3. Logik-Programmierung

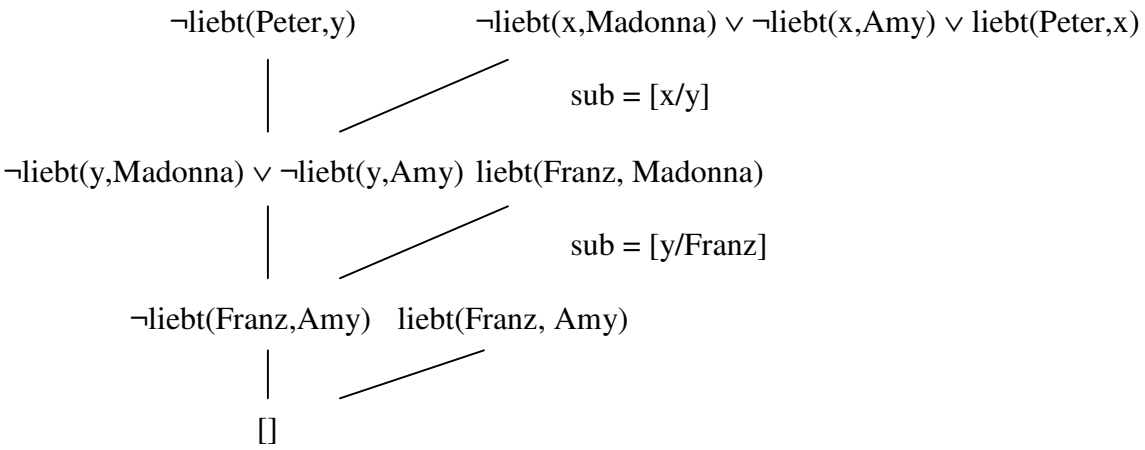
3.1. Vorbemerkungen

Idee: Ausführen eines Logik-Programms entspricht Herleitung leerer Klausel. Zusätzliche Verwendung einer Antworterzeugungskomponente (liefert Rechenergebnis).
Erzeugen von Antworten: Beispiel

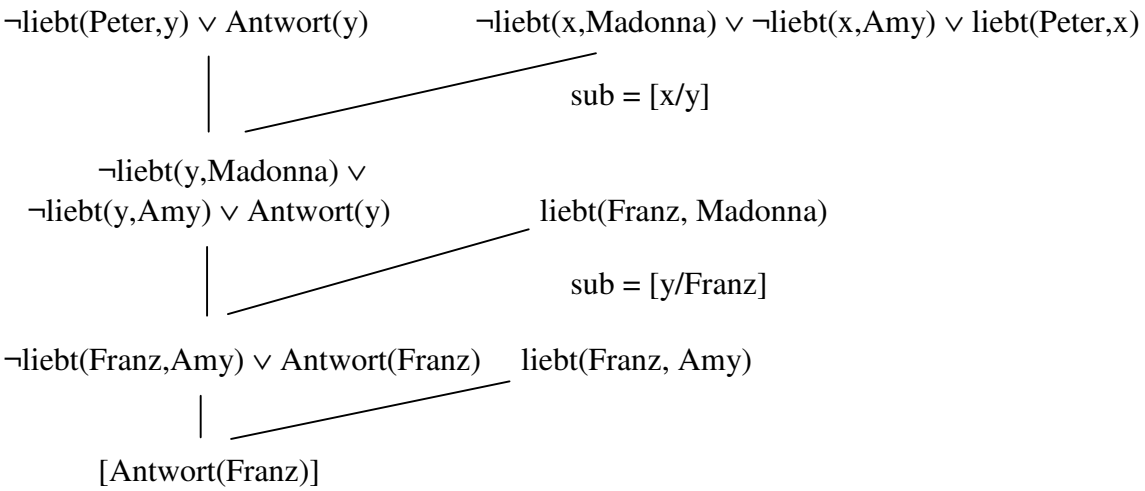
F: liebt(Franz, Amy),
liebt(Franz, Madonna),
¬liebt(x,Madonna) ∨ ¬liebt(x,Amy) ∨ liebt(Peter,x)

"Aufruf" des Programms: Gibt es jemanden, den Peter liebt? G = ∃y liebt(Peter,y)
Gesucht nicht nur JA-NEIN-Antwort, sondern falls JA auch das betreffende Individuum.

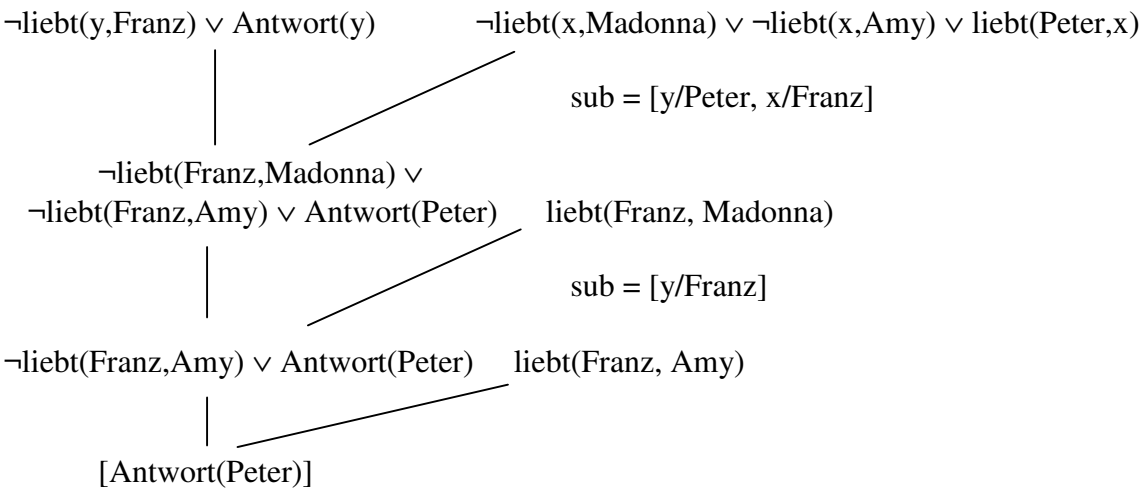
Herleitung der leeren Klausel:



Gesuchtes Individuum kann aus Substitution für y abgelesen werden: y = Franz
Kann auch hergeleitet werden durch Einführung eines Antwortprädikats: statt der Klausel für das negierte Ziel ¬liebt(Peter,y) verwendet man: ¬liebt(Peter,y) ∨ Antwort(y). Jetzt wird nicht mehr leere Klausel abgeleitet, sondern Klausel, die nur das Antwortprädikat enthält:

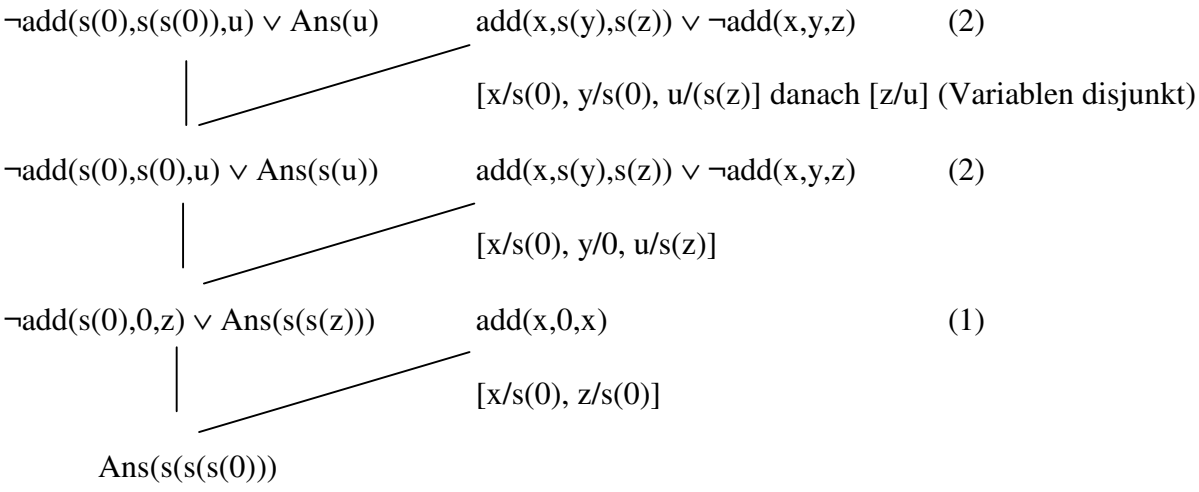


Zu beachten: in obigem Madonna-Beispiel wurde in 2stelligem Zielprädikat "liebt" das erste Argument als Eingabeparameter verwendet (Peter), das zweite als Ausgabeparameter (Wen liebt Peter)? Dasselbe Programm kann aber auch verwendet werden, um etwa zu prüfen, wer Franz liebt. man spricht von Invertibilität der Parameter.

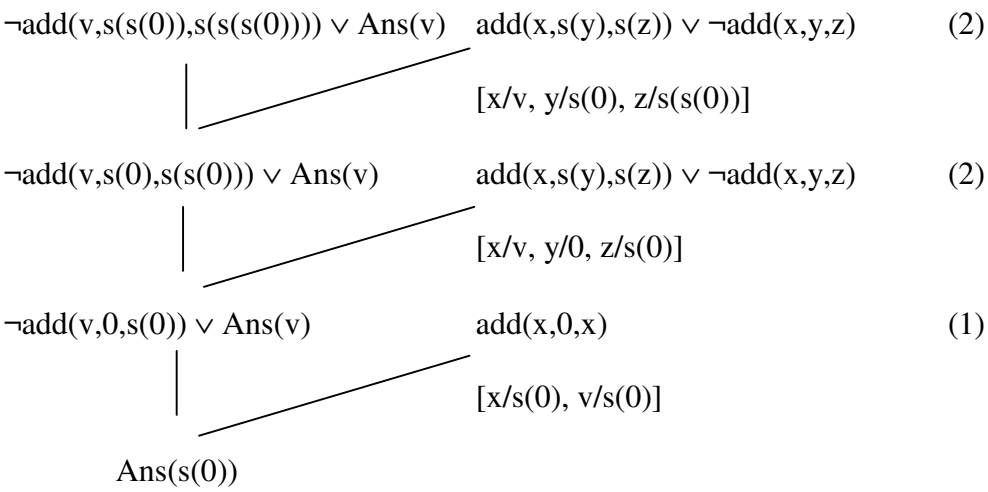


Bsp. Invertibilität: Programm

add(x,0,x) (1)
add(x,s(y),s(z)) ∨ ¬add(x,y,z) (2)



kann auch für Subtraktion verwendet werden:



Planungsbeispiel:
Peter braucht Bananen, hat kein Geld, aber ein Konto bei der Bank. Bananen gibt es bei Aldi. Peter kann alle relevanten Örtlichkeiten zu Fuß erreichen.

Idee: wenn Peter in einer Situation s Objekt t besitzt, so schreiben wir Has(t,s).
Aktionen (wie Buy-Bananas) sind Funktionen, die von einer Situation in einen Nachfolgesituation führen.

Ziel: ∃s Has(Bananas, s)
Programm:

Has(Bananas,Buy-Bananas(s)) ∨ ¬At-Aldi(s) ∨ ¬Has(Money,s)	1
Has(Money,Get-Cash(s)) ∨ ¬ At-Bank(s)	2
At-Bank(Go(Bank,s))	3
At-Aldi(Go(Aldi,s))	4
Has(Money,Go(x,s)) ∨ ¬Has(Money(s))	5
¬Has(Bananas, s) ∨ Antwort(s)	6 +1
¬At-Aldi(s) ∨ ¬Has(Money,s) ∨ Antwort(Buy-Bananas(s))	7 +4
¬Has(Money,Go(Aldi,s)) ∨ Antwort(Buy-Bananas(Go(Aldi,s)))	8 +5
¬Has(Money,s) ∨ Antwort(Buy-Bananas(Go(Aldi,s)))	9 +2
¬At-Bank(s) ∨ Antwort(Buy-Bananas(Go(Aldi,Get-Cash(s))))	10 +3
Antwort(Buy-Bananas(Go(Aldi,Get-Cash(Go(Bank,s)))))	

3.2 Hornklauselprogramme

Man beschränkt sich beim logischen Programmieren oft auf Hornklauseln:
recht ausdrucksmächtig, Antworterzeugung einfach, effizient: SLD-Resolution vollständig.

Fakten (Tatsachenklauseln): einelementige positive Klauseln.

Regeln (Prozedurklauseln): gemischte Hornklauseln: Notation in Prolog[1]: P :- Q_1,..., Q_k
P heißt Kopf, Q_1,...,Q_n Körper der Regel.
Logik-Programm: endliche Menge von Fakten und Regeln.
Aufruf eines Logikprogramms durch Zielklausel: :- Q_1,...,Q_k (man schreibt auch ?- Q_1,...,Q_k)

Wir werden im Folgenden sowohl Klausel- wie Prolog-Regel-Notation verwenden.

Prozedurale Interpretation von P :- Q_1,..., Q_k:
wenn P abgeleitet werden soll, dann leite Q_1, ..., Q_k ab.

Annahme: Variablenumbenennungen immer in Programmklauseln: standardisierte SLD-Resolution.

Def.: Konfiguration, Konfigurationsübergang
Sei F ein Logikprogramm. Eine Konfiguration ist ein Paar (G,sub), wobei G Zielklausel und sub Substitution ist. Ein Konfigurationsübergang (bzgl. F) wird wie folgt definiert:
(G_1,sub_1) |-- (G_2,sub_2) falls gilt:
$G_1 = \{\neg A_1,...,\neg A_k\}$ und es gibt eine Programmklausel $K = \{B,\neg C_1,..., \neg C_n\}$ in F die durch Variablenumbenennung bereits keine Variablen mit G_1 mehr gemeinsam hat. B und A_i seien durch mgu s unifizierbar. G_2 hat die Form

$$\{\neg A_1,...,\neg A_{i-1}, \neg A_{i+1},...,\neg A_k,\neg C_1,..., \neg C_n\}s$$

und $sub_2 = sub_1$ s.

Eine Berechnung von F bei Eingabe G_1 ist eine Folge von Konfigurationen (G_i,sub_i) so dass gilt $sub_1 = []$ und (G_i,sub_i) |-- (G_{i+1}, sub_{i+1}), für alle $i \geq 1$.
Falls die Berechnung endlich ist und mit der Konfiguration ([],sub) terminiert, so heißt die Berechnung erfolgreich und $(A_1 \wedge ... \wedge A_k)$sub ist das Rechenergebnis.

Also: standardisierte SLD-Resolutionen, wobei Substitution mitgeführt wird.
Bemerkung: da die berechnete Substitution nur dazu verwendet wird, das Rechenergebnis zu erzeugen, genügt es, sich die Ersetzungen von Variablen in G_1 zu „merken“.

Berechnungen nichtdeterministisch: Konfiguration kann mehrere Nachfolgekonfigurationen haben. Mögliche Berechnungen bilden Baum: endlich verzweigt, aber möglicherweise unendliche Pfade.

Beispiel: P(x) :- P(f(x))
P(a)

Ziel: :- P(a)

({¬P(a)}, []) |-- ({¬P(f(a))}, [x/a]) |-- ({¬P(f(f(a)))}, [x/a][x/f(a)]) |-- ... unendlich

|-- ([], [])

[1] Prolog ist eine Klasse von Logikprogrammiersystemen.

Satz: (Clark) Sei F ein Logik-Programm und G = :- $A_1,...,A_k$ eine Zielklausel.
1. (Korrektheit) Falls es eine erfolgreiche Berechnung von F bei Eingabe G gibt, so ist jede Grundinstanz des Rechenergebnisses $(A_1 \wedge ... \wedge A_k)$sub Folgerung von F.
2. (Vollständigkeit) Falls jede Grundinstanz von $(A_1 \wedge ... \wedge A_k)$sub' Folgerung von F ist, so gibt es eine erfolgreiche Rechnung von F bei Eingabe G mit Rechenergebnis $(A_1 \wedge ... \wedge A_k)$sub, so dass für eine geeignete Substitution s

$$(A_1 \wedge ... \wedge A_k)sub' = (A_1 \wedge ... \wedge A_k)sub\ s$$

Def.: Prozedurale vs. modelltheoretische Semantik
Sei F ein Logik-Programm, G = :- $A_1,..., A_k$ Zielklausel.
Die prozedurale Semantik von (F,G) ist die Menge

$$Sp(F,G) = \{H \mid H \text{ ist Grundinstanz eines Rechenergebnisses von F bei Eingabe G}\}$$

Die modelltheoretische Semantik von (F,G) ist die Menge

$$Sm(F,G) = \{H \mid H \text{ ist Grundinstanz von } (A_1 \wedge ... \wedge A_k) \text{ und folgt aus F}\}$$

Satz: Für alle Hornklauselprogramme F und Zielklauseln G: Sp(F,G) = Sm(F,G).

Def.:
Sei F ein Hornklauselprogramm. Wir definieren einen Operator O_F über Mengen von Grundatomen wie folgt:

$O_F(M) = \{A' \mid$ es gibt Klausel $K = \{A,\neg B_1,...,\neg B_n\}$ in F,
$\{A',\neg B_1',...,\neg B_n'\}$ ist Grundsubstitution von K und
$B_1',..., B_n'$ in M$\}$

Dieser Operator ist monoton und besitzt kleinsten Fixpunkt $Fp = \cup\, n \geq 0\ O_F^n(\{\})$.
Hierbei ist $O_F^n(M) = M$ falls $n = 0$, $O_F(O_F^{n-1}(M))$ sonst.
O_F liefert alle Atome, die in einem Schritt aus Regeln von F abgeleitet werden können, falls M bereits abgeleitet ist.

Fixpunktsemantik:

Sf(F,G) = {H | H Grundinstanz von $(A_1 \wedge ... \wedge A_k)$ und für alle Atome A in H gilt $A \in Fp$}.

Satz: Für alle Hornklauselprogramme F und Zielklauseln G: Sp(F,G) = Sm(F,G) = Sf(F,G).

Auswertungsstrategien:

Logik-Programme sind nichtdeterministisch: nach jedem Berechnungsschritt kann es verschiedene Nachfolgekonfigurationen geben.
Auswertungsstrategie schreibt vor, in welcher Reihenfolge die Schritte ausgeführt werden.
2 Gründe für Nichtdeterminismus:

Nichtdet. 1. Art: Auswahl der Programmklausel, die für Resolutionsschritt verwendet wird
Nichtdet. 2. Art: Auswahl des Literals der jeweiligen Zielklausel, das als nächstes abgearbeitet wird

Nichtdet. der 2. Art (Reihenfolge der Abarbeitung der Teilziele) irrelevant in folgendem Sinn:

Def.: Eine Berechnung eines Logik-Programmes heißt kanonisch, falls bei jedem Konfigurationsübergang mit dem ersten (am weitesten links stehenden) Literal der Zielklausel resolviert wird (Klauseln werden hier also nicht mehr als Mengen, sondern als Listen aufgefaßt).

Satz: Sei (G,[]) |-- ... |-- ([], sub) eine erfolgreiche Berechnung des Logik-Programms F. Dann gibt es eine erfolgreiche kanonische Berechnung von F bei Eingabe G derselben Länge und mit demselben Rechenergebnis.

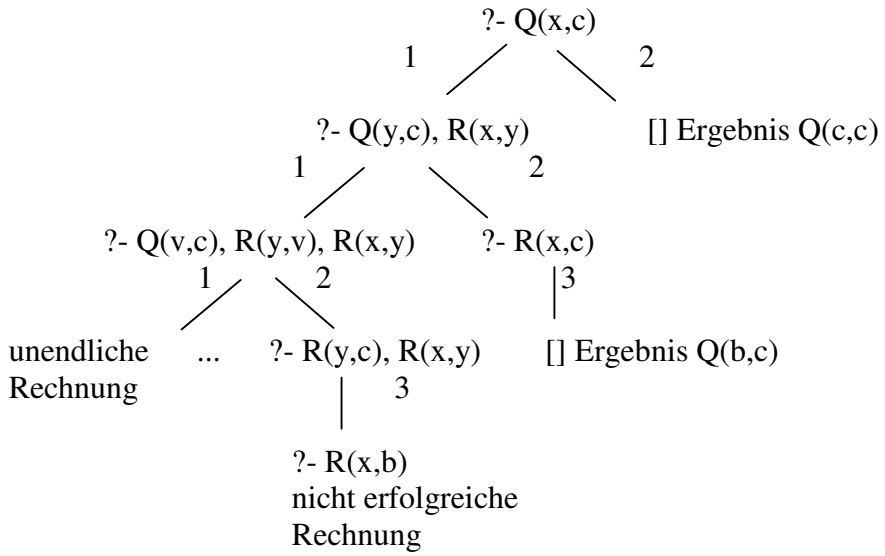
Bemerkung: Reihenfolge der Teilziele spielt natürlich eine Rolle bei nicht erfolgreichen Berechnungen: je später man merkt, dass ein Teilziel nicht abgeleitet werden kann, desto schlechter ist das für die Effizienz des Systems.

Kanonische Berechnungen werden oft als Bäume dargestellt: Wurzel (G,[]), Nachfolger jeweils alle in einem kanonischen Rechenschritt erreichbaren Nachfolgekonfigurationen. Substitutionen häufig weggelassen.

Beispiel:

1) Q(x,z) :- Q(y,z), R(x,y).
2) Q(x,x)
3) R(b,c)

Zielklausel: ?- Q(x,c)



Beispiel zeigt, dass Auswahl der im jeweiligen Resolutionsschritt verwendeten Programmklauseln kritisch ist. Berechnungsbaum muss vollständig durchsucht werden

2 wichtige Strategien:

- Breitensuche (breadth first): alle Knoten im Baum der Tiefe t werden abgearbeitet, bevor ein Knoten der Tiefe t+1 abgearbeitet wird.
- Tiefensuche (depth first): von den noch weiter resolvierbaren Knoten der größten Tiefe wird jeweils der am weitesten links stehende Knoten abgearbeitet.

Wichtig: Breitensuche ist vollständig, d.h. wenn es eine Lösung gibt, dann wird sie auch nach endlicher Zeit gefunden, aber extrem ineffizient: wenn der Baum einen durchschnittlichen Verzweigungsgrad von m hat und die Lösung in Tiefe t liegt, so sind im schlechtesten Fall $m + m^2 + ... + m^t$ Resolutionsschritte auszuführen (also exponentiell in der Lösungstiefe).

Tiefensuche ist nicht vollständig: im obigen Beipielbaum führt Tiefensuche zu keiner Lösung.
Allerdings häufig effizienter als Breitensuche, wird deshalb in den meisten PROLOG-Systemen verwendet.

Aufgabe des Programmierers ist es dann, Programme so zu schreiben, dass System nicht in Endlosschleife gerät. Ziel der Logik-Programmierung (Programmierer sagt nur, was gilt, nicht wie abgeleitet werden soll) in Prolog also nur teilweise erreicht.

Prolog-Algorithmus: Eingabe: Logik-Programm $F = (K_1, ..., K_n)$, wobei $K_i = B_i$:- $C_{i,1},...,C_{i,ni}$ und Zielklausel $G = ?- A_1, ..., A_n$.

Hauptprogramm:

```
success:= false;
auswerte(G,[]);
if success = false then write('nicht ableitbar');
```

Prozedur auswerte:

```
procedure auswerte(G,sub);
begin
if G = [] then begin write('Ergebnis:', (A1 ∧ ... ∧ An)sub; success := true end
else begin {G habe die Form ?- D1, ..., Dk}
        i:= 0;
        while (i < n) and not success do
        begin
          i:= i+1;
          if {D1,Bi) unifizierbar mit mgu s then auswerte(?- (Ci,1,..., Ci,ni,D2,...,Dk)s, sub s)
        end
      end
end
```

Bemerkungen:

- Tiefensuche wird zu Breitensuche, indem man im rekursiven Aufruf von *auswerte* den Körper der resolvierten Klausel hinten statt vorne im ersten Argument anhängt.
- Alle Ergebnisse werden ausgegeben, wenn man in while-Bedingung "and not success" weglässt.

Prolog Beispiel:

1) Vater(Peter, Hans)
2) Vater(Peter, Maria)
3) Vater(Hans, Uli)
4) Vater(Karl, Anton)
5) Mutter(Maria, Anna)
6) Großvater(X,Y) :- Vater(X,Z), Vater(Z,Y)
7) Großvater(X,Y) :- Vater(X,Z), Mutter(Z,Y)

Anfrage: :- Großvater(Peter,V)	6) [X/Peter, Y/V]
:- Vater(Peter,Z), Vater(Z,V)	1) [Z/Hans]
:- Vater(Hans,V)	3) [V/Uli]
:- []	erste Antwort: V = Uli

falls weitere Antworten gewünscht: backtracking (Rücksprung zu letzter Stelle, an der eine alternative Regelauswahl existiert):

:- Vater(Peter,Z), Vater(Z,V)	2) [Z/Maria]
:- Vater(Maria,V)	nicht erfolgreich

backtracking:

:- Großvater(Peter,V)	7) [X/Peter, Y/V]
:- Vater(Peter,Z), Mutter(Z,V)	1) [Z/Hans]
:- Mutter(Hans,V)	nicht erfolgreich

backtracking:

:- Vater(Peter,Z), Mutter(Z,V)	2) [Z/Maria]
:- Mutter(Maria,V)	5) [V/Anna]
;- []	zweite Antwort: V = Anna

keine weiteren Lösungen.

3.3 Behandlung negativer Information in Logikprogrammen

Anfragen bisher: G = :- $A_1,...,A_k$, intuitiv: "ist $A_1 \wedge ... \wedge A_n$ ableitbar?" bzw, gibt es Substitution sub für freie Variablen in G, so dass $(A_1 \wedge ... \wedge A_n)$ sub ableitbar?

Also nur positive Anfragen (werden bei Widerspruchsbeweis zu negativen Klauseln).

Negative Anfragen:

Negative Literale können aus bisher betrachteten Logikprogrammen im klassischen Sinn nie abgeleitet werden: es gibt für jedes Atom ein Modell, das das Atom wahr macht: man nehme das (Herbrand-) Modell, in dem alle Atome wahr sind => macht alle Regeln wahr.

Im Kontext von Datenbanken und Logikprogrammen geht man häufig davon aus, dass man über vollständige positive Information verfügt, d.h. man betrachtet Atome, die nicht abgeleitet werden können, als falsch.

Modelltheoretische Semantik: nicht alle Modelle betrachten, sondern nur das (Herbrand-) Modell, in dem am wenigsten Atome wahr sind.
Modelle repräsentiert durch Menge der Atome, die wahr sind. $M_1 \leq M_2$ gdw $M_1 \subseteq M_2$.
Kleinstes Modell M_m existiert für Hornklauseln.
Wir definieren: Literal "not A" folgerbar aus P gdw. A nicht in M_m.

(Aussagenlogisches) Beispiel P_1:

a :- b, c
c :- d
d

Modelle: $M_1 = \{d,c\}$, $M_2 = \{d,c,a\}$, $M_3 = \{d,c,b,a\}$
M_1 minimal, damit ist "not a" und "not b" ableitbar, da a und b in M_1 falsch sind.
Verwendung von not statt ¬ zeigt an, dass wir es hier nicht mit klassischer Negation zu tun haben.

Bemerkung: der hier zugrundeliegende Ableitbarkeitsbegriff ist nichtmonoton:
Monotonie: p ableitbar aus F => p ableitbar aus $F \cup \{q\}$ für beliebige Formel q.

Beispiel: "not a" aus P_1 ableitbar, aber nicht aus $P_1 \cup \{b\}$.

Prozedurale Semantik für negative Anfragen: SLDNF: SLD mit negation as finite failure:
Überprüfen eines negativen Zielliterals not L durch Starten eines SLD-Beweises für L.
Schlägt der Beweis in endlicher Zeit fehl, dann ist not L ableitbar, sonst nicht.

Zu beachten: da das Resolutionsverfahren nicht immer terminiert, klaffen modelltheoretische und prozedurale Semantik hier auseinander.
Negation as finite failure korrekt bzgl. der Semantik des kleinsten Modells, aber nicht vollständig.

Normale Logikprogramme:

Da ein Ableitungsmechanismus für "not" vorhanden ist, lässt sich Negation auch in den Programmen selbst verwenden (genauer: im Körper der Regeln): normale Logik-Programme:

Regeln der Form A :- B_1,...,B_n, not C_1, ..., not C_m
wobei A, B_i und C_j Atome (möglicherweise freie Variablen)

Problem: es gibt nicht immer ein kleinstes Herbrand-Modell, sondern möglicherweise mehrere minimale:

(Aussagenlogisches) Beispiel:

a :- c, not b
b :- c, not a
c :- not d

3 minimale Modelle: $M_1 = \{a,c\}$, $M_2 = \{b,c\}$, $M_3 = \{d\}$.
Nicht jedes minimale Modell intendiert: M_3 unerwünscht, da d nicht hergeleitet werden kann. (Grundprinzip: für jedes wahre Atom A sollte es mindestens eine Regel mit Kopf A geben, deren Vorbedingung auch wahr ist).

Begriff des stabilen Modells (nur definiert für Programme ohne Variablen):
Idee: wir raten ein Modell und überprüfen dann, ob man mit den sich daraus ergebenden Auswertungen der not-Literale dasselbe Modell wieder herleiten kann. Diese Konstruktion garantiert, dass nur minimale Modelle erzeugt werden, und unter diesen nur solche, in denen jedes wahre Atome eine gültige Herleitung hat.

Def. Sei M ein Modell, P ein variablenfreies Programm. Das um M reduzierte Programm, P^M, ergibt sich aus P durch

1. Streichen aller Regeln mit (not C_i) im Körper und $C_i \in M$.
2. Streichen aller not-Literale aus allen anderen Regeln.

Das reduzierte Programm enthält kein Vorkommen von not, hat also ein kleinstes Modell. M heißt stabil gdw. M das kleinste Modell von P^M ist.

obiges Beispiel:

P^{M_1}: a :- c — minimales Modell: $\{a,c\} = M_1$
c

P^{M_2}: b :- c — minimales Modell: $\{b,c\} = M_2$
c

P^{M_3}: a :- c — minimales Modell: $\{\} \neq M_3$
b :- c

Also 2 stabile Modelle, M_1 und M_2.

Prozedural: dieselbe Idee wie vorher: wann immer ein negatives goal (not L) abgearbeitet wird, wird ein Beweis für L gestartet. Wenn der endlich misslingt, ist not L wahr, sonst nicht. Programm läuft möglicherweise endlos:

Beispiel:

a :- not b
b :- not a
c :- not c, a

stabiles Modell: {b}

aber prozedurale Interpretation (SLDNF) geht in Endlosschleife: Überprüfen von b liefert Teilziel not a, Beweis für a wird gestartet, liefert Teilziel not b, Beweis für b wird gestartet, ...

Deshalb: Beweiser für normale Logikprogramme unter stabiler Semantik deutlich aufwändiger als SLDNF.