Logikprogrammierung SS 2000

Dozent: Prof. Dr. K. Indermark

Eine Vorlesungsmitschrift

Thorsten Uthke
thorsten.uthke@post.rwth-aachen.de

1. August 2000

Inhaltsverzeichnis

0 Einleitung

Programmiersprachen: imperativ - deklarativ
/ \
funktional logisch

`imperativ:` Alg. zur Lösung eines Problems, Folge von Zustandstransformationen; rechnerorientiert
Beispiel: Pascal, Modula, C, Ada, Java

`deklarativ:` Beschreibung eines Problems durch **Funktionen** (funktional) oder **Relationen** (logische PS); problemorientiert

`funktional:` Beruht auf λ-Kalkül und Termersetzung
Beispiel: Haskell, ML, Erlang (Telekommunikation)

`logisch:` Beruht auf Horn-Logik und SLD-Resolution mit Unifikation
Beispiel: Prolog, Gödel, Cunny
Anwendung: KI, deduktive DB, Expertensysteme, Rapid prototyping

Logikprogrammierung: Beschreibung des Problemwissens durch logische Formeln, welche Eigenschaften und Beziehungen von Objekten angeben.

$\Phi = \{F_1, \ldots, F_n\}$ Anfrage F
Fragen: Folgt F aus Φ? Ist F in den Modellen von Φ gültig? Ist $\Phi \cup \{\neg F\}$ unerfüllbar?
Operationale Behandlung durch **Resolution** und **Unifikation**.

Weitere Beziehungen:
Verwendung von Variablen, Konjunktion, Implikation
z.B. V ist Vater von K, *wenn* V und M verheiratet *und* M Mutter von K ist.

Formal: verheiratet$(V, M) \wedge$istMutterVon$(M, K) \rightarrow$istVaterVon(V, K)

Prolog-Regel:
```
istVaterVon(V,K) :- verheiratet(M,V), istMutterVon(M,K).
istGroßvaterVon(G,E) :- istVaterVon(G,V), istVaterVon(V,E).
```

Anfrage in Prolog:

?- weiblich(eva).	?- männlich(eva).
yes	no

Welche Enkel hat Hans?
Varaible E
```
?- istGroßvaterVon(Hans,E)
E=Petra;
E=Peter;
no   ← keine weitere Lösung
```

Es fehlt: "Otto", weitere Regel:
```
istGroßvaterVon(G,E) :- istVaterVon(G,M), istMutterVon(M,E)
```

Aussagenlogische Abstraktion um Resolution herzuleiten.
Später: Übertragen auf PL1 (insbesondere Horn-Logik)

Fakten:	$A.$	$1 \to A.$	
Regeln:	$D :- C, A.$	$C \wedge A \to D$	
Anfragen:	$? :- A, B.$	$A \wedge B \to 0$	$(\neg(A \wedge B) \vee \neg A \vee \neg B)$

1 Aussagenlogik und Resolution

1.1 Syntax und Semantik der Aussagenlogik

Definition 1.1 (AL-Syntax)
Sei $Var := \{A_i | i \in \mathbb{N}\}$ eine abbildbare Menge von Variablen (**AL-Variablen**, atomare Formeln) und $V \subseteq Var$. Dann ist die Menge $AF(V)$ der **AL-Formeln** über V induktiv definiert durch:

- $V \subseteq AF(V)$
- $F, G \in AF(V) \implies (F \wedge G), (F \vee G), \neg F, \neg G \in AF(V)$

Abh.
$(F \to G) := (\neg F \vee G)$ Implikation
$(F \leftrightarrow G) := (F \to G) \wedge (G \to F)$ Äquivalenz
$(\bigcup_{i=1}^{n} F_i) := (F_1 \vee F_2 \vee \ldots \vee F_n)$
$1 := (A_0 \vee \neg A_0)$
$0 := (A_0 \wedge \neg A_0)$

Definition 1.2 (AL-Semantik)
Sei $\mathbb{B} = \{0, 1\}$ die Menge der **Wahrheitswerte** (0: falsch, 1: wahr) und $\alpha : V \to \mathbb{B}$ eine **Belegung**. Dann läßt sich α fortsetzen zu $\overline{\alpha} : AF(V) \to \mathbb{B}$.

- $\overline{\alpha}(A) := \alpha(A) \quad \forall A \in V$
- $\overline{\alpha}(F \wedge G) = 1 \iff \overline{\alpha}(F) = 1$ und $\overline{\alpha}(G) = 1$
- $\overline{\alpha}(F \vee G) = 1 \iff \overline{\alpha}(F) = 1$ oder $\overline{\alpha}(G) = 1$
- $\overline{\alpha}(\neg F) = 1 \iff \overline{\alpha}(F) = 0$

Definition 1.3 (Modell, Gültigkeit, Erfüllbarkeit)
Sei $F \in AL(V)$ und $\alpha : V \to \mathbb{B}$. Wenn $\overline{\alpha}(F) = 1$, so schreibt man $\alpha \models F$ und sagt: "F gilt unter α" oder "α ist **Modell** von F" oder "α **erfüllt** F".
F heißt **erfüllbar** wenn es ein Modell für F gibt, andernfalls **unerfüllbar**.
$\Phi \subseteq AF(V)$ heißt **erfüllbar**, wenn es ein Modell für Φ, d.h. für alle $F \in \Phi$ gibt.
$F \in AF(V)$ heißt **allgemeingültig (Tautologie)**, wenn $\alpha \models F \forall \alpha : V \to \mathbb{B}$, in Zeichen $\models F$; $\not\models F$ falls F keine Tautologie.

Lemma 1.1 Für $F \in AF(V)$ gilt: F ist eine Tautologie $\iff$ $\neg F$ ist unerfüllbar

Anwendungsbezug: Beschreibung einer Welt durch $\Phi \subseteq AF(V)$
Frage: Was kann aus Φ gefolgert werden? (Anfragen)

Definition 1.4 (Semantischer Folgerungsbegriff)
Sei $\Phi \cup \{F\} \subseteq AF(V)$. Dann sagt man "aus Φ **folgt** F", in Zeichen $\Phi \models F$, falls für alle $\alpha : V \to \mathbb{B}$ gilt: $\alpha \models \Phi \implies \alpha \models F$ (α Modell für $\Phi \implies \alpha$ Modell für F). $Cons(\Phi) := \{F \in AF(V) | \Phi \models F\}$ heißt **Folgerungsmenge** von Φ.

Ziel: Semantischen Folgerungsbegriff operationalisieren - Resolutionskalkül
Vorbereitung: Reduktion auf Unerfüllbarkeit

Satz 1.1 Für $F_1, \ldots, F_k, F \in AF(V)$ gilt:
$\{F_1, \ldots, F_k\} \models F \iff ((\bigwedge_{i=1}^{k} F_i) \wedge \neg F)$ unerfüllbar

Beweis: $\{F_1, \ldots, F_k\} \models F$
$\iff \forall \alpha : V \to \mathbb{B} : \overline{\alpha}(F_1) = \ldots = \overline{\alpha}(F_k) = 1$ genau dann, wenn $\overline{\alpha}(F) = 1$
$\iff \forall \alpha : V \to \mathbb{B} : \overline{\alpha}((F_1 \wedge \ldots \wedge F_k) \to F) = 1$
$\iff \forall \alpha : V \to \mathbb{B} : \overline{\alpha}(\neg(F_1 \wedge \ldots \wedge F_k) \vee F) = 1$
$\iff \forall \alpha : V \to \mathbb{B} : \overline{\alpha}(\neg(\neg(F_1 \wedge \ldots \wedge F_k) \vee F)) = 0$
$\iff \forall \alpha : V \to \mathbb{B} : \overline{\alpha}(F_1 \wedge \ldots \wedge F_k \wedge \neg F) = 0$
Dies bedeutet, daß die Formel unerfüllbar ist, da für alle möglichen α 0 herauskommt.

Anwendung auf die Logikprogrammierung:
$\Phi = \{F_1, \ldots, F_k\}$ Logikprogramm mit deklarativer Semantik $Cons(\Phi)$
$F \in AF(V)$ Anfrage an Φ (Ziel,Goal), d.h. prüfen, ob $F \in Cons(\Phi)$ oder nicht.
Dazu: Unerfüllbarkeit von $((\bigwedge_{i=1}^{k} F_i) \wedge \neg F) =: G$ testen
Methode: Für $n \in \mathbb{N}$ mit $F_1, \ldots, F_k \in AF(\{A_0, \ldots, A_{n-1}\})$ gibt es 2^n Belegungen $\alpha : \{A_0, \ldots, A_{n-1}\} \to \mathbb{B}$
Exponentieller Aufwand. SAT ist NP-vollständig.

Ziel: Effiz. Verfahren für relevante Teilklassen (Hornlogik und SLD-Resolution)

Satz 1.2 (Endlichkeitssatz, Kompaktheitssatz) Für $\Phi \subseteq AF(V)$ gilt:
Φ erfüllbar $\iff$ jede endliche Teilmenge von Φ erfüllbar.

Korollar 1.1 Φ unerfüllbar $\iff$ Es gibt eine endliche unerfüllbare Teilmenge von Φ.

1.2 Normalformen

Definition 1.5 (Semantische Äquivalenz)
$F, G \in AF(V)$ heißen **äquivalent** $\iff \forall \alpha : V \to \mathbb{B} : \overline{\alpha}(F) = \overline{\alpha}(G)$, in Zeichen $F \equiv G$. Offensichtlich gilt: $F \equiv G \iff F \leftrightarrow G$ allgemeingültig.

Definition 1.6 (Konjunktive Normalform)
$F \in AF(V)$ ist in **konjunktiver Normalform (KNF)** genau dann, wenn $F = (\bigwedge_{i=1}^{n}(\bigvee_{j=1}^{m_i} L_{ij}))$ mit $L_{ij} \in V \cup \{\neg A | A \in V\}$. Dabei heißt $L \in V$ **positives Literal** und $L \in \{\neg A | A \in V\}$ **negatives Literal**.

Satz 1.3 Jedes F ist äquivalent in KNF transformierbar.

Definition 1.7 (Hornformeln)
$F \in AF(V)$ heißt **Hornformel** genau dann, wenn F in KNF ist und jedes Glied aus Disjunktionen ($\bigvee_{j=1}^{m_i} L_{ij}$) höchstens ein positives Literal enthält.

Beispiel 1.1 $F = (A \vee \neg B) \wedge (\neg C \vee \neg A \vee D) \wedge (\neg A \vee \neg B) \wedge D \wedge \neg E$
Schreibweise durch Implikationen:
$F \equiv (B \to A) \wedge (C \wedge A \to D) \wedge (A \wedge B \to 0) \wedge (1 \to D) \wedge (E \to 0)$
Ermöglicht **prozedurale Deutung** von Logikprogrammen

- Fakten: 1 positives Literal, keine negativen Literale, z.B. `D.`
- Regeln: 1 positives Literal und negative Literale, z.B. `D :- C,A.`
- Ziele: kein positives Literal, nur negative Literale, z.B. `?- A,B.`

Nicht jede AL-Formel besitzt eine äquivalente Hornformel. Die Hornlogik ist eine echte Einschränkung der AL. Die Erfahrung zeigt, daß dies kein Handicap für die Praxis ist.
Schöning: Effizienter Erfüllbarkeitstest in $\mathcal{O}(n)$

1.3 Resolution

AL-Formeln in KNF: Unerfüllbarkeit durch Resolution feststellen
Später: Resolutionskalkül auf PL1 übertragen
Klauselnotation für AL-Formeln in KNF

Definition 1.8 (Klausel)
$K \subseteq \{A_0, \neg A_0, A_1, \neg A_1, \ldots\}$ heißt **Klausel**, falls K endlich. $\square$ bezeichnet die **leere Klausel**.

Definition 1.9 (Klauselmenge einer AL-Formel in KNF)
Sei $F = (L_{11} \vee \ldots \vee L_{1m_1}) \wedge \ldots \wedge (L_{n1} \vee \ldots \vee L_{nm_n})$ eine AL-Formel in KNF. Dann heißt $\mathcal{K}(F) := \{\{L_{11}, \ldots, L_{1m_1}\}, \ldots, \{L_{n1}, \ldots, L_{nm_n}\}\}$ die **Klauselmenge** von F.

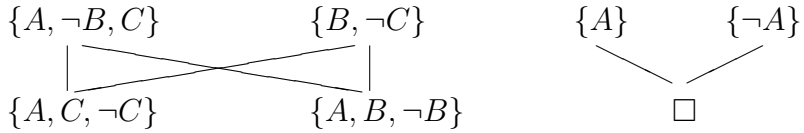
Beachte:

- $|\mathcal{K}(F)| \leq n$ und $|\{L_{i1}, \ldots, L_{im_i}\}| \leq m_i$
- Für $F, G \in AF(V)$ gilt: $\mathcal{K}(F) = \mathcal{K}(G) \implies F \equiv G$
- Semantik von $\mathcal{K}(F)$:= Semantik von F

Vereinbarung: $\overline{\alpha}(\square) = 0$ für alle Belegungen α. ($\square$ als Klausel einer unerfüllbaren Formel.)

Definition 1.10 (Resolvent zweier Klauseln)
Seien K_1, K_2 und R Klauseln. Dann heißt R **Resolvent** von K_1 und K_2, in Zeichen $\begin{array}{ccc} K_1 & & K_2 \\ & \searrow R \swarrow & \end{array}$ falls gilt: Es gibt ein Literal L mit $L \in K_1$ und $\overline{L} \in K_2$ und $R = (K_1 \setminus \{L\}) \cup (K_2 \setminus \{\overline{L}\})$.
Dabei ist $\overline{L} = \left\{ \begin{array}{ll} \neg A & \text{falls } L = A \\ A & \text{falls } L = \neg A \end{array} \right\}$ das **inverse Literal**.
Sprechweise: R wird aus K_1 und K_2 (nach L) **resolviert**.

Beispiel 1.2



Definition 1.11 (Resolutionskalkül)
Sei $\mathcal{K}$ eine Klauselmenge. Dann ist $Res(\mathcal{K}) := \mathcal{K} \cup \{R | R \text{ ist Resolvent zweier Klauseln in } \mathcal{K}\}$ die **Resolventenerweiterung** von $\mathcal{K}$.
Ferner: $Res^0(\mathcal{K}) := \mathcal{K}$ und $Res^{n+1}(\mathcal{K}) := Res(Res^n(\mathcal{K}))$ für $n \geq 0$
Dann ist $Res^*(\mathcal{K}) := \bigcup_{k \in \mathbb{N}} Res^k(\mathcal{K})$ die **Resolventenhülle** von $\mathcal{K}$.

Bem.: Enthält $\mathcal{K}$ n AL-Variablen, so enthält $Res^*(\mathcal{K})$ höchstens 2^{2n} Klauseln.
Folgerung: Zu $\mathcal{K}$ gibt es ein $k \in \mathbb{N}$, so daß für alle $i \in \mathbb{N}$ $Res^k(\mathcal{K}) = Res^{k+i}(\mathcal{K}) = Res^*(\mathcal{K})$ gilt.

Lemma 1.2 (Semantische Invarianz der Resolution) Sei $\mathcal{K}$ eine Klauselmenge. Dann gilt für 2 Klauseln $K_1, K_2 \in \mathcal{K}$: $\begin{array}{ccc} K_1 & & K_2 \\ & \searrow R \swarrow & \end{array} \implies$ $\mathcal{K}$ ist (semantisch) äquivalent zu $\mathcal{K} \cup \{R\}$.

Beweis: $\{A_1, \ldots, A_n\}$ enthalte alle AL-Variablen von $\mathcal{K}$.
Sei $\alpha : \{A_1, \ldots, A_n\} \to \mathbb{B}$. Zu zeigen: $\overline{\alpha}(\mathcal{K}) = \overline{\alpha}(\mathcal{K} \cup \{R\})$. Es genügt der Nachweis von $\overline{\alpha}(\mathcal{K}) = 1 \implies \overline{\alpha}(R) = 1$.
Sei $R = (K_1 \setminus \{L\}) \cup (K_2 \setminus \{\overline{L}\})$ und $\overline{\alpha}(K_1) = \overline{\alpha}(K_2) = 1$.

Fall $\overline{\alpha}(L) = 1 : \overline{\alpha}(\overline{L}) = 0 \implies \overline{\alpha}(K_2 \setminus \{\overline{L}\}) = 1 \implies \overline{\alpha}(R) = 1$
Fall $\overline{\alpha}(L) = 0 : \implies \overline{\alpha}(K_1 \setminus \{L\}) = 1 \implies \overline{\alpha}(R) = 1$

Spezialfälle der AL-Resolution:

1. $\{A, \neg A\} \subset K \implies \begin{array}{ccc} K & & K \\ & \searrow K \swarrow & \end{array}$
2. $\begin{array}{ccc} \{A\} & & \{\neg A\} \\ & \searrow \square \swarrow & \end{array}$ nicht erfüllbar ($\square$ macht die Nichterfüllbarkeit sichtbar)

Satz 1.4 *(Korrektheit und Vollständigkeit der Resolutionskalküls für die Unerfüllbarkeit einer AL-Formel in KNF)*
Für eine Klauselmenge $\mathcal{K}$ gilt: $\mathcal{K}$ unerfüllbar $\iff \square \in Res^*(\mathcal{K})$

Beweis: *Korrektheit:* "$\Longleftarrow$"
$\square \in Res^*(\mathcal{K}) = Res^k(\mathcal{K})$ für geignetes $k \in \mathbb{N} \quad \Longrightarrow \quad Res^k(\mathcal{K})$ nicht erfüllbar.
Nach Lemma 1.2: $\mathcal{K} \equiv Res(\mathcal{K}) \equiv \ldots \equiv Res^k(\mathcal{K}) \quad \Longrightarrow \quad \mathcal{K}$ nicht erfüllbar.

Vollständigkeit:
Sei $\mathcal{K}$ unerfüllbar.
Induktion über $n \in \mathbb{N}$ mit: A Variable in $\mathcal{K} \quad \Longleftrightarrow \quad A \in \{A_0, \ldots, A_{n-1}\}$
$n = 0 : \mathcal{K} = \{\square\}, \square \in Res^*(\mathcal{K})$
$n \to n+1$: Sei $\mathcal{K}$ unerfüllbar mit Variablen aus $\{A_0, \ldots, A_{n-1}\}$ (Vor.)
$\mathcal{K}_0$ entstehe aus $\mathcal{K}$ durch Weglassen von A_n aus allen Klauseln und Weglassen aller Klauseln mit $\neg A_n$
$\mathcal{K}_1$ entstehe aus $\mathcal{K}$ durch Weglassen von $\neg A_n$ aus allen Klauseln und Weglassen aller Klauseln mit A_n

$\mathcal{K}_0$ ist unerfüllbar:
Wäre $\mathcal{K}_0$ erfüllbar durch $\alpha : \{A_0, \ldots, A_{n-1}\} \to \mathbb{B}$ mit $\overline{\alpha}(\mathcal{K}_0) = 1$ so gälte für die Erweiterung von α zu α' mit $\alpha'(A_n) = 0$, daß $\overline{\alpha}'(\mathcal{K}) = 1$
$\Longrightarrow$ Wiederspruch

$\mathcal{K}_1$ ist unerfüllbar: Beweis analog

Nach Induktionsvoraussetzung: $\square \in Res^*(\mathcal{K}_0)$ und $\square \in Res^*(\mathcal{K}_1)$
Dann existiert eine Herleitung von $\square$ aus $\mathcal{K}_0$, d.h. eine Klauselfolge $K_1, \ldots, K_m$
mit $K_m = \square$ und für alle $i \in \{1, \ldots, m\}$ gilt: $K_i \in \mathcal{K}_0$ oder $\begin{array}{ccc} K_j & & K_k \\ & \searrow \quad \swarrow & \\ & K_i & \end{array} \quad j, k < i$

Wiedereinfügen der A_n ergibt:	Wiedereinfügen der $\neg A_n$ ergibt:
$\square \in Res^*(\mathcal{K})$ oder $Res^*(\mathcal{K}) \ni \{A_n\}$	$\{\neg A_n\} \in Res^*(\mathcal{K})$ oder $\square \in Res^*(\mathcal{K})$

$\square$

Folgerung: Für eine AL-Formel F in KNF ist durch Resolution in endlich vielen Schritten entscheidbar, ob F erfüllbar ist, oder nicht:
Berechne $\mathcal{K}(F)$, $Res(\mathcal{K}(F))$, $Res^2(\mathcal{K}(F))$, ...
Bis $\square \in Res^n(\mathcal{K}(F))$ ($\Longrightarrow$ F unerfüllbar) oder
$Res^n(\mathcal{K}(F)) = Res^{n+1}(\mathcal{K}(F))$ mit $\square \notin Res^{n+1}(\mathcal{K}(F))$ ($\Longrightarrow$ F erfüllbar)

Für die Unerfüllbarkeit von F genügt eine Herleitung von $\square$.
Daher am Besten nur die notwendigen Resolventen bilden.
Ueb: Unerfüllbarkeit von Hornformeln mit linearem Aufwand.

1.4 Lineare Resolution, SLD-Resolution

Problem: Viele Resol. Möglichkeiten - komb. Explosion
Ziel: Suchraum verkleinern durch Beschränkung auf spezielle Resolutions-Herleitungen (Vollständigkeit)

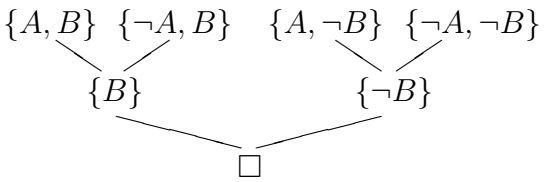
Ergebnis:

- Lineare Resolution (ist vollständig für beliebige Klauselmengen)
- SLD-Resolution (ist vollständig für Horn-Klauselmengen)

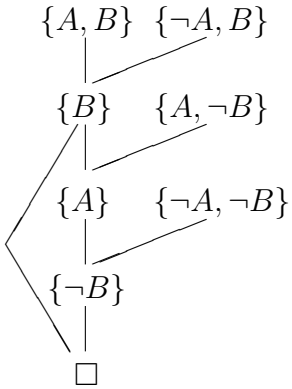
Definition 1.12 (Lineare Resolution)
Sei $\mathcal{K}$ eine Klauselmenge und $K \in \mathcal{K}$. Dann sagt man "$\square$ ist aus K in $\mathcal{K}$ **linear resolvierbar**", falls eine Klauselfolge $K_0, \ldots, K_n$ existiert, so daß $K_0 = K$, $K_n = \square$ und für $i = 1, \ldots, n$ gilt: K_i ist Resolvent von K_{i-1} und von $K' \in \{K_0, \ldots, K_{i-2}\} \cup \mathcal{K}$.

Beachte: Falls $\square \in \mathcal{K}$, so ist $\square$ aus $\square$ linear resolvierbar.

Beispiel 1.3



Linerare Resolutionsherleitung:



Satz 1.5 (Vollständigkeit der linearen Resolution) Sei $\mathcal{K}$ eine unerfüllbare Klauselmenge. Dann existiert eine Klausel $K \in \mathcal{K}$, so daß $\square$ aus K in $\mathcal{K}$ linear resolvierbar ist.

Beweis: Sei $\mathcal{K}$ unerfüllbar und $\mathcal{K}_{min}$ eine minimale unerfüllbare Teilmenge von $\mathcal{K}$. [1] *Behauptung:* Für jedes $K \in \mathcal{K}_{min}$ gilt: $\square$ aus K in $\mathcal{K}$ linear resolvierbar.
Beweis: Induktion über $n = \#$(AL-Variablen in $\mathcal{K}_{min}$). Sei $K \in \mathcal{K}_{min}$.
$n = 0$: $\mathcal{K}_{min} = \{\square\} \implies K = \square$ und $\square$ aus $\square$ linear resolvierbar.
$n \to n+1$: $\mathcal{K}_{min}$ hat $n+1$ verschiedene AL-Variablen
Fall $|K| = 0$: $K = \square$ ✓

[1] d.h. entweder $\mathcal{K}_{min} = \{\square\}$ oder für jedes $K \in \mathcal{K}_{min}$ ist $\mathcal{K}_{min} \setminus \{K\}$ erfüllbar

Fall $|K| = 1$: $K = \{L\}$ und $L \in Var$ oder $\overline{L} \in Var$.

Entferne aus $\mathcal{K}_{min}$ alle Vorkommen von $\overline{L}$ und alle Klauseln, die L enthalten (insbesondere K). Das Ergebnis $\mathcal{K}_{min-\overline{L}}$ ist auch unerfüllbar (siehe Beweis von Satz 1.4), enthält aber nur noch n verschiedene AL-Variablen.
Sei $\mathcal{K}'_{min}$ eine minimale unerfüllbare Teilmenge von $\mathcal{K}_{min-\overline{L}}$. $\mathcal{K}'_{min}$ enthält ein K' mit $K' \cup \overline{L} \in \mathcal{K}_{min}$, denn andernfalls wäre $\mathcal{K}'_{min} \subset \mathcal{K}_{min}$ erfüllbar.
Nach Induktionsvoraussetzung ist $\square$ aus K' in $\mathcal{K}_{min-\overline{L}}$ linear resolvierbar.
Bild siehe Skript vom 20.4.2000
Wiedereinfügen von $\overline{L}$ in die Klauseln liefert eine Herleitung in $\mathcal{K}$, aber eventuell bleibt $\{\overline{L}\}$ übrig. Dann letzter Resolutionsschritt mit $K = \{L\}$ zu $\square$.

Fall $|K| > 1$: Wähle $L \in K$ und setze $K' = K \setminus \{L\}$

Entferne aus $\mathcal{K}_{min}$ alle Vorkommen von L und alle Klauseln, die $\overline{L}$ enthalten. Das Ergebnis $\mathcal{K}_{min-L}$ ist auch unerfüllbar, enthält aber nur noch n verschiedene AL-Variablen. Sei $\mathcal{K}'_{min}$ eine minimale unerfüllbare Teilmenge von $\mathcal{K}_{min-\overline{L}}$.
(*) $K' \in \mathcal{K}'_{min}$
Beweis: Sei $\alpha : Var \to \mathbb{B}$ mit $\overline{\alpha}(\mathcal{K}_{min} \setminus \{K\}) = 1$, dann $\overline{\alpha}(K) = 0$, weil $\overline{\alpha}(\mathcal{K}_{min}) = 0$ wegen der Unerfüllbarkeit. $L \in K \implies \overline{\alpha}(L) = 0 \implies \overline{\alpha}(\mathcal{K}_{min-L} \setminus \{K'\}) = 1$ (entspricht $\overline{\alpha}(\mathcal{K}_{min} \setminus \{K\}) = 1$, ohne das L, wobei hier $\overline{\alpha}(L) = 0$ ist, d.h. sich in den Disjunktionen nicht bemerkbar macht)
Wäre $K' \notin \mathcal{K}_{min}$, so wäre $\mathcal{K}'_{min} \subseteq \mathcal{K}_{min-L} \setminus \{K'\}$, also erfüllbar. Wiederspruch!
Nach Induktionsvoraussetzung ist $\square$ aus K' in $\mathcal{K}_{min-\overline{L}}$ linear resolvierbar.
Durch Wiedereinfügen von L in die Klauseln von $\mathcal{K}_{min-L}$ folgt:
$\{L\}$ ist aus K in $\mathcal{K}$ linear resolvierbar.
(**) $\square$ ist aus $\{L\}$ in $\mathcal{K}$ linear resolvierbar.
Bild siehe Skript vom 20.4.2000

Bemerkung: SLD steht für "linear resolution with selection function for definite clauses".
Hier eigentlich: LUSH, "linear resolution with unrestricted selection for horn clauses".

Satz 1.6 (Vollständigkeit der SLD-Resolution für Hornklauseln) Sei $\mathcal{K}$ eine unerfüllbare Hornklauselmenge. Dann existiert eine negative Klausel $N \in \mathcal{K}$, so daß $\square$ aus N SLD-resolvierbar ist.

Beweis: Sei $\mathcal{K}_{min}$ eine minimale unerfüllbare Teilmenge von $\mathcal{K}$. Dann existiert eine negative Klausel $N \in \mathcal{K}_{min}$. Nach Vollständigkeitsbeweis der linearen Resolution ist $\square$ aus N in $\mathcal{K}$ linear resolvierbar, und da $\mathcal{K}$ eine Hornklauselmenge ist, auch SLD-resolvierbar.

2 Prädikatenlogik, Resolution und Unifikation

Verfeinerung der AL: statt atomarer Aussagen $A_0, A_1, \ldots$ nun Aussagen über Strukturen
Struktur:

- Menge von Objekten (Universum)
- Beziehungen zwischen Objekten
  - funktional (deterministisch)
  - relational (nicht-deterministisch)

2.1 Syntax der PL1

PL1 als Sprache für Aussagen über Strukturen neben AL, Individuenvariablen, Funktions- und Prädikatssymbole und Quantoren.

Definition 2.1 (Signatur)
Seien Ω und Π Mengen mit $\rho : \Omega \cup \Pi \to \mathbb{N}$. Dann heißt $\Sigma = \langle \Omega, \Pi, \rho \rangle$ **Signatur** mit der Menge Ω von **Funktions- bzw. Operationssymbolen**, der Menge Π von **Prädikats- bzw. Relationssymbolen** und der **Stelligkeit** ρ.

Bezeichnungen: $\Omega^{(n)} := \{f \in \Omega | \rho(f) = n\}$ und $K := \Omega^{(0)}$ Konstantensymbole
$\Pi^{(n)} := \{P \in \Pi | \rho(P) = n\}$

Definition 2.2 (Σ-Term)
Sei $X \subseteq \{x_0, x_1, \ldots\}$ eine Menge von (Individuen-)Variablen. Dann ist die Menge $T_\Sigma(X)$ der **Σ-Terme** über X, kurz (Σ, X)-Terme, induktiv definiert durch

- $X \cup K \subseteq T_\Sigma(X)$
- $f \in \Omega^{(n)}$, $n > 0$, $t_1, \ldots, t_n \in T_\Sigma(X) \implies f(t_1, \ldots, t_n) \in T_\Sigma(X)$

Spezialfall: Für $X = \emptyset$ ist $T_\Sigma := T_\Sigma(\emptyset)$ die Menge der **Σ-Grundterme**.

Definition 2.3 (Σ-Formeln der Prädikatenlogik 1. Stufe)
Die Menge $PF_\Sigma(X)$ der **PL1-Formeln** über (Σ, X) ist definiert durch

- $P \in \Pi^{(n)}$, $t_1, \ldots, t_n \in T_\Sigma(X) \implies P(t_1, \ldots, t_n) \in PF_\Sigma(X)$
- $F, G \in PF_\Sigma(X) \implies \neg F$, $(F \wedge G)$, $(F \vee G) \in PF_\Sigma(X)$
- $x \in X$, $F \in PF_\Sigma(X) \implies \exists x F$, $\forall x F \in PF_\Sigma(X)$

Bemerkungen: $P(t_1, \ldots, t_n)$ ist **atomare Formel**, **Primformel**
Ein Vorkommen von x in F heißt **gebunden**, falls x in einer Teilformel der Form $\exists x G$ oder $\forall x G$ auftritt, sonst **frei**.

2.2 Semantik der PL1

PL1-Formeln über Σ-Strukturen mit Belegung freier Variablen interpretieren.

Definition 2.4 (Operationen und Relationen über einer Menge A)
A nicht-leere Mnege, $n \in \mathbb{N}$
$Ops^{(n)}(A) := \{f | f : A^n \to A\}$ **n-stellige Operationen** auf A
$Rel^{(n)}(A) := \{r | r \subseteq A^n\}$ **n-stellige Relationen** auf A
$Ops(A) := \bigcup_{n\in\mathbb{N}} Ops^{(n)}(A)$
$Rel(A) := \bigcup_{n\in\mathbb{N}} Rel^{(n)}(A)$

Schreibweise: $r(a_1, \ldots, a_n)$ statt $(a_1, \ldots, a_n) \in r$

Definition 2.5 (Herbrand-Struktur)
$\mathcal{T}_\Sigma := \langle T_\Sigma; \varphi_H \rangle$ heißt **Herbrand-Struktur**, falls
$\varphi_H(f)(t_1, \ldots, t_n) := f(t_1, \ldots, t_n)$ für alle $n \in \mathbb{N}$, $f \in \Omega^{(n)}$ und $t_1, \ldots, t_n \in T_\Sigma(X)$.

Beachte: Freie Interpretation der $f \in \Omega$, nicht möglich für $P \in \Pi$, daher viele Herbrand-Strukturen über Σ.

Zur Interpretation von PL1-Formeln über (Σ, X) ist neben einer Struktur $\mathfrak{A} = \langle A; \varphi \rangle$ noch eine Variablenbelegung nötig: $\beta : X \to A$. Läßt sich fortsetzen zu $\overline{\beta} : T_\Sigma(X) \to A$ mit $\overline{\beta}(X) = \beta(X)$ und $\overline{\beta}(f(t_1, \ldots, t_n)) := f_{\mathfrak{A}}(\overline{\beta}(t_1), \ldots, \overline{\beta}(t_n))$.

Definition 2.6 (Modell, Gültigkeit, Erfüllbarkeit)
Sei $F \in PF_\Sigma(X)$, $\mathfrak{A}\langle A; \varphi \rangle$ eine Σ-Struktur und $\beta : X \to A$ eine Variablenbelegung. Wir definieren induktiv über $PF_\Sigma(X)$ die Semantik von F bezüglich $\langle \mathfrak{A}, \beta \rangle$ als die Modellbeziehung $\langle \mathfrak{A}, \beta \rangle \models F$, mit

- $\langle \mathfrak{A}, \beta \rangle \models P(t_1, \ldots, t_n) \iff P_{\mathfrak{A}}(\overline{\beta}(t_1), \ldots, \overline{\beta}(t_n))$
- $\langle \mathfrak{A}, \beta \rangle \models (F \wedge G) \iff \langle \mathfrak{A}, \beta \rangle \models F$ und $\langle \mathfrak{A}, \beta \rangle \models G$
- $\langle \mathfrak{A}, \beta \rangle \models (F \vee G) \iff \langle \mathfrak{A}, \beta \rangle \models F$ oder $\langle \mathfrak{A}, \beta \rangle \models G$
- $\langle \mathfrak{A}, \beta \rangle \models \neg F \iff \langle \mathfrak{A}, \beta \rangle \not\models F$
- $\langle \mathfrak{A}, \beta \rangle \models \exists x F \iff$ es gibt ein $a \in A$ mit $\langle \mathfrak{A}, \beta[x/a] \rangle \models F$
- $\langle \mathfrak{A}, \beta \rangle \models \forall x F \iff$ für alle $a \in A$ gilt $\langle \mathfrak{A}, \beta[x/a] \rangle \models F$

Sprechweise: Für $\langle \mathfrak{A}, \beta \rangle \models F$ sagt man "$\langle \mathfrak{A}, \beta \rangle$ ist Modell für F" oder "$\langle \mathfrak{A}, \beta \rangle$ erfüllt F" oder "F gilt in $\langle \mathfrak{A}, \beta \rangle$".

F heißt **erfüllbar**, falls ein Modell $\langle \mathfrak{A}, \beta \rangle$ für F existiert, andernfalls **unerfüllbar**. Eine Formelmenge Φ ist erfüllbar, wenn es ein Modell gibt, das alle $F \in \Phi$ erfüllt. F heißt **allgemeingültig**, wenn jedes Modell $\langle \mathfrak{A}, \beta \rangle$ F erfüllt.

Definition 2.7 (Semantischer Folgerungsbegriff)
Für $\Phi \cup \{F\} \subseteq PF_\Sigma(X)$ sagt man: F folgt aus Φ, in Zeichen: $\Phi \models F$, falls für alle Σ-Strukturen $\mathfrak{A}$ und Belegungen $\beta : X \to A$ gilt: $\langle \mathfrak{A}, \beta \rangle \models \Phi \implies \langle \mathfrak{A}, \beta \rangle \models F$.
$Cons(\Phi) := \{F \in PF_\Sigma(X) | \Phi \models F\}$ heißt **Folgerungsmenge** von Φ.

Anwendung auf die Logikprogrammierung:
$\Phi = \{F_1, \ldots, F_k\}$ Logikprogramm mit $Cons(\Phi)$ als Semantik.
F sei Anfrage an Φ: gilt $\Phi \models F$?

Satz 2.1 Für $\{F, F_1, \ldots, F_k\} \subseteq PF_\Sigma(X)$ gilt:
$\{F_1, \ldots, F_k\} \models F \iff (\bigwedge_{i=1}^{k} F_i \wedge \neg F)$

Problem: Im Unterschied zur AL unendlich viele Möglichkeiten $\langle \Sigma, X \rangle$ durch $\langle \mathfrak{A}, \beta \rangle$ zu interpretieren. Daher: Unerfüllbarkeit wird unentscheidbar. Aber: Unerfüllbarkeit ist noch semi-entscheidbar (aufzählbar).

Idee:

1) PL-Formeln in Skolem-NF (SNF) bringen
2) $\langle \Sigma, X \rangle$-Interpretationen auf Herbrand-Interpretationen beschränken
3) Resolution um Unifikation erweitern

2.3 Skolem-Normalform und Herbrand-Strukturen

Definition 2.8 (Skolem-Formel)
$F \in PF_\Sigma(X)$ heißt **Skolem-Formel** (Formel in Skolem-NF), in Zeichen: $F \in SF_\Sigma(X)$, wenn F keine freien Variablen enthält und von der Form $F = \forall y_1 \forall y_2 \ldots \forall y_n \widetilde{F}$ mit n Variablen $y_1, \ldots, y_n \in X$ und quantorenfreiem $\widetilde{F} \in PF_\Sigma(X)$ ist.

Satz 2.2 Jedes $F \in PF_\Sigma(X)$ ist transformierbar in eine erfüllbarkeitsäquivalente[2] Formel $F' \in SF_{\Sigma'}(X)$, wenn Σ' eine geeignete Erweiterung von Σ um Funktionssymbole ist.

Beweisidee:

1) Umwandlung in Pränex-NF: Herausziehen innerer Quantoren
2) Skolemisierung: Elemination von $\exists$-Quantoren durch neue Funktionssymbole (erfüllbarkeitsäquivalent)
3) Ersetzen freier Variablen durch neue Konstanten-Symbole

Problem: Unendlich viele Σ-Strukturen (im Gegensatz zur AL)
Vereinfachung: Herbrand-Strukturen
Wichtiges Hilfsmittel: Substitution

[2] F erfüllbar $\iff$ F' erfüllbar

Definition 2.9 (Substitution)
Eine Variablenbelegung durch Terme $s : X \to T_\Sigma(X)$ läßt sich fortsetzen zu einer **(Term-)Substitution** $\widehat{s} : PF_\Sigma(X) - \to PF_\Sigma(X)$: ersetze jedes Vorkommen eines $x \in X$ durch $s(x)$. Wird dabei eine freie Variable von $s(x)$ gebunden, so ist $\widehat{s}(F)$ nicht definiert.
Beachte: Solche Variablenkonflikte sind durch Umbenennung gebundener Variablen vermeidbar.

Spezialfall:
Seien $y_1, \ldots, y_n \in X$ paarweise verschieden und $t_1, \ldots, t_n \in T_\Sigma(X)$. Wenn $s : X \to T_\Sigma(X)$ so, daß $s(y_i) = t_i$ und $s(x) = x$ für $x \notin \{y_1, \ldots, y_n\}$, dann schreibt man $F[y_1/t_1, \ldots, y_n/t_n]$ statt $\widehat{s}(F)$.

Lemma 2.1 (Substitutionslemma) Syntaktische und semantische Substitution sind in folgendem Sinn vertauschbar:
Ist $F[y_1/t_1, \ldots, y_n/t_n]$ definiert, so gilt: $\langle \mathfrak{A}, \beta \rangle \models F[y_1/t_1, \ldots, y_n/t_n] \iff \langle \mathfrak{A}, \beta[y_1/\overline{\beta}(t_1), \ldots, y_n/\overline{\beta}(t_n)] \rangle \models F$.

Beweis: Durch Induktion über den Formelaufbau (Übung 3)

Definition 2.10 (Die Herbrand-Struktur $\mathcal{T}_\mathfrak{A}$ einer Struktur $\mathfrak{A}$)
Sei $\Sigma = \langle \Omega, \Pi, \rho \rangle$ Signatur mit $K = \Omega^{(0)} \neq \emptyset$ und $\mathfrak{A} = \langle A; \varphi \rangle$ eine Σ-Struktur. Dann bestimmt $\mathfrak{A}$ eine **Herbrand-Struktur** $\mathcal{T}_\mathfrak{A} := \langle T_\Sigma; \varphi_H \rangle$ durch $\varphi_H(P)(t_1, \ldots, t_n) \iff \varphi(P)(t_{1_\mathfrak{A}}, \ldots, t_{n_\mathfrak{A}})$ für alle $n \in \mathbb{N}$, $P \in \Pi^{(n)}$, $t_1, \ldots, t_n \in T_\Sigma(X)$. ($t_{i_\mathfrak{A}}$ entspricht $\overline{\beta}_\mathfrak{A}(t_i)$. Da hier aber keine freien Variablen auftreten, wird t_i direkt in der Struktur $\mathfrak{A}$ interpretiert.)

Satz 2.3 Für eine Skolemformel $F \in SF_\Sigma(X)$ gilt: $\mathfrak{A} \models F \implies \mathcal{T}_\mathfrak{A} \models F$

Beweis: durch Induktion über $n \in \mathbb{N}$ in $F = \forall y_1 \ldots \forall y_n \widetilde{F} \in SF_\Sigma(X)$

$n = 0 : F = \widetilde{F}$ ohne Quantoren
$\mathfrak{A} \models F \implies \mathfrak{A} \models \widetilde{F} \implies \mathfrak{T}_\mathfrak{A} \models \widetilde{F}$ (wegen Def. 2.10)

$n \to n+1 : F = \forall y_1 \ldots \forall y_{n+1} \widetilde{F}$
$F' = \forall y_1 \ldots \forall y_n \widetilde{F}$ kann y_{n+1} frei enthalten
$\mathfrak{A} \models F \implies$ Für alle $a \in A$ gilt: $\langle \mathfrak{A}, \beta[y_{n+1}/a] \rangle \models F'$
$\implies$ Für alle $t \in T_\Sigma$ gilt: $\langle \mathfrak{A}, \beta[y_{n+1}/t_\mathfrak{A}] \rangle \models F'$
$\implies$ Für alle $t \in T_\Sigma$ gilt: $\mathfrak{A} \models F'[y_{n+1}/t]$ (nach Subst.-Lemma)
$\implies$ Für alle $t \in T_\Sigma$ gilt: $\mathcal{T}_\mathfrak{A} \models F'[y_{n+1}/t]$ (nach Induktionsvoraussetzung)
$\implies$ Für alle $t \in T_\Sigma$ gilt: $\langle \mathcal{T}_\mathfrak{A}, \beta[y_{n+1}/t] \rangle \models F'$
$\implies \mathcal{T}_\mathfrak{A} \models F$

Korollar 2.1 Für $F \in SF_\Sigma(X)$ gilt: F erfüllbar $\iff$ F besitzt Herbrand-Modell

Die Erfüllbarkeit einer Skolemformel läßt sich weiter reduzieren auf die AL-Erfüllbarkeit unendlich vieler Formeln.
Idee: Expansion der Skolem-Formel durch Grundterm-Substitution.

Definition 2.11 (Herbrand-Expansion einer Skolem-Formel)
Für $F = \forall y_1 \ldots \forall y_n \widetilde{F} \in SF_\Sigma(X)$ heißt $E(F) := \{\widetilde{F}[y_1/t_1, \ldots, y_n/t_n] | t_i \in T_\Sigma\}$ die **Herbrand-Expansion** von F.

Satz 2.4 (Reduktion der Erfüllbarkeit von Skolem- auf AL-Formeln)
Für eine Skolem-Formel F gilt: F erfüllbar $\iff$ $E(F)$ erfüllbar.

Beweis: Sei $F = \forall y_1 \ldots \forall y_n \widetilde{F} \in SF_\Sigma(X)$.
F erfüllbar $\iff$ es gibt eine Herbrand-Struktur $\mathcal{T} = \langle T_\Sigma, \varphi \rangle$ mit $\mathcal{T} \models F$
$\iff$ es gibt $\mathcal{T}$, so daß $\forall t_1, \ldots, t_n \in T_\Sigma$ und beliebiges β gilt:
$\langle \mathcal{T}, \beta[y_1/t_1, \ldots, y_n/t_n] \rangle \models \widetilde{F}$
$\iff$ es gibt $\mathcal{T}$, so daß $\forall t_1, \ldots, t_n \in T_\Sigma$ und $\mathcal{T} \models \widetilde{F}[y_1/t_1, \ldots, y_n/t_n]$
$\iff$ es gibt $\mathcal{T}$, so daß $\mathcal{T} \models E(F)$ $\iff$ $E(F)$ erfüllbar

Beachte: $G \in E(F)$ enthält keine Variablen mehr und hat fast die Gestalt einer AL-Formel: $P(t_1, \ldots, t_n)$ anstelle A_i.
Bemerkung: Ersetzt man in $E(F)$ die Primformeln $P(t_1, \ldots, t_n)$ durch AL-Variablen, so gilt für das Ergebnis $E(F)_{AL}$: $E(F)$ PL-erfüllbar $\iff$ $E(F)_{AL}$ AL-erfüllbar

Denn: Es gibt ein Herbrand-Modell für $E(F)$ $\iff$ Es gibt eine erfüllende Belegung von $E(F)_{AL}$, weil ein Herbrand-Modell die Werte der Prädikate auf allen Grundtermen festlegt und so eine entsprechende Belegung der AL-Variablen bestimmt ist (und umgekehrt ebenso).

Aus diesem Satz und dem Endlichkeitssatz der AL ergibt sich ein Verfahren zur Semi-Entscheidbarkeit der Unerfüllbarkeit einer PL1-Formel.

Algorithmus von Gilmore:
Problem: Ist $F \in PF_\Sigma(X)$ unerfüllbar?
Verfahren:

1) Transformiere F in erfüllbarkeits-äquivalentes $F' \in SF_\Sigma(X)$
2) Wähle Aufzählung $\{F_1, F_2, \ldots\} = E(F')_{AL}$ der Herbrand-Expansion von F' und ersetze die Primformeln durch AL-Variablen
3) Prüfe, ob $F_1, F_1 \wedge F_2, F_1 \wedge F_2 \wedge F_3, \ldots$ AL-erfüllbar
4) Abbruch, falls $F_1 \wedge \ldots \wedge F_n$ unerfüllbar

Korrektheit: F unerfüllbar $\iff$ $E(F')_{AL}$ AL-unerfüllbar
$\iff$ es gibt eine endliche Teilmenge von $E(F')_{AL}$, die unerfüllbar ist
$\iff$ es gibt ein $n \in \mathbb{N}$, so daß $F_1 \wedge \ldots \wedge F_n$ unerfüllbar

2.4 Resolution und Unifikation

Ziel: Effizienter Erfüllbarkeitstest

1) Übergang zu Klauselmengen, Resolution
   Sei $F = \forall y_1 \ldots \forall y_n \widetilde{F} \in SF_\Sigma(X)$ und $\widetilde{F}$ in KNF, d.h. $\widetilde{F} = (L_{11} \vee \ldots \vee L_{1n_1}) \wedge \ldots \wedge (L_{m1} \vee \ldots \vee L_{mn_m})$ mit $L_{ij} = P(t_1, \ldots, t_l) \in PF_\Sigma(\{y_1, \neg y_1, \ldots, y_k, \neg y_k\})$ und $\mathcal{K}(\widetilde{F}) = \{\{L_{11}, \ldots, L_{1n_1}\}, \ldots, \{L_{m1}, \ldots, L_{mn_m}\}\}$.
   Entsprechend ist jede Formel F_i der Herbrand-Erweiterung $E(F)$, also jede **Grundinstanz** $F_i = \widetilde{F}s$ mit einer **Grundsubstitution** $s : \{y_1, \ldots, y_k\} \to T_\Sigma$ in KNF. Die Literale von $\mathcal{K}(\widetilde{F}s)$ sind nun varaiblenfrei. Damit wird die AL-Resolution anwendbar.

Lemma 2.2 Sei $F = \forall y_1 \ldots \forall y_n \widetilde{F} \in SF_\Sigma(X)$ und $\widetilde{F}$ in KNF. Dann gilt: F unerfüllbar $\iff$ es gibt ein $n \in \mathbb{N}$, Klauseln $K_1, \ldots, K_n \in \mathcal{K}(\widetilde{F})$ und Grundsubstitutionen $s_1, \ldots, s_n : \{y_1, \ldots, y_k\} \to T_\Sigma$, so daß $\{K_1 s_1, \ldots, K_n s_n\}$ unerfüllbar

Beachte: Weder die K_i noch die s_i müssen paarweise verschieden sein.

Beweis: F unerfüllbar $\iff$ es gibt ein $n \in \mathbb{N}$, so daß $F_1 \wedge \ldots \wedge F_n$ unerfüllbar, wobei $\{F_1, F_2, \ldots\} = E(F)$ $\iff$ es gibt ein $n \in \mathbb{N}$, Klauseln $K_1, \ldots, K_n \in \mathcal{K}$ und $s_1, \ldots, s_n : \{y_1, \ldots, y_k\} \to T_\Sigma$, so daß $\{K_1 s_1, \ldots, K_n s_n\}$ unerfüllbar ist

Beachte: Ist $\mathcal{K}' \subseteq \mathcal{K}$ unerfüllbar, so auch $\mathcal{K}$.

Korollar 2.2 (Grundresolutionssatz) Sei $F = \forall y_1 \ldots \forall y_n \widetilde{F} \in SF_\Sigma(X)$ und $\widetilde{F}$ in KNF. Dann gilt: F unerfüllbar $\iff$ es gibt Klauseln $K_1, \ldots, K_n$ mit $K_n = \square$ und für alle $i \in \{1, \ldots, n\}$ gilt:

K_i ist Grundinstanz einer Klausel $K \in \mathcal{K}(\widetilde{F})$, d.h. $K_i = K[y_1/t_1, \ldots, y_k/t_k]$ mit $T_i \in T_\Sigma$

oder

K_i ist AL-Resolvent von K_j und K_l mit $j, l < i$

Problem: Geeignete Grundinstanzen für die Herleitung von $\square$ zu finden
Strategie:

- Zurückhaltend substituieren, um Möglichkeiten offenzuhalten
- Grundsubstitution aufbrechen durch die Substitution beliebiger Terme, z.B. $[x/f(g(a))]$ in 3 Schritten $[x/f(y)][y/g(z)][z/a] \rightsquigarrow$ Prädikatenlogische Form der Resolution (J. A. Robinson)
- Resolution mit Substitution kombinieren

Beispiel 2.1 Siehe Skript 4.5.2000

Definition 2.12 (Unifikator)
Sei $K = \{L_1, \ldots, L_n\}$ eine (Σ, X)-Klausel, d.h. L_i ist ein (Σ, X)-Literal (atomare (Σ, X)-Formel, oder ihre Negation). Für $i = 1, \ldots, k$ sei $y_i \in X$ und $t_i \in T_\Sigma(X)$. Dann heißt die Substitution $sub = [y_1/t_1, \ldots, y_k/t_k]$ **Unifikator** von $K \iff L_1 sub = L_2 sub = \ldots = L_n sub$. (Kürzer: $|Ksub| = 1$)
Man sagt: K ist mit sub unifizierbar.
Ferner: sub heißt **allgemeinster Unifikator** von K, falls für jeden Unifikator sub' eine Substitution sub'' existiert, mit $sub = sub'sub''$.

Folgerung: Eindeutigkeit des allgemeinsten Unifikators (in Übung bewiesen)
Sind sub_1 und sub_2 allgemeinste Unifikatoren von K, so gibt es eine Varaible-numbenennung sub mit $sub_1 = sub_2 sub$.

Satz 2.5 (Unifikationssatz) (Robinson) Die Unifizierbarkeit einer Klausel ist entscheidbar. Eine unifizierbare Klausel besitzt einen allgemeinsten Unifikator, der sich berechnen läßt.

Beweis: Durch Konstruktion
Unifikationsalgorithmus
Eingabe: (Σ, X)-Klausel $K = \{L_1, \ldots, L_n\}$
Sei sub eine Variable vom Typ "(Σ, X)-Substitution".
sub:=[] (* leere Substitution=Identität *)
while $|Ksub| > 1$ **do**
begin
  Lese alle $L_i sub$ parallel von links nach rechts, bis in 2
  Literalen die entsprechenden Zeichen verschieden sind
  if Keines der beiden Zeichen ist eine Variable **then**
    STOP(''Nicht unifizierbar'') (* STOP1 *)
  else begin
    Sei x die Variable und t der Teilterm im anderen Literal
    mit Kopf an entsprechender Stelle
    if x kommt in t vor **then** (* occur check *)
      STOP(''Nicht unifizierbar'') (* STOP2 *)
    else
      $sub := sub[x/t]$
    endif
  end endif
end
STOP(''sub ist allgemeinster Unifikator von K'') (* STOP3 *)

Korektheit:

1) Der Unifikationsalgorithmus terminiert immer!
   Denn: Bei jedem Schleifendurchlauf verringert sich die Zahl der Variablen von $Ksub$ um 1.

2) STOP1 $\vee$ STOP2 $\iff$ K ist nicht unifizierbar
STOP3 $\iff$ K ist unifizierbar

Es fehlt noch: Bei STOP3 ist *sub* ein allgemeinster Unifikator.
Sei K unifizierbar: $|Ksub'| = 1$. Sei sub_i der Wert der Variablen *sub* nach i Schleifendurchläufen.
Behauptung: Es existiert eine Substitution s_i mit $sub_i s_i = sub'$. *Beweis:* durch Induktion über $i \in \mathbb{N}$

$i = 0 : sub_0 = [], \quad s_0 = sub' \implies sub_0 s_0 = sub' \quad \checkmark$

$i \to i+1$: Induktionsvor. für i Durchläufe $sub_i s_i = sub' \implies xs_i = ts_i$

$$sub_{i+1} = sub_i[x/t], \quad s_{i+1}(y) = \begin{cases} s_i(y) & \text{für } y \neq x \\ y & \text{für } y = x \end{cases}$$

$$sub_{i+1}s_{i+1} = sub_i[x/t]s_{i+1} \overset{x \text{ nicht in } t}{=} sub_i[x/t]s_i \overset{xs_i = ts_i}{=} sub_i s_i \overset{IV}{=} sub'$$

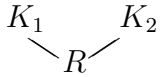
Definition 2.13 (Prädikatenlogische Resolution)
Seien K_1, K_2 und R (Σ, X)-Klauseln. Dann heißt R **Resolvent** (genauer PL-Resolvent) von K_1 und K_2, falls gilt:

(i) Es gibt Variablenumbenennungen s_1 und s_2, so daß $K_1 s_1$ und $K_2 s_2$ keine gemeinsamen Variablen enthalten.

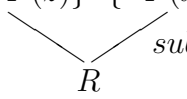
(ii) Es gibt $L_1, \ldots, L_m \in K_1 s_1$ mit $m \geq 1$ und $L'_1, \ldots, L'_n \in K_2 s_2$ mit $n \geq 1$, so daß $K := \{\overline{L_1}, \ldots, \overline{L_m}, L'_1, \ldots, L'_n\}$ unifizierbar ist, mit allgemeinstem Unifikator *sub*.

(iii) $R = ((K_1 s_1 \setminus \{L_1, \ldots, L_m\}) \cup (K_2 s_2 \setminus \{L'_1, \ldots, L'_n\}))sub$

Schreibweise:

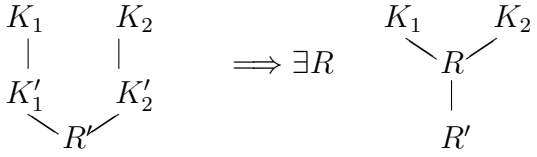


Beispiel 2.2 $\{P(f(x)), \neg Q(z), P(z)\}$ $\{\neg P(x), R(g(x)), T(a)\}$ $s_1 = [], s_2 = [x/u]$



Übertragung des Resolutionskalküls: $Res(\mathcal{K})$, $Res^2(\mathcal{K})$,..., $Res^*(\mathcal{K})$
$\rightsquigarrow$ PL-Resolutionskalkül

Lemma 2.3 (Lifting-Lemma)
Seien K_1 und K_2 PL-Klauseln ((Σ, X)-Klauseln) mit Grundinstanzen K'_1 und K'_2 und deren AL-Resolventen R'. Dann gibt es einen PL-Resolventen R von K_1 und K_2, so daß R' Grundinstanz von R ist.



Beweis: Wähle Umbenennungen s_1, s_2, so daß $K_1 s_1$ und $K_2 s_2$ keine gemeinsamen Variablen besitzen. K_i' Grundinstanz von $K_i \implies K_i'$ Grundinstanz von $K_i s_i \implies K_i' = K_i s_i sub_i$ für $i = 1, 2$. Wegen der Variablentrennung gibt es eine Substitution mit $K_i' = K_i s_i sub$ für $i = 1, 2$.

$$\begin{array}{ccc} K_1' & & K_2' \\ & \searrow \quad \swarrow & \\ & R' & \end{array} \implies \exists L \in K_1' \text{ mit } \overline{L} \in K_2', \text{ so daß } R' = (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\})$$

Wähle alle Urbilder von L bzw. $\overline{L}$ unter sub, d.h. alle $L_1, \ldots, L_m \in K_1 s1$ und $L_1', \ldots, L_n' \in K_2 s2$ mit $L = L_1 sub = \ldots = L_m sub$ und $\overline{L} = L_1' sub = \ldots = L_n' sub$. sub unifiziert also $\{\overline{L_1}, \ldots, \overline{L_m}, L_1', \ldots, L_n'\} =: K$. Sei sub_0 allgemeinster Unifikator von K, also insbesondere $sub = sub_0 s$ für gegebenes s. Dann gilt: $R := ((K_1 s_1 \setminus \{L_1, \ldots, L_m\}) \cup (K_2 s_2 \setminus \{L_1', \ldots, L_n'\}))sub_0$ ist PL-Resolvent von K_1 und K_2.
$R' = (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\}) = (K_1 s_1 sub \setminus \{L\}) \cup (K_2 s_2 sub \setminus \{\overline{L}\}) = ((K_1 s1 \setminus \{L_1, \ldots, L_m\}) \cup (K_2 s2 \setminus \{L_1', \ldots, L_n'\}))\underbrace{sub_0 s}_{sub} = Rs$

Satz 2.6 (Resolutionssatz der Prädikatenlogik) Für $F = \forall y_1 \ldots \forall y_n \widetilde{F} \in SF_\Sigma(X)$ und $\widetilde{F}$ in KNF gilt: F unerfüllbar $\iff \square \in Res^*(\mathcal{K}(\widetilde{F}))$

Beweis: "$\Longrightarrow$" (Vollständigkeit)
F unerfüllbar. Dann existiert ein Grundresolutionsbeweis von $\square$ (in AL). Mit dem Lifting-Lemma läßt sich dieser "anheben" zu einem PL-Resolutionsbeweis.

"$\Longleftarrow$" (Korrektheit)
Sei $\mathcal{K}(\widetilde{F}) = \{K_1, \ldots, K_r\}$. Für eine Klausel $K = \{L_1, \ldots, L_s\}$ mit Variablen $x_1, \ldots, x_n$ heißt $\forall K := \forall x_1 \ldots \forall x_n (L_1 \vee \ldots \vee L_s)$ die **All-Formel** von K. Dann gilt: F erfüllbar $\iff \forall K_1 \wedge \ldots \wedge \forall K_r$ erfüllbar. Wir zeigen, daß für Klauseln K, K' und R gilt:

$$\bigotimes: \begin{array}{ccc} K & & K' \\ & \searrow \quad \swarrow & \\ & R & \end{array} \implies \forall K \wedge \forall K' \models \forall R$$

Aus $\bigotimes$ folgt wegen $\square \in Res^*(\mathcal{K}(\widetilde{F}))$: $\forall K_1 \wedge \ldots \wedge \forall K_r \models \forall \square$, also die Unerfüllbarkeit von F.

Beweis für $\bigotimes$: Wähle Σ-Struktur $\mathfrak{A}$, welche $\forall K$ und $\forall K'$ erfüllt: $\mathfrak{A} \models \forall K$ und $\mathfrak{A} \models \forall K'$. $R = ((Ks \setminus \{L_1, \ldots, L_m\}) \cup (K's' \setminus \{L_1', \ldots, L_n'\}))sub$, wobei sub allgemeinster Unifikator von $\{\overline{L_1}, \ldots, \overline{L_m}, L_1', \ldots, L_n'\}$. Sei $sub = [y_1/t_1, \ldots, y_k/t_k]$.
Annahme: $\mathfrak{A} \not\models \forall R$
Dann existiert ein $\beta : X \to A$ mit $\langle \mathfrak{A}, \beta \rangle \not\models R$, also
$\langle \mathfrak{A}, \beta \rangle \not\models ((Ks \setminus \{L_1, \ldots, L_m\}) \cup (K's' \setminus \{L_1', \ldots, L_n'\}))sub$
$\implies \langle \mathfrak{A}, \beta[y_1/\beta(t_1), \ldots, y_k/\beta(t_k)] \rangle \not\models (Ks \setminus \{L_1, \ldots, L_m\}) \cup (K's' \setminus \{L_1', \ldots, L_n'\})$
$\implies \langle \mathfrak{A}, \beta[\ldots] \rangle \not\models Ks \setminus \{L_1, \ldots, L_m\}$ und $\langle \mathfrak{A}, \beta[\ldots] \rangle \not\models K's' \setminus \{L_1', \ldots, L_n'\}$
$\overset{3}{\implies} \langle \mathfrak{A}, \beta[\ldots] \rangle \models Ks$ und $\langle \mathfrak{A}, \beta[\ldots] \rangle \models K's'$
$\implies \langle \mathfrak{A}, \beta[\ldots] \rangle \models \{L_1, \ldots, L_m\}$ und $\langle \mathfrak{A}, \beta[\ldots] \rangle \models \{L_1', \ldots, L_n'\}$

$\Longrightarrow$ $\langle \mathfrak{A}, \beta \rangle \models \{L_1, \ldots, L_m\} sub =: L$ und $\langle \mathfrak{A}, \beta \rangle \models \{L'_1, \ldots, L'_n\} sub =: L'$
aber $\overline{L} = L'$ $\Longrightarrow$ Wiederspruch! (denn kein Modell kann eine Formel und ihre Negation erfüllen).

3 Logikprogramme

- Syntax und Semantik von Logikprogrammen
- deklarative, prozedurale Semantik und Fixpunktsemantik
- Universalität
- Nichtdeterminismus und Auswertungsstrategien

Definition 3.1 (Logikprogramm)
Eine nicht leere endliche Menge Lp von definiten (Σ, X)-Hornklauseln heißt **Logikprogramm** über (Σ, X). $K \in Lp$ heißt **Programmklausel**, und zwar

- **Tatsachenklausel** (Fakt), falls $K = \{P\}$
- **Prozedurklausel** (Regel), falls $K = \{P, \neg Q_1, \ldots, \neg Q_k\}$ mit $k \geq 1$ und P, Q_i atomare (Σ, X)-Formeln.

Aufruf eines Logikprogramms durch eine

- **Zielklausel** (Aufruf-, Abfrageklausel, goal) der Form $K = \{\neg Q_1, \ldots, \neg Q_k\}$ mit $k \geq 1$ und P, Q_i atomare (Σ, X)-Formeln.

Anwendung:
Sei Lp ein Logikprogramm über (Σ, X).Eine Anfrage an Lp habe die Form $G[y_1, \ldots, y_n]$ mit $G = Q_1 \wedge \ldots \wedge Q_k$, wobei die Q_i atomare (Σ, X)-Formeln sind und $\{y_1, \ldots, y_n\}$ alle freien Variablen von G enthält und bedeutet:

(i) Gibt es Grundterme $gt_1, \ldots, gt_n \in T_\Sigma$, so daß $G[y_1/gt_1, \ldots, y_n/gt_n]$ aus Lp folgt?

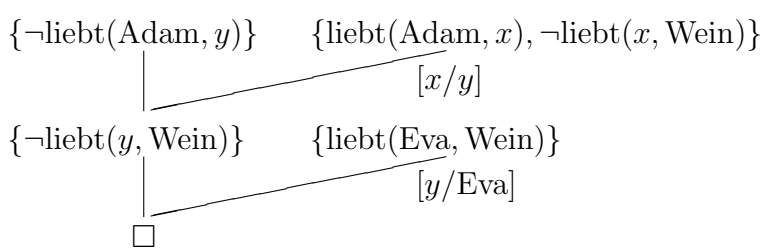
(ii) Für welche Grundterme $gt_1, \ldots, gt_n \in T_\Sigma$ folgt $G[y_1/gt_1, \ldots, y_n/gt_n]$ aus Lp?

Dabei läßt sich i) formulieren als die Frage:
Ist $Lp \cup \{\neg(\exists y_1 \ldots \exists y_n G)\}$ unerfüllbar?
oder
Ist $Lp \cup \{\neg Q_1, \ldots, \neg Q_k\}$ unerfüllbar?
Lösbarkeit mit SLD-Resolution.

Zu ii): Der Resolutionsbeweis von $\square$ aus $Lp \cup \{\neg Q_1, \ldots, \neg Q_k\}$ liefert als Seiteneffekt der PL-Resolution eine Variablenbelegung. Ihre Einschränkung auf die freien Variablen der Anfrage heißt **Antwortsubstitution**.

Beispiel 3.1 Lp sei gegeben durch die Fakten $\{\text{liebt}(\text{Eva}, \text{Essen})\}$, $\{\text{liebt}(\text{Eva}, \text{Wein})\}$ und die Regel $\{\text{liebt}(\text{Adam}, x), \neg\text{liebt}(x, \text{Wein})\}$.

Anfrage:



Antwortsubstitution: $[y/\text{Eva}]$

Konvention: Variablenumbenennung bei Resolutionsschritten nur in Programmklauseln $\Longrightarrow$ standardisierte SLD-Resolution

Definition 3.2 (Deklarative Semantik eines Logikprogramms)
Sei Lp ein Logikprogramm über (Σ, X) mit Zielklausel $G = \{\neg A_1, \ldots, \neg A_k\}$. Dann ist die **deklarative Semantik** von Lp bezüglich G definiert als $\mathcal{D}[\![Lp, G]\!] := \{H | Lp \models H,\ H \text{ Grundinstanz von } A_1 \wedge \ldots \wedge A_k\}$.

Bemerkung:

1) Jede solche Grundinstanz H enthält als "Lösung" die entsprechenden Grundsubstitutionen der Variablen aus $A_1, \ldots, A_k$.
2) Statische, modelltheoretische Semantik

Prozedurale Semantik
Idee: Berechnung der Grundinstanz aus $\mathcal{D}[\![Lp, G]\!]$ in 2 Stufen

1) Ableitung von Rechenergebnissen mit SLD-Resolution
2) Grundsubstitutionen freier Variablen in Rechenergebnissen

Berechnung = Folge von Konfigurationen
Konfiguration = (negative Klausel, Substitution)
$\Longrightarrow$ Protokoll der Unifikationen

Definition 3.3 (Prozedurale Semantik eines Logikprogramms)
Sei Lp ein Logikprogramm über (Σ, X).

- Eine **Konfiguration** ist ein Paar (G, sub) mit $G = \{\neg A_1, \ldots, \neg A_k\}$, A_i atomare (Σ, X)-Formel und $sub : X \to T_\Sigma(X)$
- Ein Rechenschritt $(G_1, sub_1) \vdash_{Lp} (G_2, sub_2)$ ist definiert, falls gilt:
  - $G_1 = \{\neg A_1, \ldots, \neg A_k\}$ mit $k \geq 1$
  - Es gibt $K = \{B, \neg C_1, \ldots, C_n\} \in Lp$ mit $n \geq 0$ und $i \in \{1, \ldots, k\}$, so daß
    - G_1 und K haben keine gemeinsamen Variablen, eventuell Umbenennung der Variablen in K

$*$ A_i und B unifizierbar mit allgemeinstem Unifikator s
$*$ $G_2 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, A_k\}s$
$*$ $sub2 = sub1 \cdot s$

- Eine **Berechnung** von Lp bei Eingabe von $G = \{\neg A_1, \ldots, \neg A_k\}$ ist eine (endliche oder unendliche) Folge von Konfigurationen (G_0, sub_0), $(G_1, sub_1), \ldots$ mit $G_0 = G$, $sub_0 = []$ und $(G_i, sub_i) \vdash_{Lp} (G_{i+1}, sub_{i+1})$.
- Eine mit $(\Box, sub)$ terminierende Berechnung heißt **erfolgreich** mit dem **Rechenergebnis** $(A_1 \wedge \ldots \wedge A_k)sub$.

Dann ist die **prozedurale Semantik** von Lp bezüglich G definiert als
$\mathcal{P}[\![Lp, G]\!] := \{H | (G, []) \vdash^*_{Lp} (\Box, sub)$ und H Grundinstanz von $(A_1 \wedge \ldots \wedge A_k)sub\}$

Bemerkung:

- Rechenschritte sind standardisierte SLD-Resolutionen mit Buchführung über Substitution
- Nicht-Determinismus in den Berechnungen
  a) Auswahl des negativen Resolutionsliterals $\neg A_i$
  b) Auswahl der Programmklausel K

Beispiel 3.2 Lp sei gegeben durch $\{P(x,z), \neg Q(x,y), \neg P(y,z)\}$, $\{P(u,u)\}$, $\{Q(a,b)\}$ mit $var: u, v, x, y, z$ und $const: a, b$. Anfrage $G = \{\neg P(v,b)\}$

a) Eine nicht-erfolgreiche Berechnung
$(\{\neg P(v,b)\}, []) \vdash (\{\neg Q(v,y), \neg P(y,b)\}, [x/v, z/b])$
$\vdash (\{\neg P(b,b)\}, [x/v, z/b, v/a, y/b])$
$\vdash (\{\neg Q(b,y), \neg P(y,b)\}, [x/v, z/b, v/a, y/b, x/b, z/b])$
$\vdash (\{\neg Q(b,b)\}, [x/v, z/b, v/a, y/b, x/b, z/b, y/b, u/b])$
$\Longrightarrow$ "failure", nicht erfolgreich

b) Eine kurze erfolgreiche Berechnung
$(\{\neg P(v,b)\}, []) \vdash (\{\Box\}, [v/b, u/b]) \rightsquigarrow P(b,b)$ Rechenergebnis

c) Eine längere erfolgreiche Berechnung
$(\{\neg P(v,b)\}, []) \vdash (\{\neg Q(v,y), \neg P(y,b)\}, [x/v, z/b])$
$\vdash (\{\neg P(b,b)\}, [x/v, z/b, v/a, y/b])$
$\vdash (\{\Box\}, [x/v, z/b, \underline{v/a}, y/b, u/b]) \rightsquigarrow P(a,b)$ Rechenergebnis

Satz 3.1 (Clark) Sei Lp ein Logikprogramm mit Zielklausel $G = \{\neg A_1, \ldots, \neg A_k\}$. Dann gilt: $\mathcal{D}[\![Lp, G]\!] = \mathcal{P}[\![Lp, G]\!]$

Beweis:

1) Korrektheit der prozeduralen Semantik bezüglich der deklarativen Semantik, also $\mathcal{P}[\![Lp, G]\!] \subseteq \mathcal{D}[\![Lp, G]\!]$
Zu zeigen: JedeGrundinstanz eines Rechenergebnisses $(A_1 \wedge \ldots \wedge A_k)sub$ von Lp mit Eingabe G ist Folgerung von Lp.

Beweis: durch Induktion über die Länge n der Berechnung

$n = 0$: $G = \square$, keine Grundinstanzen, Beh. trivialerweise erfüllt ($\emptyset = \emptyset$)

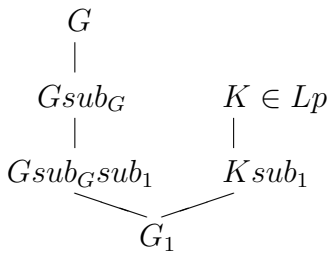
$n \to n+1$: Sei $(G, []) \vdash_{Lp} (G_1, sub_1) \vdash_{Lp} \ldots \vdash_{Lp} (\square, sub_1 \cdot \ldots \cdot sub_{n+1})$ eine Berechnung der Länge $n+1$. Dann ist $G = \{\neg A_1, \ldots, \neg A_k\}$,
$G_1 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_l, \neg A_{i+1}, \ldots, A_k\}sub_1$ mit
$\{B, \neg C_1, \ldots, \neg C_l\} \in Lp$ $(l \geq 0)$ und $\{B, A_i\}$ unifizierbar mit allgemeinstem Unifikator sub_1.
Für die Teilberechnung $(G_1, []) \vdash_{Lp} \ldots \vdash_{Lp} (\square, sub_2 \cdot \ldots \cdot sub_{n+1})$ folgt nach IV, daß jede Grundinstanz von $(A_1 \wedge \ldots \wedge A_{i-1} \wedge C_1 \wedge \ldots \wedge C_l \wedge \ldots \wedge A_{i+1} \wedge \ldots \wedge A_k)sub_1 \cdot \ldots \cdot sub_{n+1}$ eine Folgerung von Lp ist.
Also auch jede Grundinstanz von $(C_1 \wedge \ldots \wedge C_l)sub_1 \cdot \ldots \cdot sub_{n+1}$.
Da $\{B, \neg C_1, \ldots, \neg C_l\} \in Lp$, also $C_1 \wedge \ldots \wedge C_l \to B$ und $Bsub_1 \cdot \ldots \cdot sub_{n+1} = A_i sub_1 \cdot \ldots \cdot sub_{n+1}$, ist auch jede Grundinstanz von $(A_1 \wedge \ldots \wedge A_k)sub_1 \cdot \ldots \cdot sub_{n+1}$ eine Folgerung von Lp.

2) Vollständigkeiteit der prozeduralen Semantik bezüglich der deklarativen Semantik, also $\mathcal{D}[\![Lp, G]\!] \subseteq \mathcal{P}[\![Lp, G]\!]$
Sei $H \in \mathcal{D}[\![Lp, G]\!]$. Dann existiert eine Grundsubstitution sub_G mit $H = (A_1 \wedge \ldots \wedge A_k)sub_G$ und $Lp \models H$, bzw. $Lp \cup \{\{\neg A_1, \ldots, \neg A_k\}sub_G\}$ ist unerfüllbar. Wegen der Vollständigkeit der SLD-Resolution existiert eine erfolgreiche Berechnung $(\{\neg A_1, \ldots, \neg A_k\}sub_G, []) \vdash^*_{Lp} (\square, sub_1 \cdot \ldots \cdot sub_n)$.

$(*)$ Diese Berechnung läßt sich liften zu $(\{\neg A_1, \ldots, \neg A_k\}, []) \vdash^*_{Lp} (\square, sub_1 \cdot \ldots \cdot sub_n)$ mit $sub_G \cdot sub_1 \cdot \ldots \cdot sub_n = sub'_1 \cdot \ldots \cdot sub'_n \cdot sub'_G$, wobei die sub_i allg. Unif. sind und sub'_G passende Grundsubstitution ist.

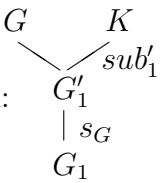
Damit folgt: $H = (A_1 \wedge \ldots \wedge A_k)sub_G = (A_1 \wedge \ldots \wedge A_k)sub_G \cdot sub_1 \cdot \ldots \cdot sub_n = (A_1 \wedge \ldots \wedge A_k)sub'_1 \cdot \ldots \cdot sub'_n \cdot sub'_G \implies H \in \mathcal{P}[\![Lp, G]\!]$

Beweis von $(*)$*:* $G = \{\neg A_1, \ldots, \neg A_k\}$
1. Berechnungsschritt:



Beachte: G und K haben keine gemeinsamen Variablen, also
$Ksub_1 = Ksub_Gsub_1$
Wähle allg. Unif. sub'_1 mit $sub_Gsub_1 = sub'_1 s_G$

Liften: $\implies$



Durch Iteration über alle Berechnungsschritte folgt $(*)$.
Beachte: Zum Beweis von $(*)$ benutzen wir eine Verallgemeinerung des Lifting-Lemmas:

Lemma 3.1 $(Gsub, []) \vdash_{Lp} (G_1, sub_1) \implies (G, []) \vdash_{Lp} (G'_1, sub'_1)$ mit $sub \cdot sub_1 = sub'_1 \cdot sub'$ mit geignetem sub' und $G = G'_1 sub'_1$.

Beweis: Schöning, 5. Auflage, Übung 106

Fixpunktsemantik:
Sei $\mathcal{G}$ die Menge der atomaren Σ-Formeln, das sind die variablenfreien (Σ, X)-Primformeln. Ein Logikprogramm Lp über (Σ, X) bestimmt eine Transformation $trans_{Lp} : \mathcal{P}(\mathcal{G}) \to \mathcal{P}(\mathcal{G})$ mit $trans_{Lp}(M) := \{A' |$ Es gibt $\{A, \neg B_1, \ldots, \neg B_k\} \in Lp$ mit $k \geq 0$ und davon eine Grundinstanz $\{A', \neg B'_1, \ldots, \neg B'_k\}$, so daß $B'_1, \ldots, B'_k \in M\}$.
$trans_{Lp}$ beschreibt die logischen Implikationen aus M mit Lp. Dann gilt:

1) $\langle \mathcal{P}(\mathcal{G}); \subseteq \rangle$ ist eine vollständige Halbordnung, d.h. insbesindere: jede gerichtete Teilmenge von $\mathcal{P}(\mathcal{G})$ hat eine kleinste obere Schranke.
   $T \subseteq \mathcal{P}(\mathcal{G})$ ist **gerichtet** $\iff \forall M_1, M_2 \in T \exists M \in T : M_1 \subseteq M$ und $M_2 \subseteq M$
2) $trans_{Lp}$ ist monoton: $M_1 \subseteq M_2 \implies trans_{Lp}(M_1) \subseteq trans_{Lp}(M_2)$
3) $trans_{Lp}$ ist stetig: Für jede gerichtete Teilmenge $D \subseteq \mathcal{P}(\mathcal{G})$ gilt:
   $trans_{Lp}(\bigcup_{M \in D} M) = \bigcup_{M \in D} trans_{Lp}(M)$

1)+2) sind klar
3) "$\supseteq$" folgt aus Monotonie
   "$\subseteq$": Sei $A' \in trans_{Lp}(\bigcup_{M \in D} M)$. Dann existiert $\{A, \neg B_1, \ldots, \neg B_k\} \in Lp$ mit Grundinstanz $\{A', \neg B'_1, \ldots, \neg B'_k\}$ wobei $B'_1, \ldots, B'_k \in \bigcup_{M \in D} M$. Dann existiert ein $M_i \in D$ mit $B'_i \in M_i$ für $1 \leq i \leq k$. Da D gerichtet, existiert ein $M \in D$ mit $B'_1, \ldots, B'_k \in M$ und somit $A' \in trans_{Lp}(M) \subseteq \bigcup_{M \in D} trans_{Lp}(M)$.

Satz 3.2 (Satz von Tarski)
$trans_{Lp}$ besitzt als kleinsten Fixpunkt $fix(trans_{Lp}) = \bigcup_{n \in \mathbb{N}} trans^n_{Lp}(\emptyset)$.

Bemerkung: $trans_{Lp}(\emptyset)$ enthält die Grundinstanzen der Tatsachenklauseln. Daraus kann man mit den Prozedur-Klauseln sukkzessive weitere Grundinstanzen ableiten.

Definition 3.4 (Fixpunkt-Semantik eines Logikprogramms)
Sei Lp ein Logikprogramm mit Zielklausel $G = \{\neg A_1, \ldots, \neg A_k\}$. Dann ist die **Fixpunkt-Semantik** von Lp bezüglich G definiert durch $\mathcal{F}p[\![Lp, G]\!] := \{H | H$ ist Grundinstanz $A'_1 \wedge \ldots \wedge A'_k$ von $A_1 \wedge \ldots \wedge A_k$ und $A'_i \in fix(trans_{Lp})\}$.

Satz 3.3 $\mathcal{D}[\![Lp, G]\!] = \mathcal{P}[\![Lp, G]\!] = \mathcal{F}p[\![Lp, G]\!]$

Beweis:

(i) $\mathcal{P}[\![Lp, G]\!] \subseteq \mathcal{F}p[\![Lp, G]\!]$
Analog zum Beweis von $\mathcal{P}[\![Lp, G]\!] = \mathcal{D}[\![Lp, G]\!]$ durch Induktion über die Länge der erfolgreichen Berechnungen:
Induktionsschritt: Rolle von $Cons(Lp)$ wird von $fix(trans_{Lp})$ übernommen, beide sind unter logischen Implikationen abgeschlossen.

(ii) $\mathcal{F}p[\![Lp, G]\!] \subseteq \mathcal{D}[\![Lp, G]\!]$
$G = \{\neg A_1, \ldots, \neg A_k\}$, $H = A'_1 \wedge \ldots \wedge A'_k \in \mathcal{F}p[\![Lp, G]\!]$ und H ist Grundinstanz von $A_1 \wedge \ldots \wedge A_k$ mit $A'_i \in \mathcal{F}p[\![Lp, G]\!]$.
Zu zeigen: $Lp \models H$
Dies folgt aus $(*)$ $A \in fix(trans_{Lp}) \implies Lp \models A'$
Beweis von $(*)$ durch Induktion über $n \in \mathbb{N}$ mit $fix(trans_{Lp}) = \bigcup_{n=0}^{\infty} trans_{Lp}(\emptyset)$

$n = 0$: $trans^0_{Lp}(\emptyset) = \emptyset$ enthält kein A'.

$n \to n+1$: $A' \in trans^{n+1}_{Lp}(\emptyset) = trans_{Lp}(trans^n_{Lp}(\emptyset))$. Also existiert $\{A, \neg B_1, \ldots, \neg B_l\} \in Lp$ mit Grundinstanz $\{A', \neg B'_1, \ldots, \neg B'_l\}$ und alle $B'_j \in trans^n_{Lp}(\emptyset)$. Nach IV $Lp \models B'_j$, also auch $Lp \models A'$.

Aus i) und ii) folgt $\mathcal{D}[\![Lp, G]\!] = \mathcal{P}[\![Lp, G]\!] = \mathcal{F}p[\![Lp, G]\!]$.

Universalität
Programmierbarkeit berechenbarer Funktionen
Imp. Progr.: Jede berechenbare arithmetische Fkt. ist WHILE-programmierbar
Log. Progr.: Jede berechenbare arithmetische Fkt. ist LP-berechenbar

Man sagt: LP ist **universell**.

Definition 3.5 (Berechnung aritmetischer Funktionen)

- Eine arithmetische Funktion $f : \mathbb{N}^n \dashrightarrow \mathbb{N}$ wird dargestellt durch ihren Graphen $\overline{f}(k_1, \ldots, k_n, k) \iff f(k_1, \ldots, k_n) = k$
- $k \in \mathbb{N}$ wird dargestellt durch den Term $k := \underbrace{S(S(\ldots(S}_{k \text{ mal}}(0)\ldots))$
- Ein Logikprogramm Lp berechnet $f : \mathbb{N}^n \dashrightarrow \mathbb{N} \iff$ es existiert $\overline{f} \in \Pi^{(n+1)}$, so daß für alle $k, k_1, \ldots, k_n \in \mathbb{N}$ $f(k_1, \ldots, k_n) = k \iff Lp \models \overline{f}(k_1, \ldots, k_n, k)$

Definition 3.6 (μ-rekursive Funktionen)
Die Klasse $Rek = \bigcup_{k \in \mathbb{N}} Rek^{(n)}$ der **μ-rekursiven Funktinen** ist die kleinste Klasse arithmetischer Funktionen mit folgenden Eigenschaften

(i) Die Grundfunktionen $null^{(n)}$, suc und $proj_i^{(n)}$ sind μ-rekursiv, wobei $null^{(n)}(k_1, \ldots, k_n) = 0$, $suc(k) = k + 1$ und $proj_i^{(n)}(k_1, \ldots, k_n) = k_i$

(ii) Rek ist abgeschlossen unter Komposition, primitiver Rekursion und unbeschränkter Minimalisierung, d.h.

- $f \in Rek^{(m)}$, $f_1, \ldots, f_m \in Rek^{(n)} \implies comp(f; f_1, \ldots, f_m)(\widetilde{k}) = f(f_1(\widetilde{k}), \ldots, f_m(\widetilde{k}))$ "**Komposition**"
- $f \in Rek^{(n)}$, $g \in Rek^{(n+2)} \implies prim(f, g) \in Rek^{(n+1)}$ "**primitive Rekursion**", wobei $h := prim(f, g)$ definiert durch $h(\widetilde{k}, 0) = f(k)$ und $h(\widetilde{k}, k + 1) = g(\widetilde{k}, k, h(\widetilde{k}, k))$
- $f \in Rek^{(n+1)} \implies min(f) \in Rek^{(n)}$ "**Minimalisierung**", wobei $g := min(f)$ definiert durch $g(\widetilde{k}) = k \iff f(\widetilde{k}, k) = 0$ und für alle $0 \leq k' < k$ gilt $f(\widetilde{k}, k') > 0$ (insbesondere muß $f(\widetilde{k}, k')$ definiert sein!).

Satz 3.4 (Universalität der LP) Jede μ-rekursive Funktion ist durch ein Logikprogramm berechenbar.

Beweis: durch Induktion über den Aufbau von Rek

(i) Grundfunktion: $null^{(n)}$ wird berechnet durch $\{Null(x_1, \ldots, x_n, 0)\}$, suc wird berechnet durch $\{Suc(X, S(X))\}$ und $proj_i^{(n)}$ wird berechnet durch $\{Proj_i(x_1, \ldots, x_n, x_i)\}$

(ii) a) **Komposition:** Sei $h = comp(f; f_1, \ldots, f_m)$
Nach IV: f werde berechnet durch das Logikprogramm Lp_f, mit Hilfe von $\overline{f}$. f_i werde berechnet durch das Logikprogramm Lp_{f_i}, mit Hilfe von $\overline{f_i}$. Dann wird h berechnet durch das Logikprogramm Lp_h, mit Hilfe von $\overline{h}$:
$Lp_h := \{\{\overline{h}(x_1, \ldots, x_n, z), \neg\overline{f_1}(x_1, \ldots, x_n, y_1), \ldots, \neg\overline{f_m}(x_1, \ldots, x_n, y_m)\}\} \cup Lp_f \cup \bigcup_{i=1}^{m} Lp_{f_i}$

b) **Primitive Rekursion:** Sei $h = prim(f, g)$
Nach IV: f werde berechnet durch das Logikprogramm Lp_f, mit Hilfe von $\overline{f}$. g werde berechnet durch das Logikprogramm Lp_g, mit Hilfe von $\overline{g}$. Dann wird h berechnet durch das Logikprogramm Lp_h, mit Hilfe von $\overline{h}$:
$Lp_h := \{\{\overline{h}(x_1, \ldots, x_n, 0, z), \neg\overline{f}(x_1, \ldots, x_n, z)\}, \{\overline{h}(x_1, \ldots, x_n, S(x), z), \neg\overline{h}(x_1, \ldots, x_n, x, y), \neg\overline{g}(x_1, \ldots, x_n, x, y, z)\}\} \cup Lp_f \cup Lp_g$

c) **Minimalisierung:** Sei $g = min(f; f_1, \ldots, f_m)$
Also $g(x_1, \ldots, x_n) = x \iff f(x_1, \ldots, x_n, x) = 0$ und für alle $0 \leq x' \leq x$ gilt $f(x_1, \ldots, x_n, x') > 0$, insbesondere $f(x_1, \ldots, x_n, x')$ definiert.
Nach IV: f werde berechnet durch das Logikprogramm Lp_f, mit Hilfe von $\overline{f}$. Dann wird g berechnet durch das Logikprogramm Lp_g, mit

Hilfe von $\overline{g}$:
$Lp_g := \{\{\overline{g}(x_1,\ldots,x_n,z), \neg\overline{f}(x_1,\ldots,x_n,z,0), \neg\widehat{f}(x_1,\ldots,x_n,z)\},$
$\{\neg\widehat{f}(x_1,\ldots,x_n,0)\}, \{\neg\widehat{f}(x_1,\ldots,x_n,S(x)), \neg\widehat{f}(x_1,\ldots,x_n,x),$
$\neg\overline{f}(x_1,\ldots,x_n,x,S(u))\}\} \cup Lp_f$
Beachte: $\neg\widehat{f}(x_1,\ldots,x_n,x) \iff \forall x' : (0 \leq x' \leq x) \implies f(x_1,\ldots,x_n,x) > 0$
"> 0" $\iff$ $S(u)$ (Nachfolger von irgendwas, d.h. mindestens 1)

Nichtdeterminismus und Auswetungsstrategien
Prozedurale Semantik von Logikprogrammen ist nichtdeterministisch: Eine Konfiguration (G, sub) kann mehrere Folgekonfigurationen (G, sub_i) mit $(G, sub) \vdash_{Lp} (G, sub_i)$ für alle $i = 1,\ldots,n$ haben.
2 Gründe:

a) Resolution mit verschiedenen Programmklauseln möglich =(**Nichtdeterminismus erster Art**)=
b) Resolution mit verschiedenen Literalen von G möglich =(**Nichtdeterminismus zweiter Art**)=

Beispiel 3.3 Logikprogramm: $\ldots\{B,\neg D\}\quad\{B\}\quad\{B,\neg E,\neg F\}\ \ldots$
Zielklausel: $G = \{\neg A,\neg B,\neg C\}$

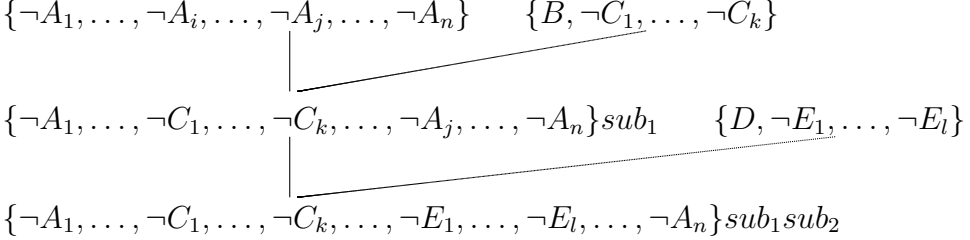
	$\{\underline{\neg A},\neg B,\neg C\}$	$\{\neg A,\underline{\neg B},\neg C\}$	$\{\neg A,\neg B,\underline{\neg C}\}$
$\{B,\neg D\}$		$\{\neg A,\neg C,\neg D\}$	
$\{B\}$		$\{\neg A,\neg C\}$	
$\{B,\neg E,\neg F\}$		$\{\neg A,\neg C,\neg E,\neg F\}$	

$\implies$ Implementierung erfordert Auswahlstrategie

ND2:

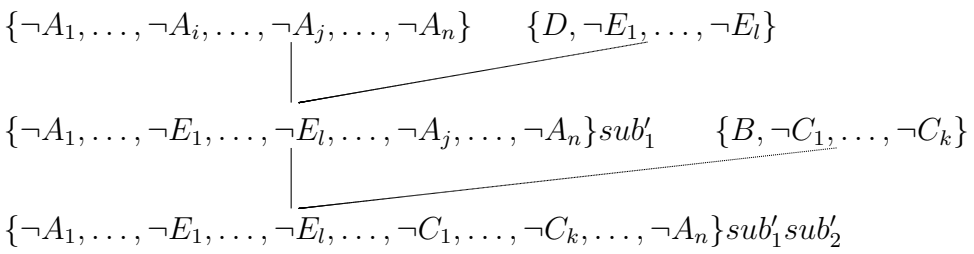
- Auswahl des Zielklauselliterals (UND-Parallelität)
- "Don't care"-ND: Die Auswahl beeinflußt das Rechenergebnis nicht (CR-Eigenschaft)

Lemma 3.2 (Vertauschungslemma) Die Reihenfolge der Zielklauselliterale, nach denen resolviert wird, kann wie folgt vertauscht werden:



dann existieren Unifikatoren sub'_1 und sub'_2, so daß $sub_1 sub_2 = sub'_1 sub'_2 u$ für eine

Variablenumbenennung u und es gilt:



Beweis:
O.B.d.A. sind die Variablenmengen der Zielklausel und der einzelnen Programmklauseln paarweise disjunkt.
$A_j sub_1$ und D unifizierbar mit allg. Unif. $sub_2 \implies A_j sub_1 sub_2 = D sub_2 = D_s sub_1 sub_2$ $(*)$ (wegen getrennter Variablen). Sei sub'_1 allg. Unif. von A_j und D, also $sub_1 sub_2 = sub'_1 s$ für geeignetes s. Weiter folgt: $A_i sub'_1$ und B sind mit s unifizierbar. $(A_i sub'_1 s = A_i sub_1 sub_2 = B sub_1 sub_2 = B sub'_1 s = Bs)$ Sei sub'_2 allgemeinster Unifikator von $A_i sub'_1$ und D, also $s = sub'_2 s'$. Es folgt $sub_1 sub_2 = sub'_1 sub'_2 s$.
Rest des Beweises: Schöning, Logik für Informatiker, Seite 140 ff

Folgerung: Wählt man eine Ordnung der Literale der Zielklausel, so kann o.B.d.A. stets mit dem ersten Literal der Zielklausel resolviert werden, ohne das Ergebnis zu beeinflußen.

Definition 3.7
Die Berechnung eines Logikprogramms heißt **kanonisch**, wenn bei jedem Rechenschritt nach dem ersten Literal resolviert wird.

Satz 3.5 (Elemination des Nichtdeterminismus 2. Art) Sei Lp ein Logikprogramm mit Eingabe G, und alle Klauseln sind geordnet. Dann existiert zu jeder Berechnung $(G, []) \vdash^n_{Lp} (\square, sub)$ eine kanonische Berechnung von $(\square, sub)$ mit gleicher Länge.

Beweis: Sukzessives Anwenden des Vetauschungslemmas.

Korollar 3.1 Die Vollständigkeit der SLD-Resolution bleibt mit jeder Auswahlregel für die Wahl des nächsten Resolutionsliterals der Zielklausel erhalten.

Im Folgenden betrachten wir nur noch (o.B.d.A.) kanonische Berechnungen.

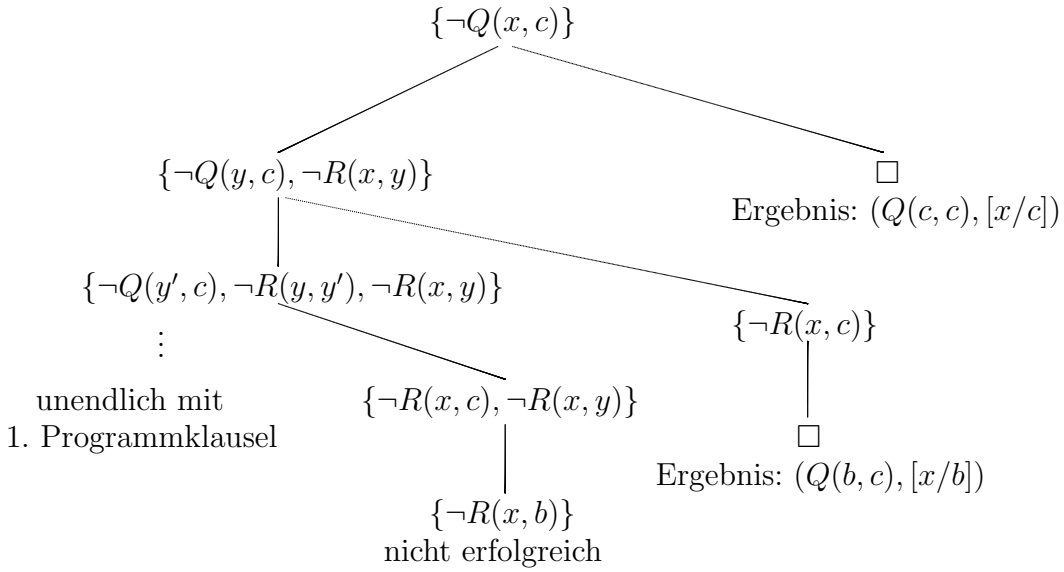
Nichtdeterminismus 1. Art
Auswahl der Programmklausel (ODER-Parallelität)
Darstellung kanonischer Berechnungen durch SLD-Bäume
Auswetungsstrategien $\widehat{=}$ Baumdurchläufe

Definition 3.8 (SLD-Baum)
Sei Lp ein Logikprogramm und G eine Zielklausel. Der **SLD-Baum** von Lp bei Eingabe G ist ein endlicher oder unendlicher Baum, dessen Knoten mit Konfigurationen markiert sind, so daß gilt:

1) Die Wurzel ist mit $(G, [])$ markiert
2) Ist ein Knoten mit (G', s') markiert, so sind die Markierungen der direkten Nachfolgerknoten genau die in einem kanonischen Rechenschritt von (G', s') erreichbaren Konfigurationen.

Beispiel 3.4 Sei Lp gegeben durch die Programmklauseln
$\{Q(x,z), \neg Q(y,z), \neg R(x,y)\}$, $\{Q(x,x)\}$, $\{R(b,c)\}$.
Anfrage: $\{\neg Q(x,c)\}$

SLD-Baum (ohne Protokoll der Substitutionen)



Bemerkungen:

- Es gibt erfolgreiche Berechnungen mit verschiedenen Ergebnissen
- Es gibt nicht erfolgreiche Berechnungen
  - endliche mit nicht resolvierbarer Zielklausel
  - unendliche

Auswertungsstrategien: Lauf durch den SLD-Baum
Mögliche Ziele:

- Eine erfolgreiche Berechnung
- Alle erfolgreichen Berechnungen

Strategietypen:

a) Breitensuche (bredth-first-search, BFS)
Besuche alle Knoten der Tiefe t und dann die Knoten der Tiefe $t + 1$, beginnend mit Tiefe 1.
Dabei gilt: Jede erfolgreiche Berechnung (jedes Ergebnis) wird nach endlich vielen Schritten gefunden.
Also: BFS vollständig
Aber: ineffizient, hoher Zeit- und Platzbedarf

b) Tiefensuche (depth-first-search, DFS)
Rekursiver Abstieg in Teilbäume
Backtracking: Rückkehr zu Elternknoten bei Auffinden einer nicht-erfolgreichen Berechnung, nächste Möglichkeit wählen
Also: DFS unvollständig wegen unendlicher Pfade
Aber: Effizienter gegenüber BFS

4 Prolog

Prolog="Programming in logic"
Ausbau von Logikprogrammierung zur Programmiersprache

Historische Betrachtung:

1965 Resolutionsprinzip von Robinson, Unifikation

1972-75 Forschungsgruppe in Marseille (colmerauer)
Effiziente Implementierung der Resolution → Prolog

1974 semantische Grundlagen (Kowalski)

1977 DEC-10-Prolog (Pereira/Warren) → WAM (Warren abstract machine)

1981 Wahl von Prolog als Sprache für Japans "5th Generation Computer Project" ⇝ weltweite Akzeptanz von Prolog als KI-Sprache

Anwendungsgebiete:

- KI, Expertensysteme
- Deduktive Datenbanken
- Verarbeitung natürlicher Sprache
- Symbolische Mathematik
- Prototyping

Vorteile von Prolog:

- Verbreiteter Sprachstandard mit vielen Implementierungen, hier: SWI-Prolog
- Klares Grundkonzept mit einfacher Syntax und Semantik
- Problemorientierte einfache Programmierung

Nachteile von Prolog:

- Programmführung manchmal ineffizient
- Kaum Unterstützung für den Entwurf großer Systeme

4.1 Syntax und Semantik

Ein **Prolog-Programm** ist eine Folge von definiten Hornklauseln, also Fakten der Form `B.` und Prozedurklauseln (Regeln) der Form `B:-`C_1`,...,`C_n`.`. Eine Zielklausel (Anfrage) hat die Form `?-`A_1`,...,`A_k`.`. Dabei sind A_i`,B,`C_j atomare Formeln. Schreibweise:

- Funktions- und Prädikatsnamen beginnen mit Kleinbuchstaben
- Variablenbezeichner beginnen mit Großbuchstaben oder Unterstrich _

Beispiel 4.1

```
(1) pfad(X,Z):-kante(X,Y),pfad(Y,Z).
(2) pfad(X,X).
(3) kante(a,b).
```

Anfrage: `?-pfad(U,b).`

Auswertung

SLD-Resolution + kanonische Berechnungen ($\widehat{=}$ SLD-Baum), "depth-first/left-to-right"-Strategie für den Durchlauf des SLD-Baums

Beachte: Unvollständigkeit dieser Strategie, Abhängig von der Reihenfolge der Klauseln in Programm und Anfrage $\Longrightarrow$ Reihenfolge so wählen, daß unendliche Berechnungen möglichst umgangen werden.

Wiederspruch zum Prinzip der deklarativen Programmierung, nach der der Programmierer nur das Problem, nicht aber die Auswertung spezifiziert. Trennung von Logik und Kontrolle nicht voll erreicht.

Prolog-Auswertungsstrategien

Eingabe: $P_p = (K_1, \ldots, K_n)$ mit $K_i = B_i : -C_{i_1}, \ldots, C_{i_{n_i}}.$ und Zielklausel $G = ? - A_1, \ldots, A_k.$

Hauptprogramm

```
success := false;
eval(G,[]);
if success then write ('ja') else write('nein');

procedure eval(G : Zielklausel, sub : Substitution)
var i : Integer;  (* Auswahl der Programmklausel *)
begin
  if G = □ then
  begin
    H := (A1 ∧ ... ∧Ak)sub;
    write ('Ergebnis:' + H);
    success := true;
  end
  else  (* G habe die Form G=?-D1,...,Dl)
```

```
begin
  i := 0;
  while (i<n) and not success do
  begin
    i := i+1;
    if D_1 und B_i unifizierbar mit allg. Unif. s then
      eval(?-(C_{i_1},...,C_{i_{n_i}},D_2,...,D_l)s,sub·s)
  end
end
end
```

Beachte:

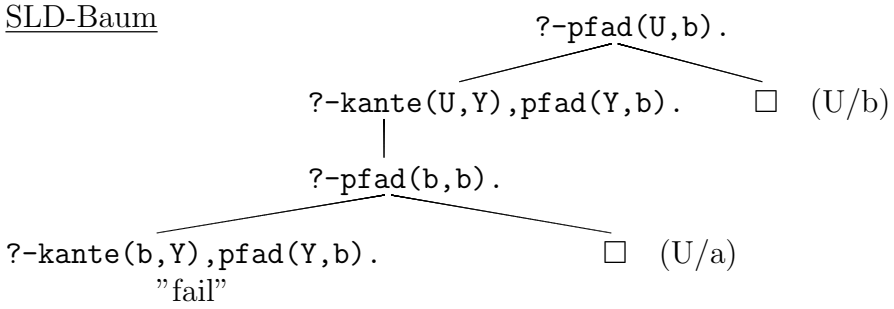
- Abbruch bei 1. Ergebnis wegen "`success = true`"
- Backtracking bei erfolgloser Unifikation am Ende der While-Schleife

SWI-Prolog: Weitere Lösungen mit `<n>` oder `<;>`

Beispiel 4.2 Bearbeitung der Anfrage aus Beispiel 4.1: `?-pfad(U,b)`

$\Longrightarrow$ `?-kante(U,Y),pfad(Y,b).`	[X/U,Z/b]
$\Longrightarrow$ `?-pfad(b,b).`	[U/a,Y/b]
$\Longrightarrow$ `?-kante(b,Y),pfad(Y,b).`	[X/b,Z/b] $\Longrightarrow$ Fehlschlag!
$\overset{Backtracking}{\Longrightarrow}$ $\square$	[X/b] $\Longrightarrow$ Ergebnis: U/a
$\overset{Backtracking}{\Longrightarrow}$ $\square$	[U/a,Y/b] $\Longrightarrow$ Ergebnis: U/b

SLD-Baum



4.2 Arithmetik

primitiver Ansatz: natürliche Zahlen als Term $null, succ(null), \ldots$

Beispiel 4.3 Addition: $Add(n,0) = n, Add(n,m+1) = Add(n,m)+1$

```
add(X,null,X).
add(X,succ(Y),succ(Z)):-add(X,Y,Z).
```

Abfrage: `?-add(null,succ(null),U).`

`?-add(succ(null),U,succ(succ(null))).`

Also: Funktionen als Relationen $\curvearrowright$ Funktionsargumente und Funktionswerte gleichberechtigt

Steigerung der Effizienz durch Einbau arithm. Ausdrücke und ihrer Auswertung
Vordefinierter Bereich: Zahlen (int,float) mit einigen Grundfunktionen und Prädikaten

Beachte: Zwei Auffassungen von Termen

- syntaktisch: frei $\rightsquigarrow$ Unifikation
- semantisch $\rightsquigarrow$ Auswertung

Arithmetische Ausdrücke sind aufgebaut aus Zahlen, Variablen und folgenden unären und binären Operatoren $+, -, *, /$ sowie // für ganzzahlige Division, mod für ganzzahligen Rest und trigonometrischen Funktionen (SWI-Manual, 3.27).

Prädikate

1) Vergleich: $t_1\ op\ t_2$ mit $op \in \{<, >, =<, >=, =:= (gleich), = \backslash = (ungleich)\}$
   Erfolg, falls t_i auswertbar zu z_i $(i = 1, 2)$ und $z_1\ op\ z_2$ gültig.
   Beachte: t nur auswertbar, wenn Werte der Variablen bekannt.
2) Wertzuweisung: $t_1\ is\ t_2$
   Erfolg, falls t_2 auswertbar zu z_2 und t_1 mit z_2 unifizierbar.

Beispiel 4.4
`X is 3+4` $\rightsquigarrow$ `X=7`
`7 is 3+4` $\rightsquigarrow$ `ja`
`3+4 is 3+4` $\rightsquigarrow$ `nein`
`..., X is 3+Y` $\rightsquigarrow$ `X=3+Y` nur erfolgreich, wenn `Y` durch eine Zahl instantiert

Unterschiede 3 Sichten von Gleichheit

- Wertgleichheit: $t_1 =:= t_2$ Auswertung von t_1 und t_2
- Wertzuweisung: $t_1\ is\ t_2$ Auswertung von t_2 und Unifikation
- Termgleichheit: $t_1 = t_2$ keine Auswertung, nur Unifikation

Beispiel 4.5
`X=3+4, Y is X` $\rightsquigarrow$ `Y=7`
`1+2=2+1` $\rightsquigarrow$ `nein`
`1+2=:=2+1` $\rightsquigarrow$ `ja`
`1+X=Y+1` $\rightsquigarrow$ `X=1, Y=1`
`1+X=:=Y+1` $\rightsquigarrow$ `nein`, falls `X` oder `Y` nicht passend instantiert.

Verwendung

1) `add(X,Y,Z):-Z is X+Y.`
   Vorteil: Effizienz
   Nachteil: Verlust der Bidirektionalität
2) `fact(N,F):-N>0, N`$_1$ `is N-1, fact(N`$_1$`,F`$_1$`), F is N*F`$_1$`.`
   `fact(0,1).`
3) `gcd(X,X,X).`
   `gcd(X,Y,Z):-X<Y, Y`$_1$ `is Y-X, gcd(X,Y`$_1$`,Z).`
   `gcd(X,Y,Z):-Y<X, X`$_1$ `is X-Y, gcd(X`$_1$`,Y,Z).`

4.3 Listen

Definition 4.1
Sei `A` eine Menge. Eine **Liste über** `A` ist eine lineare Liste (Wort) über `A` oder Listen über `A`: `L(A)=(A`$\cup$`L(A))`*. Bezeichnung: `[],[a],[a,b,c],[[a,b],a,[]]`
Beachte: `A`$\cap$`L(A)=`$\emptyset$ (Atome sind keine Listen)

Termdarstellung von Listen
Seien `cons`$\in \Omega^{(2)}$ und `nil`$\in \Omega^{(0)}$. Wir definieren $\tau : \mathtt{L(A)} \to T_\Omega(\mathtt{A})$. τ`([]):= nil.`
τ`([z`$_0$`,...,z`$_n$`]):= if z`$_0$ ``$\in$`A then cons(z`$_0$`,`τ`([z`$_1$`,...,z`$_n$`]))`
`else cons(`τ`(z`$_0$`),`τ`([z`$_1$`,...,z`$_n$`]))`

Beispiel 4.6
```
[]    nil.
[a]    cons(a,nil).
[a,b,c]    cons(a,cons(b,cons(c,nil))).
[[a,b],a,[]]    cons(cons(a,cons(b,nil)),cons(a,cons(nil,nil))).
```

Prolog-Notation
`[,]` wie oben, außerdem: `[z|l]` statt `cons(z,l)`
aber auch: `[a|[b,c]]=[a,b,c]`

Beispiel 4.7

- `member(X,Xs`[4]`)` "`X` ist Element der Liste `Xs`"
  `member(X,[X|_Xs]).`
  `member(X,[_Y|Ys):- member(X,Ys).`
  `_Xs, _Y` "anonyme Variablen", kommen nur einmal in der Klausel vor, Bindung unwichtig

[4] `Xs` steht für den Plural, Konvention

- `append(Xs,Ys,Zs)` "`Zs` ist Verknüpfung der Listen `Xs` und `Ys`"
  `append([],Ys,Ys).`
  `append([X|Xs],Ys,[X|Zs]):- append(Xs,Ys,Zs).`
  Anfragen:
  `?-append([a,b],[c,d],[a,b,c,d]).` $\rightsquigarrow$ `ja`
  `?-append(Xs,Ys,[a,b,c,d]).` $\rightsquigarrow$ `Xs=[], Ys=[a,b,c,d]`
  $\rightsquigarrow$ `Xs=[a], Ys=[b,c,d]` $\rightsquigarrow$ `...`
- `length([],0).`
  `length([_|Xs],N):- length(`X_1`,`N_1`), N is N+1.`
  (_ ist eine anonyme Variable)

4.4 Operatoren

Ziel: Vermeidung starrer Präfix-Syntax durch zusätzliche Infix-, Postfix-Notation mit Präzedenzvereinbarung

- Vorfdefinierte Operatoren: z.B. Arithmetik wie üblich
- Selbstdefinierte Operatoren: bessere Lesbarkeit von Programmen und Ein-/Ausgabe

dazu: **Direktive** (Regel mit leerem Kopf) mit dreistelligem Systemprädikat als Programmklausel `:- op(`Präzedenz, Typ, Name(n)`).`

- Präzedenz: $0 \leq$ int ≤ 1200
- Typ: `xf,yf,fx,fy,xfx,xfy,yfx,yfy` für Post-, Prä- und Infix-Notation
  - `x`: kleinere Präzedenz, nicht assioziativ
  - `y`: kleinere oder gleiche Präzedenz, assioziativ
- Name(n): Name oder Liste von Namen für Op-Symbole

Terme ohne definierte Operatoren haben Präzdenz 0.

Beispiel 4.8

a) Operator sei `-`
`1-2-3` steht für
Typ `xfy`: `-(1,-(2,3))` "rechtsassioziativ"
Typ `yfx`: `-(-(1,2),3))` "linksassioziativ"
Typ `xfx,yfy`: nicht möglich

Operatoren `+,/`:
`12/6+1` steht für
`+(/(12,6),1)` (`/` hat kleinere Präzedenz als `+`)
`/(12,+(6,1)` (`+` hat kleinere Präzedenz als `/`)

b) `:-op(300,xfx,was).`
`:-op(250,xfy,of).`
`:-op(200,fy,the).`
`Laura was the secretary of the department`
interne Darstellung:
`was(Laura,of(the(secretary),the(department))).`
Anfrage:
`?-Who was the secretary of the department` $\rightsquigarrow$ `Who=Laura`
`?-Laura was what` $\rightsquigarrow$ `What=the secretary of the department`

4.5 Ein-/Ausgabe

Bisher: Eingabe nur in Form der Anfrage

- ohne Variablen $\rightsquigarrow$ Antwort: `ja/nein`
- mit Variablen $\rightsquigarrow$ Antwort: Variablenbindung

zusätzlich: Extra-logische Prädikate für Ein-/Ausgabe, Erfüllung mit Seiteneffekt

Ein-/Ausgabe von Termen

- `read(t)` liest eine Term `s` von der Eingabe und unifiziert `s` und `t`. Falls Unifikation nicht möglich, schlägt `read(t)` fehl und es erfolgt Backtracking **ohne** die Alternativen für `read(t)`. Außerdem gibt es kein Zurücksetzen der Eingabe.
  In SWI-Prolog: Wurf einer Exception
  Normale Verwendung: `read(X), ...`
  Schlecht: `p:- read(hallo(X)), ...`
- `write(t)` schreibt auf die Ausgabe. Nicht instantierte Variablen werden durch "_n" (n$\in \mathbb{N}$) ausgegeben.
  Ausgabe von Text: `write('Hello World').`
  Backtracking über die Ausgabe möglich:
  `q(a).`
  `q(b).`
  `p:-q(x),write(x),x=b.`
  `?-p` $\rightsquigarrow$ `a b ja`
- `nl`: Zeilenvorschub auf der Standardausgabe

Weitere Möglichkeiten im SWI-Prolog-Manual.

4.6 Das Cut-Prädikat

Nachteile der DFS-Strategie von Prolog:

- Backtracking im SLD-Baum aufwendig; Oft Alternativen nicht erforderlich
- Unvollständigkeit bei unendlichen Berechnungen

Nachteile durch das Kontrollprädikat "Cut", Bezeichnung: `!`, zum Abschneiden von Teilen des SLD-Baumes.

Syntax: In Klauselrümpfen kann der Cut als Literal auftreten:
`A:-` B_1`,...,`B_i`,!,`B_{i+1}`,...,`B_k.

Semantik: Werden bei Anwendung dieser Regel B_1 bis B_i erfüllt (1. Möglichkeit im SLD-Baum), so müssen zur Erfüllung von `A` auch die $B_{i+1}, \ldots, B_k$ erfüllt werden (Determinismus). Die Alternativen zur Erfüllung von $B_1, \ldots, B_i$ sowie die Alternativen zu `A` werden abgeschnitten (nicht ausprobiert).

Operationell:

- Beim Vorwärtsrechnen (Absteigen im SLD-Baum) wird `!` wie "true" behandelt, ist also immer erfüllt.
- Lassen sich nach Erfüllen des Cut die Teilziele $B_{i+1}, \ldots, B_k$ nicht mehr erfüllen, so werden beim Backtracking alle Cut-Knoten (Knoten mit Teilziel `!`) und der Knoten mit der Anwendung der Cut-Regel übersprungen (Rücksprung zum Großvaterknoten des 1. Cut-Knotens).

Anwendungen:

1) Vermeidung unnötiger Berechnungen
   Beispiel: Mischen zweier sortierter Listen zu einer sortierten Gesamtliste (ganze Zahlen, `=<`, `>`)
   ```
   merge([],Ys,Ys).                                    (1)
   merge(Xs,[],Xs).                                    (2)
   merge([X|Xs],[Y|Ys],[X|Zs]):-X=<Y,merge(Xs,[Y|Ys],Zs).   (3)
   merge([X|Xs],[Y|Ys],[Y|Zs]):-X>Y,merge([Y|Ys],Ys,Zs).    (4)
   ```
   Beachte: Die Teilziele `X=<Y` und `X>Y` sind nicht beide (gleichzeitig) erfüllbar. Aber Backtracking nach Fehlschlagen des zweiten Teilziels in Zeile (3) bzw. (4) führt zur Anwendung bon (4) bzw. (3). Unnötig!
   Verbesserung: Einführen des Cut in (3) und (4)
   ```
   merge([X|Xs],[Y|Ys],[X|Zs]):-X=<Y,!,merge(Xs,[Y|Ys],Zs).   (3')
   merge([X|Xs],[Y|Ys],[Y|Zs]):-X>Y,!,merge([Y|Ys],Ys,Zs).    (4')
   ```
2) Simulation von `if-then-else` (Determinismus)
   Simuliere "`A:-if B then C else D.`" durch:
   ```
   A:-B,!,C.
   A;-D.
   ```
   Nur wenn `B` fehlschlägt, wird zu `D` resolviert.

3) Gefahren beim Cut
`anz_Eltern(adam,0):-!.`
`anz_Eltern(eva,0):-!.`
`anz_Eltern(X,2).`
Anfragen: `?-anz_Eltern(adam,X).` $\rightsquigarrow$ `X=0`
`?-anz_Eltern(john,X).` $\rightsquigarrow$ `X=2`
Aber: `?-anz_Eltern(eva,2)` $\rightsquigarrow$ `ja`
Beachte: Möglichkeiten der Instantiierung berücksichtigen

4) Implementierung der Negation
bisher: Aus Lp nur positive Aussagen ableitbar, aber keine Aussagen der Form $\neg A$. Ist $Cons(Lp)$ vollständig, dann ist entweder $A \in Cons(Lp)$ oder $\neg A \in Cons(Lp)$, woraus folgt:
$\neg A \in Cons(Lp) \iff A \notin Cons(Lp) \iff \square \notin Res^*(Lp \cup \{\neg A\})$
Aber:

   1) Die **Closed-World-Assumption** (d.h. $Cons(Lp)$ vollständig) muß nicht zutreffen.
   2) Es ist im Allgemeinen nicht feststellbar, ob $\square$ nicht resolvierbar.
   Grund: unendliche Berechnungen

   Hilfslösung: **Negation as finite failure**
   $\neg A$ bezüglich Lp **erfolgreich** $\iff$ SLD-Baum zu A bezüglich Lp ist endlich und hat keinen erfogreichen Ast.

   Simulation durch den Cut:
   `not_A:-A,!,fail.` (`fail` ist nicht erfüllbares Systemprädikat)
   `not_A.`
   SWI-Prolog: `\+(A)`

Klassifizierung von Cuts

"green cut":	keine Änderung der deklarativen Semantik, nur nicht-erfolgreiche und nicht-terminierende Berechnungen werden entfernt.
"red cut":	Änderung der deklarativen Semantik, Abschneiden erfolgreicher Berechnungen

4.7 Programmiertechniken

bereits gezeigt: Techniken der imperativen sowie funktionalen Programmierung in Prolog möglich, insbesondere ist Prolog universell
jetzt: neue Techniken der logischen Programmierung

1) Nichtdeterministische (ND) Programmierung
   ND: oft einfacherer Entwurf von Algorithmen

   Beispiel 4.9 (Compilerbau)
   Scanner: Regulärer Ausdruck → NFA
   Parser: CFG → NPDA

   Prolog: nichtdeterministisches Programm, SLD-Baum, deterministische Simulation durch sequentielle Auswertung mit Backtracking (BT)
2) Programmiertechnik: "Generate and Test"
   Erzeuge Lösungsvorschläge und prüfe die Gültigkeit
   `find(X):-generate(X), test(X).`
   erfolgloser Test ↷ BT zu `generate(X)`, nächster Lösungsvorschlag

   Technik oft einfacher als direkte Lösungen, aber oft weniger effizient.

   Beispiel 4.10 (Wortsuche)
   Hilfsprädikat: `member(X,Xs)` zur Erzeugung von Listenelementen (siehe Beispiel 4.7)

```
verb(Sentence,Word):-member(Word,Sentence),verb(Word).
noun(Sentence,Word):-member(Word,Sentence),noun(Word).
article(Sentence,Word):-member(Word,Sentence),article(Word).
 noun(man).  noun(woman).
 article(a).                 } Wörterbuch
 verb(loves).
?- verb([a,man,loves,a,woman],V). ⇝ V=loves
?- article([a,man,loves,a,woman],A). ⇝ A=a
```

   Beispiel 4.11 (Permutation sort) (Top down design)
   `sort(Xs,Ys)` soll gelten, wenn die Liste `Ys` eine geordnete Permutation der `Xs` ist.

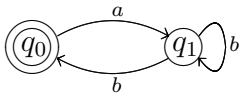
```
sort(Xs,Ys):-permutation(Xs,Ys), ordered(Ys).
permutation(Xs,[Z|Zs]):-select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]):-select(X,Ys,Zs).
ordered([X]).
ordered([X,Y|Ys]):-x≤y, ordered([Y|Ys]).
```

   Aber: sort ist sehr ineffizient, exponentielle Laufzeit

Besser: Tester in den Erzeuger integrieren, `permutation` und `ordered` zu einem rekursiven Prädikat umkonstruieren → **insertion sort**
`sort([X|Xs],Ys):-sort(Xs,Zs), insert(X,Zs,Ys).` (`Zs` geordnet)
`sort([][]).`
`insert(X,[],[X]).`
`insert(X,[Y|Ys],[Y|Zs]):-x>y, insert(X,Ys,Zs).`
`insert(X,[Y|Ys],[X,Y|Ys]):-x≤y.`
Laufzeit: $\mathcal{O}(n^2)$

3) Simulation nicht-deterministischer Berechnungen
Erkennung regulärer Sprachen durch NFA's
$\mathfrak{A} = \langle Q, \Sigma, \delta, q_0, F\rangle \in NFA$ mit $\delta : Q \times \Sigma \to \mathcal{P}(Q)$

Beispiel 4.12 $\mathfrak{A}$ erkennt $(ab^*b)^*$:



Beschreibung von $\mathfrak{A}$ durch Fakten:
`initial(q0).`
`final(q0).`
`delta(q0,a,q1).`
`delta(q1,b,q1).`
`delta(q1,b,q0).`
`accept`-Prädikat zur Worterkennung. Wörter als Buchstabenlisten.
`accept(W):-initial(Q), accept(Q,W).`
`accept(Q,[X|Xs]):-delta(Q,X,Q1), accept(Q1,Xs).`
`accept(Q,[]):-final(Q).`

Beachte: Erkennung eines Wortes benötigt im Allgemeinen Backtracking.

Beispiel 4.13 `abb` ↦ `[a,b,b]`

4) Akkumulatortechnik

Beispiel 4.14 (Spiegelung einer Liste)
`reverse([],[]).`
`reverse([X|Xs],Zs):-reverse(Xs|Ys), append(Ys,[X],Zs).`
Zeit-Aufwand: $\mathcal{O}(n^2)$, wobei n = Länge der Liste
Grund: 2 geschachtelte Rekursionen, beachte: `append` mit $\mathcal{O}(n)$
Verbesserung durch Einführen eines **Akkumulators**:
`reverse(Xs,Ys):-reverse(Xs,[],Ys).` Hier ist `[]` der Akkumulator
`reverse([X|Xs],Reversals,Ys):-reverse(Xs,[X|Reversals],Ys).`
`reverse([],Ys,Ys).`
Zeit-Aufwand: $\mathcal{O}(n)$, weil die Zerlegung der Quellliste und der Aufbau der Zielliste gleichzeitig erfolgen.

Verwandtes Problem: Transformation in Iteration

Beispiel 4.15 (Fakultät)
rekursiv:
`fac(0,1).`
`fac(N,F):-N>0, N1 is N-1, fac(N1,F1), F is N*F1.`
Beachte: Erst `N` ganz herunterzählen, dann kommen die Multiplikationen $\rightsquigarrow$
Platzbedarf: $\mathcal{O}(n)$ (wegen der Zwischenwerte)

iterativ: 2 weitere Argumente als Akkumulatoren für Schleifenvariable und Teilergebnis
`fac(N,F):-fac(0,N,1,F).`
`fac(I,N,T,F):-I<N, I1 is I+1, T1 is T*I1, fac(I1,N,T1,F).`
`fac(N,N,F,F).`
Platzbedarf: $\mathcal{O}(1)$

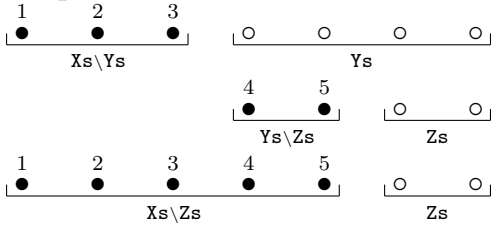
5) Differenzlisten
Alternative Darstellung von Listen, unvollständige Datenstruktur

Ziel: Zugriff auf das Listenende, damit `append` in einem Unifiktionsschritt möglich ist

Darstellung einer Liste als Differenz zweier Listen, z.B.
`[1,2,3]=[1,2,3,4,5]\[4,5]`, allgemeiner `[1,2,3]=[1,2,3|Xs]\Xs`

Beachte: Der binäre Konstruktor \ für Listenpaare erlaubt bei geeigneter Anwendung beliebige Listenzerlegung im Unterschied zu `|`.

Beispiel 4.16 Konkatenation von Differenzlisten



`append_dl(Xs\Ys,Ys\Zs,Xs\Zs).`
`?-append_dl([1,2,3|Ys]\Ys,[4,5|Zs]\Zs,Xs\Zs).`
Unifikation: `Zs=[], Ys=[4,5|Zs], Xs=[1,2,3,4,5|Zs]`
Aufwand: $\mathcal{O}(1)$

Beispiel 4.17 Flachklopfen (flatten) einer geschachtelten Liste zu einer linearen Liste.
`flatten([X|Xs],Ys):-flatten(X,Ys1), flatten(Xs,Ys2),`
`append(Ys1,Ys2,Ys).`
`flatten(X,[X]):-constant(X), X=/=[].`
`flatten([],[]).`
Nachteil: Laufzeit $\mathcal{O}(n^2)$, falls die Liste n Atome enthält
Mit Differenzlisten:
`flatten(X,Ys):-flatten_dl(X,Ys\[]).`

```
flatten_dl([X|Xs],Ys\Zs):-flatten_dl(X,As\Bs),
  flatten_dl(Xs,Cs\Ds), append_dl(As\Bs,Cs\Ds,Ys\Zs).
flatten_dl(X,[X|Xs]\Xs):-constant(X), X=/=[].
flatten_dl([],Xs\Xs).
```

Laufzeit: $\mathcal{O}(n)$ "The flattened list seems to be build by magic."

6) Definite Klauselgrammatiken
Wichtige Anwendung der LP: Syntaxanalyse (Parsing)
DCG's (Definite Clause Grammars): Verallgemeinerte CFG's
— notationelle Varianten einer Klasse von Prolog-Programmen
Verwendung von ND und Differenzlisten
Übersetzung von CFG's in Prolog-Programme:
CFG: $G = \langle N, \sigma, S, P\rangle$
Produktion $P \ni A \to \alpha$ mit $A \in N$, $\alpha \in (\Sigma \cup N)^*$
Abl.-relation: $\beta \overset{G}{\Rightarrow} \gamma \iff \exists A \to \alpha \in P, \beta = \beta_1 A \beta_2, \gamma = \beta_1 \alpha \beta_2$
$L(G) = \{w \in \sigma^* | S \overset{G}{\Rightarrow}^* w\}$
Prolog-Notation für

- Elemente aus N: Konstanten (klein geschriebene Bezeichner)
- Elemente aus Σ: einelementige Listen mit einer Konstanten
- Elemente aus Σ^*: Listen von Konstanten
- Elemente aus $(\Sigma \cup N)^*$: durch Kommata getrennte Sequenzen von Konstanten und Listen von Konstanten

a) Übersetze jede Regel der Grammatik nach folgendem Schema in eine Prolog-Klausel

$A \to w \mid a- \to w \mapsto$ `a(w).` (`w`: Liste von Konstanten)
$A \to Bc \mid a- \to a_1, a_2 \mapsto$ `a(A):-`a_1`(`A_1`),`a_2`(`A_2`),append(`A_1`,`A_2`,A).`
allgemeiner:
$a- \to w_0, a_1, w_1, \ldots, w_n, a_n, w_{n+1} \mapsto$ `a(A):-`a_0`(`A_0`),...,`a_n`(`A_n`),` `append`$_{2n+1}$`(`w_0`,`A_0`,...,`w_{n+1}`,A).`
wobei für $i \geq 3$: `append`$_i$`(`L_1`,...,`L_i`,`L_0`):-append(`L_1`,`L_2`,L'),` `append`$_{i-1}$`(L',`L_3`,...,`L_i`,`L_0`).` (mit `append=append`$_2$)

b) Ordne jedem NT-Symbol ein einstelliges Prädikat zu, welches testet, ob sein Argument mit G ableitbar ist.

Effizienzverbesserung durch Differenzlisten:

Beispiel 4.18 (zur Beispiel-Grammatik, Folie) Ausführung

```
parse([the,man,sings])
└satz([the,man,sings]\[])
  ├nominalphrase([the,man,sings]\VP)
  │ ├artikel([the,man,sings]\N)          ⇝N=[man,sings]
  │ └nomen([man,sings]\VP)                ⇝VP=[sings]
  └verbalphrase([sings]\[])
    └verb([sings])\[])
```

Prolog-Auswertungsstrategie ↷ "Top-Down-Left-to-Right-Parser" recursive descent
ND ↷ Backtracking
SWI-Prolog: unterstützt Grammatik-Regeln direkt, automatische Transformation in Prolog-Klauseln mit Differenzlisten

Erweiterung um Parameterterme zum Aufbau von Syntaxbäumen:

Beispiel 4.19

```
satz(satz(NP,VP),S,S0):-nominalphrase(NP,S,V),
  verbphrase(VP,V,S0).
?-satz(SB,[the,man,sings],[])
```

Diese Anfrage liefert den Syntaxbaum

```
SB=satz(nominalphrase(artikel(the),nomen(man),
  verbphrase(verb(sings)).
```

Außerdem: Überprüfung von Kontextabhängigkeiten ↷ echte Erweiterung der Ausdrucksstärke gegenüber CFG's

5 Datenbanken - Datalog

Prolog als Anfragesprache für relationale Datenbanken (kommerzielle Nutzung)
Nur kleine DB-Systeme im Hauptspeicher, große DB's verteilt auf großen Massenspeichern $\rightsquigarrow$ Kopplung von Prolog an DB-Systeme, z.B. PRO-SQL

Vorteile von Prolog für DB-Systeme:

- große Ausdruckskraft
- rekursive Anfrage - deduktive DB

Nachteile von Prolog für DB-Systeme:

- prozedural (abhängig von der Reihenfolge)
- einzelne Antworten (Auswertungsstrategie)
- Rekursion problematisch bei Kopplung

Alternative: Datalog

- nicht-prozedural: keine Abhängigkeit von der Reihenfolge der Klauseln
- mengenorientiert: Menge *aller* Antworten (Breitensuche)
- Einschränkung von Prolog: *keine Funktionssymbole!* (zum Aufbau komplexer Datenstrukturen)

Grund: Relationale DB *flach* im Gegensatz zu hierarchischer DB

5.1 Syntax von Datalog

Unendliches Alphabet, Signatur $\Sigma = \langle \Omega, \pi, \varrho \rangle$ mit $\Omega = \Omega^{(0)}$ und $\pi = \pi \setminus \pi^{(0)}$.

- $Var = \{X, Person, \ldots\}$ Variablen
- $Const = \{a, peter, 10, \ldots\}$ Konstanten
- $Pred = \{vater, eltern, \ldots\}$ Prädikate
  $\varrho : Pred \to \mathbb{N}^+$ (positive Stelligkeit)
- Term: Konstante oder Variable
- Grundterm: Konstanten = Herbrand-Universum
- Atome: $p(t_1, \ldots, t_n)$ falls $p \in Pred$, $\varrho(p) = n$, t_i Term

Beispiel 5.1

```
vater(peter,heinz)
ahn(jeder,eva)
```

Grundatome: ohne Variablen
Herbrand-Basis HB: alle Grundterme (unendliche Menge)
Literale: $p(t_1, \ldots, t_n)$ oder $\neg p(t_1, \ldots, t_n)$
Klauseln: $\{\neg p(X, a), p(Y, b)\}$
Hornklauseln

Prolog-Notation:

- Fakt: `vater(peter,heinz).`
- Regel: `opa(X):-eltern(X), männlich(X).`
- Anfrage: `?-vater(peter,X).`

Extensionale Datenbanken und Datalog-Programme

$Pred = EPred \cup IPred$ mit $EPred \cap IPred = \emptyset$ (E für extensional und I für intesional)
$E(HB)$ und $I(HB)$: extensionaler bzw. intensionaler Anteil der HB

Definition 5.1
Eine **extensionale DB** (EDB) ist eine endliche Teilmenge von $E(HB)$.

Beispiel 5.2
{`eltern(peter,hans), eltern(peter,maria),...`
`lebt(peter,aachen), lebt(maria,kiel),...`
⋮ }
wobei `eltern,lebt`$\in EPred$.

Annahme: EDB ist in relationaler DB gespeichert.

Definition 5.2
Ein **Datalog-Programm** P ist eine endliche Menge von Hornklauseln C mit den Eigenschaften:

(i) C ist ein Fakt $\curvearrowright C \in EDB$
(ii) C ist eine Regel `p(...):-...` mit $p \in IPred$ und alle Variablen des Kopfes treten im Rumpf auf.

Beachte: "In Fakten keine Variablen" + "Kopf-Variablen treten im Rumpf auf" $\curvearrowright$ endliche Antwortmenge

Beispiel 5.3 (Datalog-Prog.) `adam,lebt`$\in EPred$, `ahn,person`$\in IPred$
```
ahn(X,Y):-eltern(X,Y).
ahn(X,Y):-eltern(X,Z), ahn(Z,Y).
ahn(x,adam):-person(X).
person(X):-lebt(X,Y).
```

Definition 5.3
Eine **Datalog-Anfrage** enthält genau 1 Literal.

Beispiel 5.4 `?-vater(peter,X).`

Bemerkung: Wird die Anfrage weggelassen, so sind alle möglichen Antworten gefragt. Eingabe: EDB Ausgabe: Teilmenge von $I(HB)$

5.2 Semantik von Datalog

Auffassung: Ein Datalog-Programm P enthält als Anfrage eine extensionale Datenbank EDB

Deklarative Semantik: modelltheoretisch
Interpretation/Struktur: $\mathfrak{A} = \langle A; \varphi \rangle$ mit $\varphi(Const) \in A$ und $\varphi(p) \subseteq A^n$
Folgerungsbegriff: $S \models F$ mit Klauselmenge $S \cup \{F\}$ (jedes Modell für S ist auch Modell für F)
Beschränkung auf Herbrand-Interpretationen $\mathfrak{I}$
$A = const$
$\varphi(p) \subseteq Const^n$ falls $\varrho(p) = n$
Darstellung als Teilmenge der HB:
$\mathfrak{I} := \{p(c_1, \ldots, c_n) | (c_1, \ldots, c_n) \in \varphi(p)\}$

Definition 5.4 (Deklarative Semantik eines Datalog-Programms P)

a) Ohne Anfrageklausel:
$\mathcal{D}[\![P]\!] : \mathcal{P}_{fin}(E(HB)) \to \mathcal{P}(I(HB))$
$\mathcal{D}[\![P]\!](D) := \{F | F \in I(HB), P \cup D \models F\}$

b) Mit Anfrageklausel $G = \{\neg p(t_1, \ldots, t_n)\}$ mit $t_i \in Const \cup Var$, $p \in Pred$
$\mathcal{D}[\![P, G]\!] : \mathcal{P}_{fin}(E(HB)) \to \mathcal{P}(HB)$
$\mathcal{D}[\![P, G]\!](D) := \{F | P \cup D \models F, F \in HB, F$ ist Grundinstanz von $G\}$

Definition 5.5 (Folgerungsmenge ("nur Grundfakten"))
Sei S eine Menge von Datalog-Klauseln, also Grundfakten oder Datalog-Regeln. Dann ist die **Folgerungsmenge von** S definiert durch
$Cons(S) := \{F | F \in HB, S \models F\}$.

Folgerung: $\mathcal{D}[\![P]\!](D) = Cons(P \cup D) \cap I(HB)$
Ziel: Berechnung von $Cons(S)$ für endliches S.

Satz 5.1 $Cons(S) = \bigcap \{\mathfrak{I} | \mathfrak{I}$ Herbrand-Modell von $S\}$

Beweis: $F \in Cons(S) \quad \curvearrowright \quad S \models F, F \in HB \quad \curvearrowright \quad F \in \bigcap \ldots$
Umkehrung: $F \in \bigcap \ldots \quad \curvearrowright \quad F \in HB$ und F gilt in jedem Herbrand-Modell von $S \quad \curvearrowright \quad S \models F$, also $F \in Cons(S)$.

Satz 5.2 $Cons(S)$ ist das kleinste Herbrand-Modell von S.

Beweis: Zu zeigen: $Cons(S)$ ist Modell von S, d.h. jede Klausel $C \in S$ gilt in $Cons(S)$.

Fall 1: $C \in HB \quad \curvearrowright \quad C$ gilt in jedem Modell von S, also $C \in Cons(S)$, also gilt C auch in $Cons(S)$.

Fall 2: $C = \mathtt{L_0 :\text{-} L_1 , \ldots , L_n .}$. Sei $\sigma : Var \to Const$ eine Grundsubstitution. Angenommen: $L_1\sigma, \ldots, L_n\sigma \in Cons(S)$, dann gilt für jedes Herbrand-Modell $\mathfrak{I}$ von S: $L_i\sigma \in \mathfrak{I}$ für $i = 1, \ldots, n$, also auch $L_0 \in \mathfrak{I}$. Es folgt: $L_0\sigma \in Cons(S)$, so daß C in $Cons(S)$ gilt.

5.3 Operationelle Semantik (Beweistheorie von Datalog)

Definition 5.6 (Grundinferenz)

1) Sei $R = \mathtt{L_0 :\text{-} L_1 , \ldots , L_n .}$ eine Datalog-Regel und $\overline{F} = (F_1, \ldots, F_n)$ ein n-Tupel von Grundfakten. Wenn eine Grundsubstitution σ mit $F_i = L_i\sigma$ für $i = 1, \ldots, n$ existiert, so sagt man:
$L_0\sigma$ ist **aus** $\overline{F}$ **durch Grundinferenz mit** R **direkt ableitbar**.
Bemerkung: σ und damit $L_0\sigma$ ist durch $\overline{F}$ und R eindeutig bestimmt. Durch Unifikation von $\overline{F}$ und $(L_1, \ldots, L_n)$ kann die Existenz von σ entschieden und σ effektiv bestimmt werden.
2) Sei S eine Menge von Datalog-Klauseln und G ein Grundfakt. Dann heißt G **durch Grundinferenz mit** S **ableitbar**, Bez.: $S \vdash G$, wenn gilt:
   - $G \in S$ oder
   - es gibt $R \in S$ und Grundfakten $F_1, \ldots, F_n$, so daß $S \vdash F_i$ und G ist aus $(F_1, \ldots, F_n)$ durch Grundinferenz mit R direkt ableitbar.

Satz 5.3 (Korrektheit und Vollständigkeit der Grundinferenz) Ist S eine Menge von Datalog-Klauseln und G ein Grundfakt, so gilt:

$$\boxed{S \models G \iff S \vdash G}$$

Korollar 5.1

(i) $Cons(S) = \{G | G \in HB \text{ und } S \vdash G\}$
(ii) Ist S endlich, so kann $Cons(S)$ effektiv bestimmt werden.
(iii) S endlich $\curvearrowright$ $Cons(S)$ endlich (Variablenbedingungen, aus den endlich vielen Fakten und endlich vielen Regeln nur endlich viel ableitbar)

Bem.: Grundinferenz bestimmt "Bottom-up"-Berechnungen, "forward-chaining".

- Start: Fakten von S ($P \cup D$, Datenbank)

- Neue Faten sukzessiv durch Grundinferenz bestimmen.

Fixpunktsemantik eines Datalog-Programms

$\mathcal{P}(HP)$ ist bezüglich $\subseteq$ ein vollständiger Verband. Sei S eine Menge von Datalog-Klauseln, also $S = Facts(S) \cup Rules(S)$. S bestimmt eine Transformation $T_S : \mathcal{P}(HB) \to \mathcal{P}(HB)$. $T_S(W) := W \cup Facts(S) \cup \{G | G \text{ aus } Rules(S) \cup W \text{ direkt ableitbar}\}$. Dann folgt:

- T_S monoton bezüglich $\subseteq$, also ist $fix(T_S) := \bigcup_{i \in \mathbb{N}} T_S^i(\emptyset)$ kleinster Fixpunkt von T_S.
- $T_S(\emptyset) := Facts(S)$
- $T_S^2(\emptyset) := Facts(S) \cup \{G | G \text{ aus } Rules(S) \cup Facts(S) \text{ direkt ableitbar}\}$
- $T_S^3(\emptyset) := T_S^2(\emptyset) \cup \{G | G \text{ aus } Rules(S) \cup T_S^2(\emptyset) \text{ direkt ableitbar}\}$
- $Cons(S) = fix(T_S)$

Problem: Effiziente Berechnung von $\mathcal{D}[\![P]\!](EDB) = \bigcup_{i=0}^{n} T_{(P \cup EDB)}^i(\emptyset)$.
Techniken dafür: "Magic sets", "Counting", Static filtering"
Buch: Ceri, Gottlob, Tanca: LP and DB, Springer 1990

5.4 Backward chaining and Resolution

Datalog-Programm P, extensionale DB EDB, Anfrage: $G =$ `?-p(`t_1`,...,`t_n`).`
Alternative operationelle Behandlung: Resolution

Beachte: Der SLD-Baum von $(P \cup EDB, G)$ kann in der Tiefe beschränkt werden. Grund: Menge der möglichen Teilziele endlich, bestimmt durch die Zahl der Prädikate in $P \cup EDB$, deren Stelligkeit und Zahl ihrer Konstanten.

Alle Lösungen mit beschränkter Tiefensuche sowie Breitensuche berechenbar (Vollständigkeit).

Bemerkung: Für große DB sind effiziente Methoden erforderlich $\rightsquigarrow$ DB-Theorie, Endliche Modelltheorie/Komplexität

Index