeigneten Wert von `X` zu beweisen. Hierzu sucht Prolog im Programm eine Klausel, deren Kopf das Prädikatszeichen `grossvater` hat und findet

```
grossvater(X,Z) :- vater(X,Y), elternteil(Y,Z).
```

`grossvater(X,gabi)` ist also dann bewiesen, wenn man `vater(X,Y)` und `elternteil(Y,gabi)` für einen geeigneten Wert von `Y` beweist. Prolog erzeugt also eine neue Anfrage

```
?- vater(X,Y), elternteil(Y,gabi).
```

die es seinerseits zu lösen versucht. Wie hat sich diese neue Anfrage ergeben? Hierzu mußte das Ziel `grossvater(X,Z)` der ursprünglichen Anfrage und der Kopf `grossvater(X,Z)` der Klausel des Prologprogramms einander gleich gemacht (wir sagen *unifiziert*) werden durch Ersetzung (wir sagen *Substitution*) der Variablen `Z` durch `gabi`. Also

```
grossvater(X,gabi) :- vater(X,Y), elternteil(Y,gabi).
?- grossvater(X,gabi).
```

Die neue Anfrage ergibt sich nun durch Resolution dieser beiden Prologklauseln in der Form, in der wir sie bereits aus der Aussagenlogik kennen. Wir werden später die Resolution für die Prädikatenlogik genauer definieren. Hier halten wir nur grob fest, daß sich in der Prädikatenlogik einen Resolutionsschritt aufgeteilt in drei Schritte vorstellen kann: eine *Unifikation*, Anwendung einer *Substitution* auf die Elternklauseln und eine einfache aussagenlogische Resolution, wie wir sie bereits kennengelernt haben.

3.7.1 Unifikation

Es geht darum zwei Terme durch eine Substitution gleich zu machen (Gleichungslösen).

Beispiel 22
Das zweistellige Prädikat $=$ ist in Prolog vordefiniert durch

```
=(X,X).
```

Stellt man nun an Prolog die Anfrage

```
?- =( f(X,g(X)) , f(g(a),Y) ).
```

so muß Prolog die Gleichung

```
f(X,g(X)) = f(g(a),Y)
```

lösen. Für `X` und `Y` sollen Terme gefunden werden, die diese Gleichung erfüllen, egal durch was für Funktionen die Funktionszeichen `f` und `g` interpretiert werden. Dies ist nur möglich, wenn gleichzeitig die beiden Gleichungen

```
X = g(a)                    g(X) = Y
```

gelöst werden. Die Lösung lautet

```
X = g(a)
Y = g(g(a)).
```

Die ursprüngliche Gleichung wird also gelöst, indem man darin die Variable `X` durch den Term `g(a)` und die Variable `Y` durch den Term `g(g(a))` ersetzt. Wir sprechen von einer *Substitution*, die wir folgendermaßen schreiben.

$$\{\mathtt{X} \leftarrow \mathtt{g(a)}, \mathtt{Y} \leftarrow \mathtt{g(g(a))}\}$$

Mathematisch ist eine Substitution dadurch gegeben, daß wir zu jeder Variablen den Term angeben, durch den die Variable bei der Substitution ersetzt wird.

Definition 38 Eine *Substitution* ist eine Funktion σ, die jeder Variablen x einen Term $\sigma(x)$ zuordnet, wobei nur für endlich viele Variablen x gilt: $\sigma(x) \neq x$.

Beispiel 23
Sei

$$\begin{aligned}
&\sigma(x) = g(a) \\
&\sigma(y) = g(g(a)) \\
&\sigma(z) = z \qquad \text{für alle } z\text{, die nicht identisch mit } x \text{ oder } y \text{ sind.}
\end{aligned}$$

Diese Substitution σ wird geschrieben als $\{x \leftarrow g(a), y \leftarrow g(g(a))\}$.

Definition 39 Seien $x_1, \ldots, x_n$ Variablen und $t_1, \ldots, t_n$ Terme und die Substitution σ sei definiert durch

$$\begin{aligned}
\sigma(x_i) &= t_i \\
\sigma(x) &= x \qquad \text{für alle } x\text{, die nicht identisch mit einem der } x_i \text{ sind.}
\end{aligned}$$

Dann wird σ bezeichnet mit $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$.

Die Ausgabe von Prolog ist, sofern sie nicht '`No`' lautet, eine Substitution. Wenn man zum Beispiel auf die Anfrage

```
?- grossvater(X,gabi).
```

von Prolog die Antwort

```
X = hans
```

bekommt, so so bedeutet dies die Substitution $\{\mathtt{X} \leftarrow \mathtt{hans}\}$. Die Ausgabe `Yes` bedeutet die leere Substitution $\{\}$, die auch mit ε bezeichnet wird. Es ist $\varepsilon(x) = x$ für alle Variablen x.

Definition 40 Die *Anwendung* einer Substitution σ auf einen Term t ergibt einen Term $t\sigma$, der folgendermaßen definiert ist.

$$x\sigma := \sigma(x)$$
$$f(t_1, \ldots, t_n)\sigma := f(t_1\sigma, \ldots, t_n\sigma).$$

Beispiel 24
$f(x, g(x))\ \{x \leftarrow g(a), y \leftarrow g(g(a))\} = f(g(a), g(g(a)))$.

Definition 41 Seien σ und τ zwei Substitutionen. Dann ist die *Komposition* $\sigma\tau$ von σ und τ folgendermaßen definiert: $(\sigma\tau)(x) := (\sigma(x))\tau$.

Beispiel 25
Sei $\sigma = \{x \leftarrow f(x, y), y \leftarrow g(z)\}$ und $\tau = \{y \leftarrow g(y), z \leftarrow a\}$. Dann ist $\sigma\tau = \{x \leftarrow f(x, g(y)), y \leftarrow g(a), z \leftarrow a\}$.

Wenn σ und τ zwei Substitutionen sind und t ein Term ist, dann gilt $t(\sigma\tau) = (t\sigma)\tau$.

Definition 42 Seien s und t zwei Terme. Unter einem *Unifikator* von s und t verstehen wir eine Substitution σ so, daß $s\sigma = t\sigma$ ist. Wenn zwei Terme s und t einen Unifikator haben, dann sagen wir, sie seien *unifizierbar*.

Beispiel 26
Die Terme $f(x, b)$ und $f(a, y)$ haben den Unifikator $\{x \leftarrow a, y \leftarrow b\}$.
Die Terme a und $f(x)$ haben keinen Unifikator.
Die Terme x und $f(x)$ haben keinen Unifikator.
Jede Substitution ist Unifikator der Terme x und x.
Die Terme x und $f(y)$ haben den Unifikator $\{x \leftarrow f(y)\}$, aber auch den Unifikator $\{x \leftarrow f(g(z)), y \leftarrow g(z), u \leftarrow g(f(z))\}$.

Definition 43 Eine Substitution σ ist *allgemeiner* als eine Substitution τ, wenn es eine Substitution ρ gibt mit $\tau = \sigma\rho$.

Wenn ein Unifikator σ zweier Terme s und t allgemeiner ist als eine Substitution τ, dann ist auch τ ein Unifikator von s und t. Im letzten Beispiel ist der erste angegebene Unifikator von x und $f(y)$ allgemeiner als der zweite.

Definition 44 Eine Substitution σ heißt *allgemeinster Unifikator* (englisch: *most general unifier, mgu*) zweier Terme s und t, wenn σ ein Unifikator von s und t ist und außerdem σ allgemeiner als jeder Unifikator von s und t ist.

Satz 13 *Zwei unifizierbare Terme haben immer auch einen allgemeinsten Unifikator.*

Unter einem *Unifikationsalgorithmus* versteht man einen Algorithmus, der von zwei Termen festellt, ob sie unifizierbar sind, und im Falle der Unifizierbarkeit einen allgemeinsten Unifikator liefert.

Es gibt effiziente Unifikationsalgorithmen, deren Laufzeit linear mit der Länge der Terme anwächst. Da aber Prolog bei jedem Berechnungsschritt eine Unifikation durchführen muß und die Terme oft äußerst komplexe Datenstrukturen darstellen und daher sehr lang sind, genügt diese Effizienz in der Praxis noch nicht. Prolog verwendet einen in vielen Fällen noch effizienteren Algorithmus. Will Prolog zum Beispiel eine Variable x mit einem langen Term t unifizieren, so ermittelt es als Unifikator einfach die Substitution $\{x \leftarrow t\}$. Es hat den Term t bereits im Speicher und braucht ihn nicht erneut aufzubauen. Dies funktioniert allerdings nur, wenn x in t nicht vorkommt. Wenn die Möglichkeit besteht, daß x in t vorkommen könnte, muß für korrekte Unifikation t danach durchsucht werden, ob x darin vorkommt (englisch: occurs). Man nennt das den 'occur check'. In den meisten praktischen Anwendungen ist der 'occur check' nicht nötig. Prolog verwendet in seinem Unifikationsalgorithmus standardmäßig keinen 'occur check', in einigen wenigen Prologversionen läßt er sich aber optional einschalten.

Da für Prolog ein Prädikatszeichen sich syntaktisch nicht von einem Funktionszeichen unterscheidet und daher eine Atomformel sich syntaktisch nicht von einem Term unterscheidet, kann man die Begriffe der Anwendung einer Substitution und der Unifizierbarkeit sowie den Unifikationsalgorithmus genausogut für Atomformeln wie für Terme verwenden. So ist zum Beispiel $P(t_1, \ldots, t_n)\sigma \equiv P(t_1\sigma, \ldots, t_n\sigma)$. Wenn A eine Atomformel und σ eine Substitution ist, so bezeichne $(\neg A)\sigma$ die Formel $\neg(A\sigma)$. Wenn $\{L_1, \ldots, L_n\}$ eine Klausel ist, so werden wir auch $\{L_1, \ldots, L_n\}\sigma$ schreiben für $\{L_1\sigma, \ldots, L_n\sigma\}$. In Prolog-Notation schreiben wir `?-` $A_1, \ldots, A_n.\sigma$ für `?-` $A_1\sigma, \ldots, A_n\sigma.$ und A `:-` $B_1, \ldots, B_n.\sigma$ für $A\sigma$ `:-` $B_1\sigma, \ldots, B_n\sigma.$

3.7.2 SLD-Resolution

Der Resolutionskalkül läßt sich von der Aussagenlogik auf die Prädikatenlogik übertragen. Da wir es in Prolog mit Hornklauseln und dem speziellen Begriff der SLD-Resolution zu tun haben, werden wir hier die allgemeine Resolution nicht einführen, sondern gleich zur SLD-Resolution kommen.

Definition 45 Zwei Klauseln heißen *variablendisjunkt*, wenn sie keine gemeinsamen Variablen haben.

Beispiel 27
Die beiden Klauseln

```
grossvater(X,Z) :- vater(X,Y), elternteil(Y,Z).
?- grossvater(X,gabi).
```

sind nicht variablendisjunkt, da beide Klauseln die Variable `X` enthalten. Dagegen sind die beiden Klauseln

```
grossvater(X',Z) :- vater(X',Y), elternteil(Y,Z).
?- grossvater(X,gabi).
```

variablendisjunkt.

Der Begriff eines SLD-Resolutionsschrittes läßt sich am einfachsten erklären, wenn man zunächst annimmt, daß die beteiligten Elternklauseln variablendisjunkt sind. Dann schaut ein SLD-Resolutionsschritt folgendermaßen aus.

$$\frac{\texttt{?-}\ A_1, \ldots, A_j, \ldots, A_m. \qquad\qquad A\ \texttt{:-}\ B_1, \ldots, B_n.}{\texttt{?-}\ A_1, \ldots, A_{j-1}, B_1, \ldots, B_n, A_{j+1}, \ldots, A_m.\sigma}.$$

Dabei müssen A und A_j unifizierbar sein und σ muß ein allgemeinster Unifikator von A und A_j sein. Sind die Elternklauseln nicht variablendisjunkt, müssen vorher in der Programmklausel Variablen umbenannt werden wie im Beispiel oben.

Definition 46 Eine *Variante* einer Klausel c ist das Ergebnis einer Variablenumbenennung in c, d.h. einer Ersetzung von Variablen durch andere Variablen, wobei gleiche Variablen durch gleiche, verschiedene durch verschiedene ersetzt werden.

Beispiel 28

`grossvater(X',Z) :- vater(X',Y), elternteil(Y,Z).`
ist eine Variante der Klausel

```
grossvater(X,Z) :- vater(X,Y), elternteil(Y,Z).
```

In der Prädikatenlogik stellt eine Klausel einen Allabschluß dar. Daher ist eine Variante einer Klausel genau dann wahr, wenn die ursprüngliche Klausel wahr ist.

Die allgemeine SLD-Resolutionsregel schaut nun so aus.

$$\frac{\texttt{?-}\ A_1,\ldots,A_j,\ldots,A_m. \qquad\qquad A\ \texttt{:-}\ B_1,\ldots,B_n.}{\texttt{?-}\ A_1,\ldots,A_{j-1},B'_1,\ldots,B'_n,A_{j+1},\ldots,A_m.\sigma},$$

wobei $A'\texttt{:-}B'_1,\ldots,B'_n.$ eine Variante von $A\texttt{:-}B_1,\ldots,B_n.$ und variablendisjunkt zu $\texttt{?-}A_1,\ldots,A_m.$ ist und σ ein mgu von A_j und A' ist.

Da in Prolog das ausgewählte Literal immer das erste Literal der Zielklausel ist, hat die SLD-Resolutionsregel für Prolog die Form

$$\frac{\texttt{?-}\ A_1,\ldots,A_m. \qquad\qquad A\ \texttt{:-}\ B_1,\ldots,B_n.}{\texttt{?-}\ B'_1,\ldots,B'_n,A_2,\ldots,A_m.\sigma},$$

wobei $A'\texttt{:-}B'_1,\ldots,B'_n.$ eine Variante von $A\texttt{:-}B_1,\ldots,B_n.$ und variablendisjunkt zu $\texttt{?-}A_1,\ldots,A_m.$ ist und σ ein mgu von A_1 und A' ist.

In anderen Systemen der logischen Programmierung kann das ausgewählte Literal auch ein anderes Literal der Zielklausel sein.

Der Begriff der *SLD-Resolutionsableitung* ist analog definiert wie in der Aussagenlogik. Eine *SLD-Widerlegung* einer Klauselmenge ist eine SLD-Resolutionsableitung der leeren Klausel aus dieser Klauselmenge.

Beispiel 29
Betrachten wir das folgende Prologprogramm

```
p(X,Z) :- q(X,Y), p(Y,Z).
p(X,X).
q(a,b).
```

mit der Anfrage

```
?- p(X,b).
```

Dann ist eine mögliche SLD-Widerlegung der Klauselmenge, die aus diesen vier Klauseln besteht, die folgende.

```
?-p(X,b).          p(X,Z):-q(X,Y),p(Y,Z).
        \          /
   ?-q(X,Y),p(Y,b).             q(a,b).
            \                  /
             ?-p(b,b).                p(X,X).
                  \                  /
```

Im ersten Resolutionsschritt werden die beiden Klauseln

```
?- p(X,b).
p(X,Z) :- q(X,Y), p(Y,Z).
```

miteinander resolviert. Dazu muß in der Programmklausel

```
p(X,Z) :- q(X,Y), p(Y,Z).
```

die Variable `X` umbenannt werden, sagen wir in `X'`. Es wird also von dieser Klausel die Variante

```
p(X',Z) :- q(X',Y), p(Y,Z).
```

gebildet. Der Kopf `p(X',Z)` dieser Klausel ist mit dem Ziel `p(X,b)` der Zielklausel `?-p(X,b).` unifizierbar mit dem mgu $\{\mathtt{Z} \leftarrow \mathtt{b}, \mathtt{X'} \leftarrow \mathtt{X}\}$. Beim zweiten Resolutionsschritt ist keine Variablenumbenennung nötig. Der mgu ist dort $\{\mathtt{X} \leftarrow \mathtt{a}, \mathtt{Y} \leftarrow \mathtt{b}\}$. Im dritten Schritt ist eine Umbenennung der Variablen `X` in der Programmklausel `p(X,X).` in `X"` sinnvoll, da die Variable `X` bereits in der ursprünglichen Anfrage vorkommt. Man erhält dann als Variante der Programmklausel das Faktum `p(X",X").` und als mgu die Substitution $\{\mathtt{X"} \leftarrow \mathtt{b}\}$. Insgesamt wurde die in der ursprünglichen Anfrage vorkommende Variable `X` durch `a` ersetzt. Wir sagen, die Substitution $\{\mathtt{X} \leftarrow \mathtt{a}\}$ sei die *berechnete Antwortsubstitution*. Die berechnete Antwortsubstitution wird von Prolog ausgegeben in der Form

```
X = a
```

3.8 Berechnete und korrekte Antwortsubstitutionen

Definition 47 Sei P eine Menge von Programmklauseln und G eine Zielklausel.

3.9 SLD-Bäume

********* Prolog als Programmiersprache (auch nichtdeklarative Dinge).

Inhaltsverzeichnis