

Logikprogrammierung

gehalten von Prof Dr. Jürgen Giesl
im Sommersemester 2006 an der RWTH Aachen

eine studentische Mitschrift von
Florian Heller
florian@heller-web.net

Diese Mitschrift erhebt keinen Anspruch auf Richtigkeit oder Vollständigkeit.
- Think Different -

12. Juli 2006

Inhaltsverzeichnis

1	Einführung	3
1.1	Einsatzgebiet	3
1.2	Übersicht	3
1.2.1	Imperative Sprachen	3
1.2.2	Deklarative Sprachen	3
1.3	Prolog	4
1.4	Fakten und Anfragen	4
1.4.1	Kommentare in Prolog:	5
1.4.2	Ausführung	5
1.4.3	Variablen	5
1.5	Regeln	6
1.6	Aufbau der Vorlesung:	8
2	Grundlagen der Prädikatenlogik	9
2.1	Syntax der Prädikatenlogik	9
2.2	Semantik der Prädikatenlogik	11
3	Resolution	15
3.1	Skolem-Normalform	16
3.2	Herbrand-Strukturen	17
3.3	Grundresolution	21
3.4	Prädikatenlogische Resolution	22
3.5	Einschränkung der Resolution	23
3.5.1	Lineare Resolution	23
3.5.2	Input- und SLD-Resolution	26
4	Logikprogramme	31
4.1	Syntax und Semantik von Logikprogrammen	31
4.1.1	Deklarative Semantik der Logikprogrammierung	33
4.1.2	Prozedurale Semantik der Logikprogrammierung	33
4.1.3	Fixpunkt-Semantik der Logikprogrammierung	36
4.2	Universalität der Logikprogrammierung	40
4.3	Indeterminismus und Auswertungsstrategien	43

5 Die Programmiersprache Prolog	51
5.1 Arithmetik	52
5.2 Listen	55
5.3 Operatoren	57
5.4 Das Cut-Prädikat und Negation	59
5.4.1 Das Cut-Prädikat	59
5.4.2 Meta-Variablen und Negation	66
5.5 Ein- und Ausgabe	70
5.5.1 Ausgabe	70
5.5.2 Eingabe	71
5.5.3 Ein-/Ausgabe mit Files:	72
5.6 Meta-Programmierung	73
5.6.1 Verarbeitung von Termen und atomaren Formeln	73
5.6.2 Verarbeitung von Programmen	76
5.6.3 Meta-Interpreter	79
5.7 Differenzlisten und definite Klauselgrammatiken	81
5.7.1 Differenzlisten	81
5.7.2 Definite Klauselgrammatiken	82

1 Einführung

1.1 Einsatzgebiet

- Rapid Prototyping
- Deduktive Datenbanken
- Künstl. Intelligenz (vor allem Expertensysteme)

Abbildung 1.1: Uebersicht

1.2 Übersicht

1.2.1 Imperative Sprachen

Folge von nacheinander ausgeführten Anweisungen

Prozedurale Sprachen

Variablen, Zuweisungen, Kontrollstrukturen

Objektorientierte Sprachen

Objekte und Klassen
ADT und Vererbung

1.2.2 Deklarative Sprachen

Spezifikation dessen, was berechnet werden soll
Compiler legt fest, wie Berechnung verläuft

Funktionale Sprachen

Keine Seiteneffekte
Rekursion

Logische Sprachen

Regeln zur Definition von Relationen

1.3 Prolog

Bekannteste log. Programmiersprache. z.B. SWI-Prolog (freie Implementierung) (siehe <http://www.swiprolog.org>)

1.4 Fakten und Anfragen

Abbildung 1.2: Fakten und Anfragen

```
weiblich(monika).
weiblich(karin).
weiblich(renate).
weiblich(aline).
weiblich(susanne).
```

```
maennlich(klaus).
maennlich(gerd).
maennlich(peter).
maennlich(dominique).
maennlich(werner).
```

```
verheiratet(klaus,susanne).
verheiratet(werner,monika).
verheiratet(gerd,renate).
```

```
mutterVon(monika,klaus) .
mutterVon(monika,karin).
mutterVon(renate,susanne).
mutterVon(renate,peter).
mutterVon(susanne,aline).
mutterVon(susanne,dominique).
```

```
mensch(X) .
```

↓ = Mutter von, – = verheiratet

Darstellung der Wissensbasis durch Formeln er Prädikatenlogik (Prolog=Programming in Logic)

Formeln eines Logikprogramms: Klauseln

Es gibt 2 Arten von Klauseln

- Fakten: treffen Aussagen über Objekte:

$\underbrace{\textit{weiblich}}_{1 \text{ stlg. Prädikatssymbol}} \quad \underbrace{(\textit{monika})}_{\textit{Objekt}}, \quad \underbrace{\textit{verheiratet}}_{2 \text{ stlg. Prädikatssymbole}} \quad (\textit{werner}, \textit{monika}).$

Prädikatssymbole u. Objekte beginnen mit Kleinbuchstaben.

- Regeln: erlauben aus bekannten Fakten neue Fakten herzuleiten

1.4.1 Kommentare in Prolog:

- %... Zeilenende
- / *... * /

Ausführung eines LP: Stelle Anfragen. Prolog versucht Formeln zu beweisen, indem es das Wissen aus der Wissensbasis verwendet. „Rechnen“ bedeutet in LP: „Beweisen“

? – *maennlich(gerd)*. ... Yes

? – *verheiratet(gerd,monika)*. ... No ← „closed world assumption“: alles was nicht aus der Wissensbasis folgt, ist falsch.

1.4.2 Ausführung

Um Programme ausführen zu können

- Starte Prolog
- `consult(datei).` (datei.pl enthält das Programm)
[datei]
- Stelle Anfragen

1.4.3 Variablen

Variablen beginnen mit Großbuchstaben oder Unterstrich (*X, Y, Z, _G172, ...*)

Variablen im Programm stehen für **alle** möglichen Belegungen (sind allquantifiziert)

mensch(X). $\hat{=}$ „alle sind Menschen“

? – *mensch(gerd)*. ... Yes (folgt aus Wissensbasis, wenn $\{X/gerd\}$ -Substitution verwendet wird)

? – *mensch(5)*. ... Yes

Gleiche Variablen in der gleichen Klausel müssen gleich belegt werden.

Zusätzliches Faktum: *mag(X, X)*.

? – *mag(gerd,gerd)*. Yes

? – *mag(gerd,renate)*. No

Bei Fakt: *mag(X, Y)*.

Variablen in Anfragen:

existenzquantifiziert: Gibt es eine Belegung von *X*, so dass Aussage wahr ist?

? – *mutterVon(X,susanne)*. ← Gibt es eine Belegung von *X*, so dass Aussage wahr ist? „Wer ist Susis Mutter?“ *X = rene*

Antwort nicht nur „yes“ sondern **Antwortsubstitution:** *X = rene*

? – *mutterVon(rene,Y)*. „Welche Kinder hat rene?“

Y = susanne Prolog liefert erste gefundene Lösung

Wenn die Suche abgeschlossen ist, gebe *return* ein, für die Suche nach weiteren Lösungen gebe ; ein.

$Y = peter$

Klauseln in Programmen werden von oben nach unten durchsucht.

$? - mutterVon(X, Y)$

$X = monika, Y = Kevin$

Ein- und Ausgabe ist nicht durch das Programm festgelegt, sondern hängt von der Anfrage ab.

$? - mensch(Y). Y = Z \leftarrow$ neue Variable ($_G172$)

Prolog versucht möglichst allgemeine Anfragen zu finden. Diese Lösungen müssen für alle Instanzierungen der verbleibenden Variablen wahr sein.

Kombination von Anfragen

Abbildung 1.3: Kombination von Fragen

Bsp.: $? - verheiratet(gerd, F), mutterVon(F, susanne)$: Gibt es eine Belegung von F , so dass sowohl $verheiratet(gerd, F)$, als auch $mutterVon(F, susanne)$ wahr ist? „Ist gerd Vater von susanne“?

Vorgehen von Prolog:

Anfragen werden von links nach rechts bearbeitet.

Prolog bearbeitet erst $verheiratet(gerd, F) \rightarrow F = rene$, dann bearbeite $mutterVon(rene, susanne)$

Oma-Bsp. Reihenfolge ist ungünstig, da mehrfaches Rücksetzen (Backtracking) nötig ist, bis die richtige Lösung gefunden wird.

1.5 Regeln

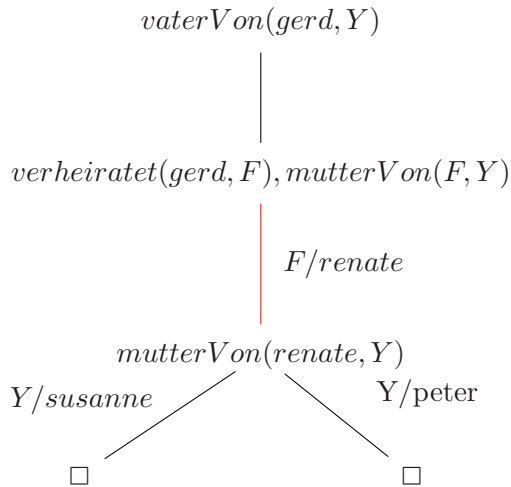
Regeln dienen dazu, um aus bekanntem Wissen, neues Wissen herzuleiten

$\underbrace{p} \quad \underbrace{:-} \quad \underbrace{q, r}$. heißt: p ist wahr falls q und r wahr sind.

Kopf der Regel *falls* Rumpf der Regel

Wenn q und r , dann p .

$vaterVon(gerd, Y) : -verheiratet(gerd, F), mutterVon(F, Y)$



□: Leere Klausel, es ist nichts mehr zu beweisen. $\rightarrow Y = susanne \leftarrow$ Belegung der Variablen aus der Anfrage

Baum entsteht indem man Regelköpfe durch Regelrumpf ersetzt. Fakten $\hat{=}$ Regeln mit leerem Rumpf.

Mehrere Regeln für ein Prädikat

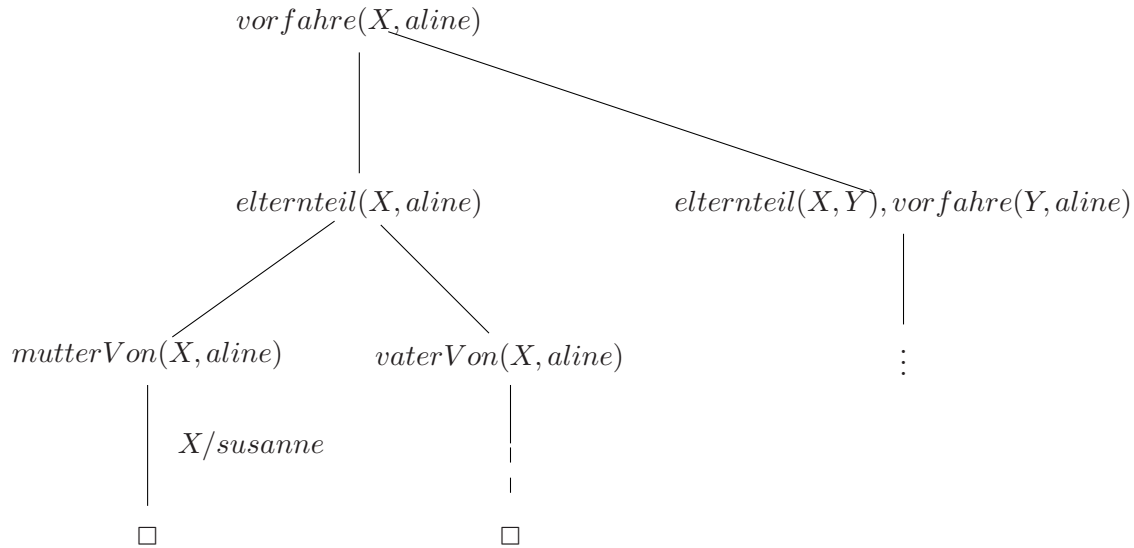
Abbildung 1.4: Mehrere Regeln für ein Prädikat

$elternteil(X, Y)$ soll gelten, falls X Mutter **oder** Vater von Y ist. \Rightarrow realisiert Disjunktion im Programm.

Rekursive Regeln

Abbildung 1.5: Rekursive Regeln

$vorfahre(V, X)$



1.6 Aufbau der Vorlesung:

1. Einführung in LP und Prolog
2. Syntax und Semantik der Prädikatenlogik: Wann folgt aus einer Formelmenge (LP) eine Formel (Anfrage)?
3. Resolution (Technik, um Folgerbarkeit automatisch zu untersuchen).
4. Logikprogrammierung
5. Prolog
6. Anwendungen und Erweiterungen der LP

Webseite: <http://www-i2.informatik.rwth-aachen.de/lufgi2/lp06/>

Skript: (wird z:Zt. erstellt).

Übungen: 2er-Gruppen (elektron. Anmeldung)

Übungsschein: 50% der Punkte + Scheinklausur am Semesterende Am 5.4.06 findet Vorlesung statt. 5.4.06

Logikprogramm = Menge v. Formeln Φ

Anfrage = Formel φ

Ausführung: Untersuche automatisch ob φ aus Φ folgt.

2 Grundlagen der Prädikatenlogik

2.1 Syntax der Prädikatenlogik

Syntax legt fest, aus welchen Worten eine Sprache besteht

Definition 2.1.1 (Signatur)

Eine Signatur (Σ, Δ) ist ein Paar mit $\Sigma = \bigcup_{n \in \mathbb{N}} \Sigma_n$ und $\Delta = \bigcup_{n \in \mathbb{N}} \Delta_n$, wobei Σ_i, Δ_i paarweise disjunkt sind. Jedes $f \in \Sigma_n$ heißt n -stelliges **Funktionssymbol**, jedes $p \in \Delta_n$ heißt n -stelliges **Prädikatensymbol**. Elemente von Σ_0 heißen auch **Konstanten**. Wir verlangen stets $\Sigma_0 \neq \emptyset$

Beispiel 2.1.2 (Σ, Δ) mit $\Sigma = \Sigma_0 \cup \Sigma_3$, $\Delta = \Delta_1 \cup \Delta_2$

Signatur, die dem LP aus Kapitel 1 entspricht:

$$\Sigma_0 = \{\textit{monika}, \textit{karin}, \dots\} \cup \mathbb{N}$$

$$\Sigma_3 = \{\textit{datum}\}$$

$$\Delta_1 = \{\textit{weiblich}, \textit{maennlich}, \textit{mensch}\}$$

$$\Delta_2 = \{\textit{verheiratet}, \textit{mutterVon}, \textit{geboren}, \dots\}$$

Abbildung 2.1: Formelmenge des Logikprogramms

Datenobjekte werden in der Sprache der Prädikatenlogik als **Terme** repräsentiert.

Definition 2.1.3 (Term)

Sei (Σ, Δ) eine Signatur von \mathcal{V} eine Menge von Variablen. Dann bezeichnet $\mathcal{T}(\Sigma, \mathcal{V})$ die Menge aller **Terme** über Σ und \mathcal{V} .

$\mathcal{T}(\Sigma, \mathcal{V})$ ist die kleinste Menge mit

- $\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$
- $f(t_1, \dots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$, falls $f \in \Sigma_n$ und $t_1, \dots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$ für ein $n \in \mathbb{N}$

$\mathcal{T}(\Sigma)$ steht für $\mathcal{T}(\Sigma, \emptyset)$. Menge der **Grundterme**

$\mathcal{V}(t)$ ist die Menge der Variablen in einem Term t .

Konvention: Funktions- und Prädikatssymbole beginnen mit Kleinbuchstaben, Variablen beginnen mit Großbuchstaben.

Beispiel 2.1.4 Σ wie auf Folie, $\mathcal{V} = \{X, Y, Z, \textit{Oma}, \textit{Mama}, \dots\}$

Beispiele für Terme: $X, \textit{monika}, 5, \textit{datum}(15, 10, 1966), \textit{datum}(X, \textit{Oma}, \textit{datum}(15, 10, 1966))$

Aussagen werden in der Sprache der Prädikatenlogik als **Formeln** repräsentiert

Definition 2.1.5 (Formeln)

Sei (Σ, Δ) eine Signatur, \mathcal{V} eine Menge von Variablen. Die **atomaren Formeln** über (Σ, Δ) und \mathcal{V} sind definiert als

$$At(\Sigma, \Delta, \mathcal{V}) = \{p(t_1, \dots, t_n) \mid p \in \Delta_n \text{ für ein } n \in \mathbb{N}, t_1, \dots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})\}$$

$\mathcal{F}(\Sigma, \Delta, \mathcal{V})$ ist die Menge der Formeln. Sie ist die kleinste Menge mit:

- $At(\Sigma, \Delta, \mathcal{V}) \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
- Wenn $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, dann auch $\neg\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
- Wenn $\varphi_1, \varphi_2 \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, dann auch $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \varphi_1 \rightarrow \varphi_2 \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$
- Wenn $X \in \mathcal{V}$ und $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$, dann $\forall X\varphi, \exists\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$

$\mathcal{V}(\varphi)$ ist die Menge aller Variablen in φ . Eine Variable X ist **frei** in φ gdw.

- φ ist atomare Formel und $X \in \mathcal{V}(\varphi)$
- $\varphi = \neg\varphi'$ und X ist frei in φ'
- $\varphi = \varphi_1 \cdot \varphi_2$ mit $\cdot \in \{\wedge, \vee, \rightarrow\}$ und X ist frei in φ_1 oder in φ_2
- $\varphi = QY \varphi'$ mit $Q \in \{\forall, \exists\}$, $X \neq Y$ und X ist frei in φ' .

Eine Formel ist **geschlossen**, wenn sie keine freien Variablen enthält.

Eine Formel ist **quantorfrei**, wenn sie weder \forall noch \exists enthält

Beispiel 2.1.6 Beispiel für Formeln:

$$weiblich(monika) \in At(\Sigma, \Delta, \mathcal{V})$$

$$mutterVon(X, susanne) \in At(\Sigma, \Delta, \mathcal{V})$$

$$geboren(monika, datum(15, 10, 1966)) \in At(\Sigma, \Delta, \mathcal{V})$$

$$\forall F(verheiratet(gerd, F) \wedge mutterVon(F, K)) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$$

freie Variable K

$$verheiratet(gerd, F) \wedge \neg\forall FmutterVon(F, K) \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$$

freie Variable: F, K

Schreibweise: $\forall X_1, \dots, X_n\varphi$ steht für $\forall X_1(\forall(X_2 \dots (\forall X_n\varphi) \dots))$

Analog: $\exists X_1 \dots X_n\varphi$

Beispiel 2.1.7 Jedes Logikprogramm entspricht einer Formelmenge.

- „: –“ $\hat{=}$ \leftarrow
- Variablen sind allquantifiziert.
- „;” $\hat{=}$ \wedge

Abbildung 2.2: Signatur des Logikprogramms

Definition 2.1.8 (Substitution)

Eine Abbildung $\sigma : \mathcal{V} \rightarrow \mathcal{T}(\Sigma, \mathcal{V})$ heißt **Substitution**, falls $\sigma(X) \neq X$ nur für endlich viele $X \in \mathcal{V}$ gilt.

$DOM(\sigma) = \{X \in \mathcal{V} \mid \sigma(X) \neq X\}$ ist der **Domain** von σ . Da $DOM(\sigma)$ endlich ist, ist eine Substitution als die endliche Menge $\{X/\sigma(X) \mid X \in DOM(\sigma)\}$ darstellbar.

σ ist **Grundsubstitution** gdw. $\mathcal{V}(\sigma(X)) = \emptyset$ für alle $X \in DOM(\sigma)$

σ ist eine **Variablenumbenennung** gdw. $\sigma(X) \in \mathcal{V}$ für alle $X \in \mathcal{V}$ und σ ist injektiv.

(Bsp.: $\sigma = \{X/Y, Y/Z, Z/X\}, \sigma(X) = Y, \sigma(U) = U, \dots$)

Substitutionen werden homomorph zu Abbildungen $\sigma : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathcal{T}(\Sigma, \mathcal{V})$ erweitert, d.h. $\sigma(f(t_1, \dots, t_n)) = f(\sigma(t_1), \dots, \sigma(t_n))$

Analog kann man Substitution auch auf Formeln anwenden:

- $\sigma(p(t_1, \dots, t_n)) = p(\sigma(t_1), \dots, \sigma(t_n))$
- $\sigma(\neg\varphi) = \neg\sigma(\varphi)$
- $\sigma(\varphi_1 \cdot \varphi_2) = \sigma(\varphi_1) \cdot \sigma(\varphi_2)$ für $\cdot \in \{\wedge, \vee, \rightarrow\}$
- $\sigma(QX\varphi) = QX\sigma(\varphi)$ für $Q \in \{\forall, \exists\}$, falls $X \notin DOM(\sigma) \cup \mathcal{V} \left(\underbrace{\sigma(DOM(\sigma))}_{\{\sigma(X) \mid X \in DOM(\sigma)\}} \right)$
- $\sigma(QX\varphi) = QX'\sigma(\delta(\varphi))$ für $Q \in \{\forall, \exists\}$ mit $\delta = \{X/X'\}$ sonst.
Hierbei ist X' eine neue Variable mit $X' \notin DOM(\sigma) \cup \mathcal{V}(\sigma(DOM(\sigma)))$

Beispiel 2.1.9 $\sigma = \{X/datum(X, Y, Z), Y/monika, Z/datum(Z, Z, Z)\}$

$\sigma(datum(X, Y, Z)) = datum(datum(X, Y, Z), monika, datum(Z, Z, Z))$

$\left(\begin{array}{l} \forall X \text{ mensch}(X) \\ \forall Y \text{ mensch}(Y) \end{array} \right)$ sollten gleich behandelt werden

$\sigma(\forall Y \text{ verheiratet}(X, Y))$ 1.Problem: $Y \in DOM(\sigma)$

2.Problem; $Y \in \mathcal{V}(\sigma(X))$

$= \forall Y' \text{ verheiratet}(datum(X, Y, Z), Y')$

Schreibweise:

statt $\sigma(datum(X, Y, Z)) : datum(X, Y, Z)[X/datum(X, Y, Z), Y/monika, Z/datum(Z, Z, Z)]$

$\sigma(t)$ ist **Instanz** von t und **Grundinstanz**, falls $\mathcal{V}(\sigma(t)) = \emptyset$

2.2 Semantik der Prädikatenlogik

Definition 2.2.1 (Interpretation)

Für eine Signatur (Σ, Δ) ist eine **Interpretation** ein Tripel $I = (\mathcal{A}, \alpha, \beta)$ mit:

- $\mathcal{A} \neq \emptyset$ ist eine bel. Menge („Träger“)
- α ordnet jedem n -stelligen Funktionssymbol $f \in \Sigma_n$ eine Funktion $\alpha_f : \underbrace{\mathcal{A} \times \dots \times \mathcal{A}}_{n \text{ mal}} \rightarrow \mathcal{A}$ zu,
und jedem $p \in \Delta_n$ eine Menge (Relation) $\alpha_p \subseteq \underbrace{\mathcal{A} \times \dots \times \mathcal{A}}_{n \text{ mal}}$ zu.
 α_f = Deutung des Funktionssymbols f
 α_p = Deutung des Prädikatssymbols p
- $\beta : \mathcal{V} \rightarrow \mathcal{A}$ ist die Variablenbelegung der Interpretation I

Zu jeder Interpretation I erhält man eine Funktion $I : \mathcal{T}(\Sigma, \mathcal{V}) \rightarrow \mathcal{A}$ mit :

$I(X) = \beta(X)$ für alle $X \in \mathcal{V}$

$I(f(t_1, \dots, t_n)) = \alpha_f(I(t_1), \dots, I(t_n))$ für alle $f \in \Sigma_n, t_1, \dots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$

$I(t)$ ist die Interpretation des Terms t .

Für $X \in \mathcal{V}$ und $a \in \mathcal{A}$ ist $\beta[X/a]$ die Variablenbelegung mit $\beta[X/a](X) = a$ und $\beta[X/a](Y) = \beta(Y)$ für alle $Y \in \mathcal{V}$ mit $Y \neq X$

$\mathcal{I}[X/a]$ steht für $(\mathcal{A}, \alpha, \beta[X/a])$ wenn $\mathcal{I} = (\mathcal{A}, \alpha, \beta)$.

Eine Interpretation $\mathcal{I} = (\mathcal{A}, \alpha, \beta)$ **erfüllt** eine Formel $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ (geschrieben $\mathcal{I} \models \varphi$) genau dann wenn

- $\varphi = p(t_1, \dots, t_n)$ und $(\mathcal{I}(t_1), \dots, \mathcal{I}(t_n)) \in \alpha_p$
- $\varphi = \neg\varphi'$ und $\mathcal{I} \not\models \varphi'$
- $\varphi = \varphi_1 \wedge \varphi_2$ und $\mathcal{I} \models \varphi_1$ und $\mathcal{I} \models \varphi_2$
- $\varphi = \varphi_1 \vee \varphi_2$ und $\mathcal{I} \models \varphi_1$ oder $\mathcal{I} \models \varphi_2$
- $\varphi = \varphi_1 \rightarrow \varphi_2$ und falls $\mathcal{I} \models \varphi_1$, dann auch $\mathcal{I} \models \varphi_2$
- $\varphi = \forall X \varphi'$ und $\mathcal{I}[X/a] \models \varphi'$ für alle $a \in \mathcal{A}$
- $\varphi = \exists X \varphi'$ und $\mathcal{I}[X/a] \models \varphi'$ für ein $a \in \mathcal{A}$
- Eine Interpretation ist **Modell** von φ falls $\mathcal{I} \models \varphi$.
 \mathcal{I} ist **Modell** einer Formlemenge Φ ($\mathcal{I} \models \Phi$) gdw. $\mathcal{I} \models \varphi$ für alle $\varphi \in \Phi$
- Zwei Formeln φ_1, φ_2 heißen **äquivalent** gdw. $\mathcal{I} \models \varphi_1$ genau dann, wenn $\mathcal{I} \models \varphi_2$ für alle Interpretationen \mathcal{I}
- Eine Formel(menge) ist **erfüllbar**, falls sie ein Modell hat
Eine Formel(menge) ist **allgemeingültig**, falls jede Interpretation ein Modell ist.
- Eine Interpretation ohne Variablenbelegung $S = (\mathcal{A}, \alpha)$ heißt **Struktur**. Falls wir nur geschlossene Formeln betrachten, kann man „Modell“, „Erfüllbarkeit“ etc. schon mit Strukturen definieren. $S \models \varphi$ mit $S = (\mathcal{A}, \alpha)$ gdw. $\mathcal{I} \models \varphi$ für eine Interpretation $\mathcal{I} = (\mathcal{A}, \alpha, \beta)$ falls φ geschlossen.
Genauso kann man bei Grundtermen t auch $S(t)$ definieren.

Beispiel 2.2.2 Betrachte (Σ, Δ) aus Bsp. 2.1.2.

Eine Interpretation $\mathcal{I} = (\mathcal{A}, \alpha, \beta)$ für diese Signatur ist z.B.:

$$\mathcal{A} = \mathbb{N}$$

$$\alpha_n = n$$

$$\alpha_{monika} = 0$$

$$\alpha_{karin} = 1$$

$$\alpha_{renate} = 2$$

⋮

$$\alpha_{datum}(n_1, n_2, n_3) = n_1 + n_2 + n_3 \text{ für alle } n_1, n_2, n_3 \in \mathbb{N}$$

$$\alpha_{weiblich} = \{n \mid n \text{ ist gerade}\} \quad \alpha_{maennlich} = \{n \mid n \text{ ist ungerade}\}$$

$$\alpha_{mensch} = \mathbb{N}$$

$$\alpha_{verheiratet} = \{(n, m) \mid n > m\}$$

$$\beta(X) = 0, \beta(Y) = 1, \beta(Z) = 2 \dots$$

$$\mathcal{I}(\text{datum}(1, X, karin)) = \alpha_{datum}(\alpha_1, \beta(X), \alpha_{karin}) = 2$$

$$\mathcal{I} \models \text{verheiratet}(\underbrace{\text{datum}(1, X, karin)}_{=2}, \underbrace{karin}_{=1}), \text{ denn } (2, 1) \in \alpha_{verheiratet}$$

$$\mathcal{I} \models \forall X \text{ weiblich}(\text{datum}(X, X, monika)), \text{ denn } \mathcal{I} \llbracket X/a \rrbracket \models \text{weiblich}(\text{datum}(X, X, monika)) \text{ für}$$

$$\text{alle } a \in \mathbb{N} \text{ (} \mathcal{I} \llbracket X/a \rrbracket (\text{datum}(X, X, monika)) = \alpha_{datum}(a, a, 0) = a + a$$

$$a + a \in \alpha_{weiblich} \text{ (ist gerade für alle } a \in \mathbb{N} \text{).}$$

$$S \models (\mathcal{A}, \alpha) \quad S \models \forall X \text{ weiblich}(\text{datum}(X, X, monika))$$

$\varphi \vee \neg\varphi$ ist allgemeingültig

$\varphi \wedge \neg\varphi$ ist unerfüllbar (hat kein Modell)

Syntaktischer Begriff **Substitution**: Abb. von Variablen auf Terme: $\sigma : \mathcal{V} \rightarrow \mathcal{T}(\Sigma, \mathcal{V})$

Semantischer Begriff **Variablenbelegung**: Abb. von Variablen auf Trägerobjekte: $\beta : \mathcal{V} \rightarrow \mathcal{A}$

Lemma 2.2.3 (Substitutionslemma) Sei: $\mathcal{I} = (\mathcal{A}, \alpha, \beta)$ eine Interpretation für (Σ, Δ) , sei $\sigma = \{X_1/t_1, \dots, X_n/t_n\}$ eine Subst.

$$a) \mathcal{I}(\sigma(t)) = \mathcal{I} \llbracket X_1/\mathcal{I}(t_1) \dots X_n/\mathcal{I}(t_n) \rrbracket (t) \text{ f.a. } t \in \mathcal{T}(\Sigma, \mathcal{V})$$

$$b) \mathcal{I} \models \sigma(\text{varphi}) \text{ gdw. } \mathcal{I} \llbracket X_1/\mathcal{I}(t_1), \dots, X_n/\mathcal{I}(t_n) \rrbracket \models \varphi \text{ für alle } \varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$$

Beispiel 2.2.4 Sei \mathcal{I} wie in Bsp. 2.2.2, $\sigma = \{X/\text{datum}(1, X, karin)\}, t = \text{datum}(X, Y, Z)$

$$\mathcal{I}(\sigma(t)) = \mathcal{I}(\text{datum}(\underbrace{\text{datum}(1, X, karin)}_{=2}, \underbrace{Y}_{=1}, \underbrace{Z}_{=2})) = 5$$

$$\mathcal{I} \llbracket X/\mathcal{I}(\text{datum}(1, X, karin)) \rrbracket (t) = \mathcal{I} \llbracket X/2 \rrbracket (\text{datum}(X, Y, Z)) = 5$$

Beweis (von Lemma 2.2.3 a))

durch Strukturelle Induktion über den Aufbau des Terms t :

Induktionsanfang: $t \in \mathcal{V}$ oder $t \in \Sigma_0$

Induktionsschluss: $t = f(s_1, \dots, s_k)$.

Induktionshypothese: Aussage stimmt für s_1, \dots, s_k

$$\begin{aligned}
& \mathbf{1. Fall} \quad t \in \mathcal{V} \text{ Falls } t = X_i, \text{ dann } \mathcal{I}(\sigma(X_i)) = \mathcal{I}(t_i) \\
& \mathcal{I}[\![X_1/\mathcal{I}(t_1), \dots, X_n/\mathcal{I}(t_n)]\!](X_i) = \mathcal{I}(t_i) \\
& \text{Falls } t = Y \notin \{X_1, \dots, X_n\}, \text{ dann } \mathcal{I}(\sigma(Y)) = \mathcal{I}(Y) \\
& \mathcal{I}[\![X_1/\mathcal{I}(t_1), \dots]\!](Y) = \mathcal{I}(Y) \\
& \mathbf{2. Fall} \quad t = f(s_1, \dots, s_k), k \geq 0 \\
& \mathcal{I}(\sigma(t)) = \mathcal{I}(f(\sigma(s_1), \dots, \sigma(s_k))) = \alpha_f(\underbrace{\mathcal{I}(\sigma(s_1))}_{=\mathcal{I}[\![X_1/\mathcal{I}(t_1)]\!](s_1)}, \dots, \underbrace{\mathcal{I}(\sigma(s_k))}_{=\mathcal{I}[\!](s_k)}). \\
& \mathcal{I}[\![X_1/\mathcal{I}(t_1)]\!](t) = \mathcal{I}[\!](f(s_1, \dots, s_k)) = \alpha_f(\mathcal{I}[\!](s_1), \dots, \mathcal{I}[\!](s_k))
\end{aligned}$$

□

Definition 2.2.5 (Folgerbarkeit)

Aus einer Formelmengemenge Φ **folgt** die Formel φ ($\Phi \models \varphi$) gdw. für alle Interpretationen \mathcal{I} mit $\mathcal{I} \models \Phi$ auch $\mathcal{I} \models \varphi$ gilt. Falls Φ, φ keine freien Variablen enthalten, ist dies gleichbedeutend mit $S \models \Phi$ folgt $S \models \varphi$ für alle Strukturen S .

Statt $\emptyset \models \varphi$ schreibt man auch $\models \varphi$. (φ ist allgemeingültig).

Beispiel 2.2.6 Sei Φ die Formelmengemenge aus Bsp. 2.1.7 (entsprechen Logikprogramm aus Kapitel 1).

Die Anfrage: ? – *maennlich(gerd)*. bedeutet, dass man $\Phi \models \text{maennlich(gerd)}$ beweisen muss.

Anfrage: ? – *muttervon(X, susanne)*. bedeutet, dass man $\Phi \models \exists X \text{mutterVon}(X, \text{susanne})$ beweisen muss.

Ziel: wie kann man $\Phi \models \varphi$ automatisch untersuchen?.

3 Resolution

Folgerbarkeit " $\Phi \models \varphi$ " ist semantisch definiert. Zur Untersuchung müsste man alle (∞ viele) Interpretationen betrachten.

Stattdessen: führe **Kalkül** ein (z.B. Resolutionskalkül), der auf syntaktische Weise untersucht, ob man aus Φ eine Formel φ herleiten kann.

Folgerbarkeit $\xleftarrow{\text{Korr. Vollst.}}$ Herleitbarkeit $\xrightarrow{\text{Idee des Res.-Kalküls}}$ Reduziere Folgerbarkeitsproblem auf Unerfüllbarkeitsproblem. 11.04.06

Lemma 3.0.1 (Folgerbarkeit \rightarrow Unerfüllbarkeit) Seien $\varphi_1, \dots, \varphi_k, \varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$. Dann gilt $\{\varphi_1, \dots, \varphi_k\} \models \varphi$ gdw. $\varphi_1 \wedge \dots \wedge \varphi_k \wedge \neg\varphi$ unerfüllbar ist.

Beweis

$\{\varphi_1, \dots, \varphi_k\} \models \varphi$

\rightsquigarrow f. a. Interp. \mathcal{I} mit $\mathcal{I} \models \{\varphi_1, \dots, \varphi_k\}$ gilt $\mathcal{I} \models \varphi$

\rightsquigarrow es gibt keine Interp. mit $\mathcal{I} \models \{\varphi_1, \dots, \varphi_k\}$ und $\mathcal{I} \models \neg\varphi$

$\rightsquigarrow \varphi_1 \wedge \dots \wedge \varphi_k \wedge \neg\varphi$ ist unerfüllbar.

□

Beispiel 3.0.2 LP enthält Faktum $mutterVon(renate, susanne)$.

Anfrage: ? – $mutterVon(X, susanne)$.

z.z. $\{muttervon(renate, susanne)\} \models \exists X\ mutterVon(X, susanne)$.

Stattdessen zeige Unerfüllbarkeit von $mutterVon(renate, susanne) \wedge \neg\exists\ mutterVon(X, susanne)$.

Nachweis der Folgerbarkeit/Unerfüllbarkeit ist **unerfüllbar** Es ex. kein automatisches Verfahren das immer terminiert und Folgerbarkeit/Unentscheidbarkeit entscheidet. Folgerbarkeit / Unentscheidbarkeit sind aber **semi-entscheidbar**: es ex. ein automatisches Verfahren, das terminiert u Folgerbarkeit/Unentscheidbarkeit nachweist, falls Folgerbarkeit/ Unentscheidbarkeit gilt. Aber wenn Folgerbarkeit/Unentscheidbarkeit nicht gilt, dann terminiert das Verfahren evtl nicht. Durch Automatisierung des Resolutionskalküls könnte man z.B. solch ein Semi-Entscheidungsverfahren erhalten.

Einführung des Resolutionskalküls in 5 Schritten:

3.1 Überführe Formeln in Skolem-Normalform: $\forall X_1, \dots, X_n \varphi$
 φ quantorfrei, $\mathcal{V}(\varphi) \subseteq \{X_1, \dots, X_n\}$

3.2 Zeige, dass man sich bei Formeln in Skolem-Normalform auf Herbrand-Interpretationen beschränken kann. \Rightarrow 1. ineffizientes Verfahren zur Unerfüllbarkeitsüberprüfung.

3.3 Aussagenlogische Resolution \Rightarrow 2. Verfahren

3.4 Prädikatenlogische Resolution \Rightarrow 3. Verfahren

3.5 Einschränkungen der Resolution \Rightarrow 4. Verfahren

3.1 Skolem-Normalform

- 2 Schritte: 1. Überführung von Formeln in Pränex-Normalform
2. Weitere Überführung in Skolem-Normalform

Definition 3.1.1 (Pränex-Normalform)

Eine Formel φ ist in **Pränex-Normalform** gdw. sie die Gestalt $Q_1X_1 \dots Q_nX_n\psi$ mit $Q_i \in \{\forall, \exists\}$ und ψ quantorfrei hat.

Satz 3.1.2 (Überführung in Pränex-Normalform)

Zu jeder Formel φ lässt sich automatisch eine äquivalente Formel φ' in Pränex-Normalform konstruieren.

Beweis

Ersetze zunächst alle Teilformeln $\varphi_1 \rightarrow \varphi_2$ durch $\neg\varphi_1 \vee \varphi_2$. Danach wende Algorithmus PRAENEX an.

PRAENEX

Eingabe: Formel φ ohne \rightarrow

Ausgabe: zu φ äquivalente Formel in Pränex-Normalform

- Falls φ quantorfrei ist, dann liefere φ zurück
- Falls $\varphi = \neg\varphi_1$, so berechne $PRAENEX(\varphi_1) = Q_1X_1 \dots Q_nX_n\psi_1$
Liefere $\bar{Q}_1X_1 \dots \bar{Q}_nX_n\neg\psi_1$ zurück, wobei $\bar{\forall} = \exists, \bar{\exists} = \forall$
- Falls $\varphi = \varphi_1 \cdot \varphi_2$ mit $\cdot \in \{\wedge, \vee\}$, so berechne $PRAENEX(\varphi_1) = Q_1X_1, \dots, Q_nX_n\psi_1$ und $PRAENEX(\varphi_2) = R_1Y_1 \dots R_mY_m\psi_2$. Durch Umbenennung gebundener Variablen erreichen wir, dass X_1, \dots, X_n nicht in $R_1Y_1 \dots R_mY_m\psi_2$ auftreten und das Y_1, \dots, Y_m nicht in $Q_1X_1 \dots Q_nX_n\psi_1$ auftreten.
Liefere $Q_1X_1 \dots Q_nX_nR_1Y_1 \dots R_mY_m(\psi_1 \cdot \psi_2)$ zurück.
- Falls $\varphi = QX\varphi_1$ mit $Q \in \{\forall, \exists\}$, so berechne $PRAENEX(\varphi_1) = Q_1X_1 \dots Q_nX_n\psi_1$
Durch Umbenennung gebundener Variablen erreichen wir, dass X_1, \dots, X_n verschieden von X sind.
Liefere $QXQ_1X_1 \dots Q_nX_n\varphi_1$ zurück.

□

Beispiel 3.1.3 $\neg\exists X (\text{verheiratet}(X, Y) \vee \underbrace{\neg\exists Y \text{mutterVon}(X, Y)}_{\forall Y \neg\text{mutterVon}(X, Y)}) = \forall X \exists Z \neg\text{verheiratet}(X, Y) \vee$

$$\underbrace{\forall Z \neg\text{mutterVon}(X, Z)}_{\forall Z \neg\text{mutterVon}(X, Z)}$$

$$\neg\exists X \forall Z (\text{verheiratet}(X, Y) \vee \neg\text{mutterVon}(X, Z))$$

$\text{mutterVon}(X, Z)$)

Beispiel 3.1.4 (Fortsetzung von Bsp. 3.0.2) $PRAENEX(\text{mutterVon}(\text{renate}, \text{susanne}) \wedge \neg\exists X \text{mutterVon}(X, \text{susanne})) = \forall X \text{mutterVon}(\text{renate}, \text{susanne}) \wedge \neg\text{mutterVon}(x, \text{susanne})$

Ziel: φ in Skolem-Normalform

- Eliminiere freie Variablen
- Eliminiere \exists

Definition 3.1.5 (Skolem-Normalform)

Eine Formel φ ist in Skolem-Normalform gdw. sie geschlossen ist und die Gestalt $\forall X_1, \dots, X_n \psi$ mit ψ quantorfrei hat.

Existiert zu jeder Formel φ eine äquivalente Formel in Skolem-Normalform?

weiblich(X), $\exists X$ weiblich(X) Nein, zu diesen Formeln ex. keine äquivalenten Formeln in Skolem-NF.

Es existiert aber zu jeder Formel φ eine **erfüllbarkeitsäquivalente** Formel in Skolem-Normalform.

Satz 3.1.6 (Überführung in Skolem-Normalform)

Zu jeder Formel φ lässt sich automatisch eine Formel φ' in Skolem-Normalform konstruieren so dass φ erfüllbar gdw. φ' erfüllbar.

Beweis

Zunächst wird φ mit dem verfahren aus Satz 3.1.2 in Pränex-Normalform überführt: ergibt φ_1

Seien X_1, \dots, X_n die freien Variablen von φ_1

Dann wird φ_1 überführt in $\varphi_2: \exists X_1, \dots, X_n \varphi_1$

φ_1 und φ_2 sind erfüllbarkeitsäquivalent. (Übung)

Danach werden die Existenzquantoren schrittweise von außen nach innen beseitigt. Falls $\varphi_2 = \forall X_1, \dots, X_n \exists Y \psi$ ($n \geq 0$, φ in Pränex-NF, kann noch Quantoren haben).

dann ersetze φ_2 durch $\forall X_1, \dots, X_n \psi[Y/f(X_1, \dots, X_n)]$ wobei f ein n -stelliges Funktionssymbol ist.

Es gilt: φ_2 und φ_2' sind erfüllbarkeitsäquivalent (Übung, verwende Subst.-Lemma)

Durch mehrfache Wiederholung dieser Technik, werden alle Existenz-quantoren beseitigt.

□

Beispiel 3.1.7 (Fortsetzung von bsp. 3.1.3)

Pränex-NF: $\forall X \exists Z \neg(\text{verheiratet}(X, Y) \vee \neg \text{mutterVon}(X, Z))$

$\Rightarrow \exists Y \forall X \exists Z \neg(\text{verheiratet}(X, Y) \vee \neg \text{mutterVon}(X, Z))$

$\Rightarrow \forall X \exists Z \neg(\text{verheiratet}(X, a) \vee \neg \text{mutterVon}(X, Z))$

$\Rightarrow \forall X \neg(\text{verheiratet}(X, a) \vee \neg \text{mutterVon}(X, f(X)))$

neue Funktionssymbole: a (0-stellig), f (1-stellig).

3.2 Herbrand-Strukturen

21.4.06

Bisher: um Unerfüllbarkeit einer geschlossenen Formel zu zeigen, muss man alle (unendlich viele) Strukturen durchprobieren.

3 Freiheitsgrade:

- Träger \mathcal{A}

- Deutung der Funktionssymbole α_f für alle $f \in \Sigma$
- Deutung der Prädikatssymbole α_p für alle $p \in \Delta$

Wir zeigen: bei Formeln in Skolem-Normalform kann man die ersten beiden Freiheitsgrade festlegen.
 \rightarrow die Suche nach Struktur die die Formel erfüllt wird wesentlich eingeschränkt.

\Rightarrow Einschränkung auf **Herbrand-Strukturen**

Definition 3.2.1 (Herbrand-Struktur)

Sei (Σ, Δ) eine Signatur. Eine Herbrand-Struktur für (Σ, Δ) hat die Gestalt $(\underbrace{\mathcal{T}(\Sigma)}_{\text{Träger, Grundterme}}, \alpha)$,

wobei für alle $f \in \Sigma_n$ mit $n \in \mathbb{N}$ gilt: $\alpha_f(t_1, \dots, t_n) = f(t_1, \dots, t_n)$.

Falls eine Herbrand-Struktur Modell einer Formel φ ist, bezeichnet man sie als Herbrand-Modell von φ .

Herbrand-Strukturen interpretieren Grundterme als "sich selbst": $S(t) = t$ für alle $t \in \mathcal{T}(\Sigma)$.

Die Deutung der Prädikatssymbole kann frei gewählt werden.

Beispiel 3.2.2 Sei die Signatur des Logikprogramms wie folgt:

$\Sigma_0 = \mathbb{N} \cup \{\text{monika, karin, rene, susanne, aline, werner, klaus, gerd, peter, dominique}\}$

$\Sigma_3 = \text{datum}$

$\Delta_1 = \{\text{weiblich, maennlich, mensch}\}$

$\Delta_2 = \{\text{verheiratet, mutterVon, vaterVon, elternteil, vor fahre, geboren}\}$

Herbrand-Struktur $S = (\mathcal{T}(\Sigma), \alpha)$ für diese Signatur:

$\alpha_n = n$ für alle $n \in \mathbb{N}$

$\alpha_{\text{monika}} = \text{monika}$

$\alpha_{\text{datum}}(t_1, t_2, t_3) = \text{datum}(t_1, t_2, t_3)$

$\alpha_{\text{weiblich}} = \{\text{monika, karin, ...}\}$

$\alpha_{\text{mensch}} = \mathcal{T}(\Sigma)$

$\alpha_{\text{geboren}} = \{(\text{monika, datum}(25, 7, 1972)), (\text{werner, datum}(12, 7, 1969))\}$

Zur Untersuchung der Unerfüllbarkeit kann man sich auf Herbrand-Strukturen beschränken

Satz 3.2.3 (Erfüllbarkeit durch Herbrand-Strukturen)

Sei $\Phi \subseteq \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ eine Menge von Formeln in Skolem-Normalform. Dann ist Φ erfüllbar gdw. sie ein Herbrand-Modell besitzt.

Beweis

\Leftarrow *Trivial*

\Rightarrow Sei $S = (\mathcal{A}, \alpha)$ ein Modell von Φ

Wir zeigen, dass die Herbrand-Struktur $S' = (\mathcal{T}(\Sigma), \alpha')$, dann auch Modell von Φ ist. Für alle $f \in \Sigma_n$ gilt $\alpha'_f(t_1, \dots, t_n) = f(t_1, \dots, t_n)$

Für alle $p \in \Delta_n$ definieren wir:

$\underbrace{(t_1, \dots, t_n)}_{S' \models p(t_1, \dots, t_n)} \in \alpha'_p$ gdw. $\underbrace{(S(t_1), \dots, S(t_n))}_{S \models p(t_1, \dots, t_n)} \in \alpha_p$

Wir zeigen, dass für jede Formel $\underbrace{\varphi}_{\forall X_1 \dots X_n \psi, \psi \text{ quantorfrei}}$ in Skolem-Normalform gilt: $S \models \varphi \rightarrow$

$S' \models \varphi$

Induktion über die Anzahl n der allquantifizierten Variablen.

Induktions-Anfang: $n = 0 \rightarrow \varphi$ ist quantorfrei. Hier gilt sogar $S \models \varphi$ gdw. $S' \models \varphi$ (leichte Induktion über den Formelaufbau)

Induktionsschluss $n \geq 0 \forall X_1, \dots, X_{n-1} \psi$ enthält evtl. die freie Variable X_n , ist also nicht in Skolem-Normalform.

$S \models \forall X_1, \dots, X_n \psi$ gdw. $S[[X_n/a]] \models \forall X_1, \dots, X_{n-1} \psi$ für alle $\alpha \in \mathcal{A}$.

$S[[X_n/a]]$ ist eine Interpretation $(\mathcal{A}\alpha, \beta[[X_n/a]])$, Hierbei ist β beliebig.

$\rightarrow S[[X_n/S(t)]] \models \forall X_1, \dots, X_{n-1} \psi$ für alle Grundterme $t \in \mathcal{T}(\Sigma)$

gdw. $S \models \forall X_1, \dots, X_{n-1} \psi[X_n/t]$ (Subst.-Lemma 2.2.3)

$\rightarrow S' \models \forall X_1, \dots, X_{n-1} \psi[X_n/t]$ (Induktions-Hyp.)

gdw. $S'[[X_n/S'(t)]] \models \forall X_1, \dots, X_{n-1} \psi$ (Subst.-Lemma 2.2.3) für alle $t \in \mathcal{T}(\Sigma)$

gdw. $S'[[X_n/t]] \models \forall X_1, X_{n-1} \psi$ für alle $t \in \mathcal{T}(\Sigma)$

gdw. $S' \models \forall X_1, \dots, X_n \psi$

□

Beispiel 3.2.4 Satz 3.2.3 gilt nur für Formeln in Skolem-Normalform. Gegenbeispiel: (Σ, Δ) mit $\Sigma = \Sigma_0 = \{a\}$ und $\Delta = \Delta_1 = \{p\}$. Formelmenge $\{p(a), \exists X \neg p(X)\}$ ist erfüllbar durch Struktur $(\{0, 1\}, \alpha)$ mit $\alpha_a = 0$ und $\alpha_p = \{0\}$.

Grund: der träger enthält ein Objekt 1, das nicht als Deutung von Grundtermen erreicht werden kann. Herbrand-Strukturen $(\{a\}, \alpha')$ mit $\alpha'_a = a$ und $\alpha'_p = \{a\}$ oder $\alpha'_p = \emptyset$

Reduziere das Unerfüllbarkeitsproblem weiter auf Unerfüllbarkeitsuntersuchung einer (unendlichen) Menge von Formeln ohne Variablen. \rightarrow entspricht der Überprüfung der Unerfüllbarkeit für eine (unendliche) Menge **aussagenlogischer** Formeln. Da Herbrand-Strukturen zu Untersuchung der Unerfüllbarkeit ausreichen, ersetze allquantifizierte Variablen durch **alle möglichen Grundterme**.

Definition 3.2.5 (Herbrand-Expansion einer Formel)

Sei $\varphi \in \mathcal{F}(\Sigma, \Delta, \mathcal{V})$ in Skolem-Normalform, mit $\varphi = \forall X_1, \dots, X_n \psi$ mit ψ quantorfrei.

Die folgende Formelmenge $E(\varphi)$ heißt **Herbrand-Expansion** von φ :

$E(\varphi) = \{\psi[X_1/t_1, \dots, X_n/t_n] \mid t_1, \dots, t_n \in \mathcal{T}(\Sigma)\}$.

($E(\varphi)$ ist die Menge aller Grundterme von ψ).

Beispiel 3.2.6 $\varphi = \forall X(\text{mutterVon}(\text{renate}, \text{susanne}) \wedge \neg \text{mutterVon}(X, \text{susanne}))$.

$E(\varphi) = \{\text{mutterVon}(\text{renate}, \text{susanne}) \wedge (\text{karin}, \text{susanne}),$
 $\text{mutterVon}(\text{renate}, \text{susanne}) \wedge \neg \text{mutterVon}(\text{renate}, \text{susanne}) *$
 $\text{mutterVon}(\text{renate}, \text{susanne}) \wedge \neg \text{mutterVon}(\text{datum}(\text{karin}, \text{werner}, 1972), \text{susanne}), \dots \}$

$E(\varphi)$ ist offensichtlich unerfüllbar (enth. Formel *).

Satz 3.2.7

Sei φ in Skolem-Normalform.

Dann gilt: φ ist erfüllbar gdw. $E(\varphi)$ ist erfüllbar.

Beweis

$\forall X_1, \dots, X_n \psi$ erfüllbar

gdw. $S \models \forall X_1, \dots, X_n \psi$ für eine Herbrand-Struktur S (Satz 3.2.3)

gdw. $S \models \psi[X_1/t_1, \dots, X_n/t_n]$ für alle $t_1, \dots, t_n \in \mathcal{T}(\Sigma)$

gdw. $S \models \psi[X_1/t_1, \dots, X_n/t_n]$ für alle $t_1, \dots, t_n \in \mathcal{T}(\Sigma)$ (Subst. Lemma)

gdw. $S \models E(\varphi)$

gdw. $E(\varphi)$ ist erfüllbar (Satz 3.2.3)

□

Prädikatenlogische Formeln ohne Variablen entsprechen aussagenlogischen Formeln. Man kann jede atomare Teilformel $p(t_1, \dots, t_n)$ als aussagenlogische Variable ansehen, die wahr oder falsch sein kann.

→ in Beispiel 3.2.6 statt

$mutterVon(renate, susanne) : V_{mutterVon(renate, susanne)}$

$mutterVon(karin, susanne) : V_{mutterVon(karin, susanne)}$

Zur Unerfüllbarkeit von $E(\varphi)$, Zeige die Unerfüllbarkeit der folgenden aussagenlogischen Formelmengen:

$\{V_{mutterVon(renate, susanne)} \wedge \neg V_{mutterVon(renate, susanne)},$
 $V_{mutterVon(renate, susanne)} \wedge \neg V_{mutterVon(renate, susanne)}, \dots \}$

Algorithmus von Gilmore: Ziel: Überprüfe $\{\varphi_1, \dots, \varphi_k\} \models \varphi^*$

1. Sei ψ die Formel $\varphi_1 \wedge \dots \wedge \varphi_k \wedge \neg \varphi$ (Lemma 3.0.1)
2. Überführe φ in Skolem-Normalform (Satz 3.1.2 und 3.2.6)
3. Wähle Aufzählung $\{\psi_1, \psi_2, \dots\} = E(\psi)$
Ersetze dabei alle atomaren Formeln durch aussagenlogische Variablen
4. Prüfe ob $\psi_1, \psi_1 \wedge \psi_2, \psi_1 \wedge \psi_2 \wedge \psi_3, \dots$ aussagenlogisch erfüllbar sind, Falls eine nicht erfüllbar ist, breche ab und gib "Yes" zurück.

$E(\psi)$ ist unendlich und variablenfrei.

25.4.06

Kompaktheitssatz (der Aussagenlogik): Wenn eine unendliche Formelmenge unerfüllbar ist, dann hat sie auch eine **endliche** unerfüllbare Teilmenge.

Der Algorithmus von Gilmore ist Semi-Entscheidungsverfahren (jede unendliche erfüllbare aussagenlogische Formelmenge hat eine endl. unerfüllbare Teilmenge (Kompaktheitssatz der Aussagenlogik.))

Nachteil: Sehr ineffizient; terminiert nie, falls $\{\varphi_1, \dots, \varphi_k\} \not\models \varphi$

3.3 Grundresolution

Um $\forall X_1, \dots, X_n \psi$ (in Skolem-NF) mit Resolution auf Unerfüllbarkeit zu untersuchen, muss ψ zunächst in **konjunktive Normalform (KNF)** überführt werden \rightarrow Darstellung als **Klauselmengen**.

Definition 3.3.1 (KNF, Literal, Klausel)

Eine Formel ψ ist in KNF gdw. sie quantorfrei ist und folgende Gestalt hat:

$$(L_{1,1} \vee \dots \vee L_{1,n_1}) \wedge \dots \wedge (L_{m,1} \vee \dots \vee L_{m,n_m}).$$

$L_{i,j}$ sind **Literale**, d.h. atomare oder negierte atomare Formeln. (dh. $p(t_1, \dots, t_n)$ oder $\neg p(t_1, \dots, t_n)$).

Zu einem Literal L def. wir sein **Negat** \bar{L} als: $\bar{L} = \begin{cases} \neg A & \text{falls } L = A \in \mathcal{At}(\Sigma, \Delta, \mathcal{V}) \\ A & \text{falls } L = \neg A \in \mathcal{At}(\Sigma, \Delta, \mathcal{V}) \end{cases}$

Eine Menge von Literalen heißt **Klausel**.

Jede Formel ψ in KNF wie oben entspricht der zugehörigen Klauselmenge $\mathcal{K}(\psi)$:

$$\mathcal{K}(\psi) = \{\{L_{1,1}, \dots, L_{1,n_1}\}, \dots, \{L_{m,1}, \dots, L_{m,n_m}\}\}.$$

Eine Klausel steht für die **allquantifizierte Disjunktion** ihrer Literale ($\{L_{1,1}, \dots, L_{1,n_1}\} \hat{=} \forall (L_{1,1} \vee \dots \vee L_{1,n_1})$).

Eine Klauselmenge steht für die **Konjunktion** ihrer Klauseln. \Rightarrow "Folgerbarkeit", "Erfüllbarkeit" ist auch für Klauselmengen definiert.

Wir betrachten meist nur **endliche** Klauselmengen.

Die leere Klausel wird als " \square " geschrieben.

\square ist nach Definition unerfüllbar (leere Disjunktion).

Satz 3.3.2 (Überführung in KNF)

Zu jeder quantorfreien Forel ψ ex. eine äquivalente Formel ψ' in KNF, die man automatisch konstruieren kann.

Beweis

Ersetze zunächst alle Teilformeln $\psi_1 \rightarrow \psi_2$ durch $\neg\psi_1 \vee \psi_2$.

Überführung der verbleibenden Formeln ψ durch den Algorithmus KNF:

- Falls ψ atomar ist, dann gib ψ zurück.
- Falls $\psi = \psi_1 \wedge \psi_2$, dann gib $KNF(\psi_1) \wedge KNF(\psi_2)$ zurück.
- Falls $\psi = \psi_1 \vee \psi_2$, dann berechne $KNF(\psi_1) = \bigwedge_{i \in \{1, \dots, m_1\}} \psi'_i$ und $KNF(\psi_2) = \bigwedge_{j \in \{1, \dots, m_2\}} \psi''_j$
 $(\psi'_1 \wedge \psi'_2) \vee (\psi''_1 \wedge \psi''_2) = ((\psi'_1 \vee \psi''_1) \wedge (\psi'_1 \vee \psi''_2) \wedge (\psi'_2 \vee \psi''_1) \wedge \dots)$
 Gib $\bigwedge_{i \in \{1, \dots, m_1\}, j \in \{1, \dots, m_2\}} \psi'_i \vee \psi''_j$ zurück.
- Falls $\psi = \neg\psi_1$, dann berechne $KNF(\psi_1) = \bigwedge_{i \in \{1, \dots, m\}} (\bigvee_{j \in \{1, \dots, n_i\}} L_{i,j})$
 DeMorgan-Regel liefert: $\bigvee_{i \in \{1, \dots, m\}} (\bigwedge_{j \in \{1, \dots, n_i\}} \bar{L}_{i,j})$
 Gib $\bigwedge_{j_1 \in \{1, \dots, n_1\}, \dots, j_n \in \{1, \dots, n_m\}} \bar{L}_{1,j_1} \vee \dots \vee \bar{L}_{m,j_n}$ zurück.

□

Beispiel 3.3.3 Sei $\Delta_0 = \{p, q, v\}$

Formel: $\neg(\neg p \wedge (\neg q \vee r))$

DeMorgan: $p \vee \neg(\neg q \vee r)$

DeMorgan: $p \vee (q \wedge \neg r)$

Distr. $(p \vee q) \wedge (p \vee \neg r)$

Klauselmenge: $\{\{p, q\}, \{p, \neg r\}\}$

Ziel: Nachweis der Unerfüllbarkeit einer Klauselmenge (Besser als der Algorithmus von Gilmore).

Definition 3.3.4 (Aussagenlogische Resolution)

Seien K_1, K_2 variablenfreie Klauseln. Dann ist die Klausel R **Resolvent** von K_1 und K_2 gdw. es ein Literal $L \in K_1$

Dieser Teil wird aufgrund eines elektriker-Wurms im Apfel nachgereicht

3.4 Prädikatenlogische Resolution

2.5.06

$\{\{p(X), \neg q(X)\}, \{\neg p(f(Y))\}, \{q(f(a))\}\}$

Verwende Substitution $\{X/f(Y)\}$ zur Resolution der ersten beiden Klauseln

$p(X)[X/f(Y)] = p(f(Y))$ und $\neg p(f(Y))[X/f(Y)] = \neg p(f(Y))$

$\{X/f(Y)\}$ ist **allgemeinster Unifikator** von $\{p(X), p(f(Y))\}$

Resolvent ist $\{\neg q(X)[X/f(Y)]\} = \{q(f(Y))\}$

$\sigma = mgu(\{\neg p(f(X)), \neg(p(Z), \neg p(U))\})$

$R = \sigma(\{\neg q(Z), r(g(U))\})$

Ziel: Klauselmenge unerfüllbar gdw. $\square \in Res * (\mathcal{K})$

Bisher: Resolution in der Aussagenlogik ist Korrekt und Vollständig (Satz 3.3.7)

Jetzt: Korrektheit der prädikatenlogischen Resolution: Beweis analog zu Aussagenlogik

Vollständigkeit der prädikatenlogischen Resolution: Führe prädikatenlogische Resolution zurück auf di

Lemma 3.4.1 (Prädikatenlogisches Resolutionslemma) Sei \mathcal{K} eine Menge von Klauseln $K_1, K_2 \in \mathcal{K}$ und R Resolvent von K_1 und K_2 . Dann sind \mathcal{K} und $\mathcal{K} \cup \{R\}$ äquivalent.

Beweis

$\mathcal{K} \cup \{R\} \models \mathcal{K}$ (trivial)

Zu zeigen: $\mathcal{K} \models \mathcal{K} \cup \{R\}$, d.h. $\mathcal{K} \models \{R\}$

Sei $S \models \mathcal{K}$, zu zeigen: $S \models R$

$\nu_1(K_1) = \{L_1, \dots, L_m, L_{m+1}, \dots, L_p\}$

$\nu_2(K_2) = \{L'_1, \dots, L'_n, L'_{n+1}, \dots, L'_q\}$ mit $p \geq m, q \geq n$

$R = \sigma(\{L_{m+1}, \dots, L_p, L'_{n+1}, \dots, L'_q\})$

$(\sigma)L_1 = \dots = \sigma(L_m) = \bar{L}$

Resolutionsalgorithmus: **Ziel:** Untersuche ob $\{\varphi_1, \dots, \varphi_k\} \models \varphi$ gilt.

1. Sei ψ die Formel $\varphi_1 \wedge \dots \wedge \varphi_k \wedge \neg\varphi$
2. Überführe ψ in Skolem-Normalform $\forall X_1 \dots X_n \xi$.
3. Überführe ξ in KNF bzw. in die entsprechende Klauselmenge $\mathcal{K}(\xi)$.
4. Berechne $Res^*(\mathcal{K}(\xi))$.
Falls die leere Klausel gefunden wurde, brich ab mit "Yes".
Falls Res^*

10.5.06

Prädikatenlogische Resolution ist Korrekt und Vollständig.

\mathcal{K} unerfüllbar gdw. $\square \in Res^*(\mathcal{K})$

Nachteil: $Res^*(\mathcal{K})$ ist zu groß, da man beliebig Klauseln miteinander resolvidieren darf.

Lösung: Schränke Resolution ein, so dass man nur noch zwischen bestimmten Klauseln resolvidieren darf, und so dass die Vollständigkeit erhalten bleibt. \rightarrow effizienter und genauso mächtig.

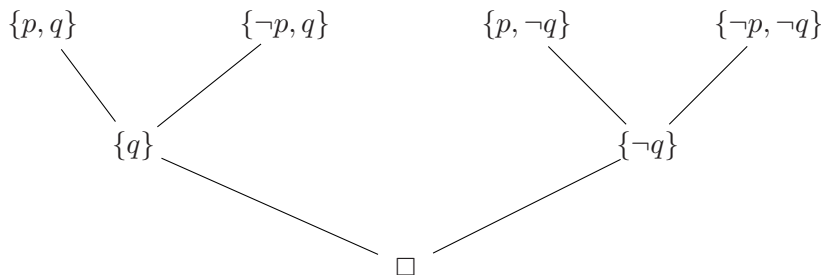
3.5 Einschränkung der Resolution

3.5.1 Lineare Resolution

Definition 3.5.1 (Lineare Resolution)

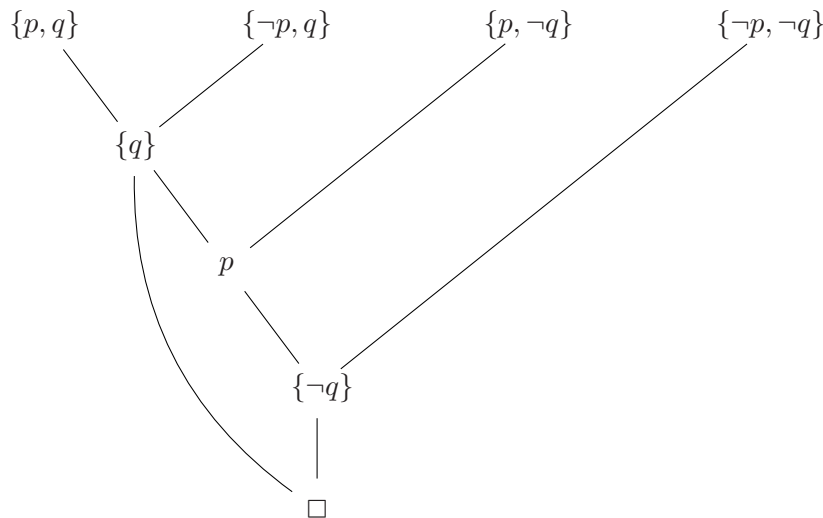
Sei \mathcal{K} eine Klauselmenge. Die leere Klausel \square ist aus der Klausel $K \in \mathcal{K}$ **linear resolvidierbar** gdw. es eine Folge von Klauseln K_1, \dots, K_n gibt mit $K_1 = K, K_m = \square$ und so dass für alle $2 \leq i \leq m$ gilt: K_i ist Resolvent von K_{i-1} und einer Klausel aus $\{K_1, \dots, K_{i-1}\} \cup \mathcal{K}$

Beispiel 3.5.2 $\Delta_0 = \{p, q\}$



Dies ist kein linearer Resolutionsbeweis, denn im zweiten Schritt hätte man mit $\{q\}$ und einer anderen Klausel resolvidieren müssen.

Lineare Resolution:



Satz 3.5.3 (Korrektheit und Vollständigkeit der linearen Resolution)

Sei \mathcal{K} eine Klauselmenge. Dann ist \mathcal{K} unerfüllbar gdw. \square aus einer Klausel in \mathcal{K} linear resolvierbar ist.

Falls \mathcal{K} eine **minimale** unerfüllbare Klauselmenge ist (d.h. $\mathcal{K} \setminus \{K\}$ ist erfüllbar für alle $k \in \mathcal{K}$), dann ist \square sogar aus **jeder** Klausel in \mathcal{K} linear resolvierbar.

Beweis

\Rightarrow [Korrektheit:] folgt aus der Korrektheit der (vollen) Resolution (Satz 3.4.10).

\Leftarrow [Vollst.:] Zeige erst Vollständigkeit der linearen Resolution in der Aussagenlogik und lifte dies dann in die Prädikatenlogik.

1. Vollständigkeit der aussagenlog. linearen Resolution (d.h. \mathcal{K} ist variablenfrei)

Sei $\mathcal{K}_{min} \subseteq \mathcal{K}$ eine minimale unerfüllbare Teilmenge von \mathcal{K} . Offensichtlich gilt $\mathcal{K}_{min} \neq \emptyset$ (\emptyset ist erfüllbar, sogar allgemeingültig).

Wir zeigen, dass \square aus **jeder** Klausel K in \mathcal{K}_{min} resolvierbar ist.

Induktion über die Anzahl n der verschiedenen atomaren Formeln in \mathcal{K}_{min} :

Ind. Anf.: $n = 0 \rightarrow K = \emptyset, \mathcal{K}_{min} = \{\square\} \checkmark$

Ind. Schluss: $n > 0$:

1. Fall: $|K| = 1$, d.h. $K = \{L\}$

2. Fall: $|K| > 1$ (Übung).

\mathcal{K}^+ entsteht aus \mathcal{K}_{min} , indem man

- alle Klauseln weglässt, die L enthalten und
- \bar{L} aus den verbleibenden Klauseln streicht.

$\Rightarrow \mathcal{K}^+$ enthält höchstens $n - 1$ verschiedene atomare Formeln.

\mathcal{K}^+ ist unerfüllbar ($S \models \mathcal{K}^+ \rightarrow \underbrace{S'} \models \mathcal{K}_{min}$, Widerspruch da \mathcal{K}_{min} wie S aber zusätzlich $S' \models L$

unerfüllbar).

$\Rightarrow \square$ ist aus jeder Klausel in \mathcal{K}^+ linear resolvierbar. (*)

Es muss eine Klausel $K^+ \in \mathcal{K}^+$ geben, so dass $K^+ \notin \mathcal{K}_{min} \rightarrow K^+ \cup \{\bar{L}\} \in \mathcal{K}_{min}$ (sonst wäre $\mathcal{K}^+ \subset \mathcal{K}_{min}$ eine echte unerfüllbare Teilmenge. Widerspruch zur Minimalität von \mathcal{K}_{min}

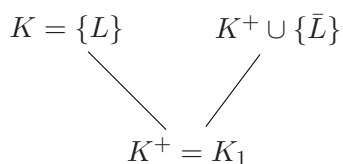
Wegen (*) ist \square aus K^+ in \mathcal{K}^+ linear resolvierbar.

Es gibt also eine Folge von Klauseln K_1, \dots, K_m mit $K_1 = K^+, K_m = \square$ und K_i ist Resolvent von K_{i-1} und einer Klausel aus $\{K_1, \dots, K_{i-1}\} \cup \mathcal{K}^+$

Für alle $1 \leq i \leq m$ ex. dann auch eine lineare Resolutionsfolge K_1, \dots, K_i in \mathcal{K}_{min}

Induktion über i

Ind. Anf.



Ind. Schluss: $i > 1$, K_i ist Resolvent von K_{i-1} und einer Klausel aus $\{K_1, \dots, K_{i-1}\}$ ist linear resolvierbar in \mathcal{K}_{min}

Falls K_i Resolv. von K_{i-1} und $K' \in \{K_1, \dots, K_{i-1}\}$ ist, dann ist auch $K, K_1, \dots, K_2, \dots, K_{i-1}, K_i$ lin. Res. in \mathcal{K}_{min}

Falls K_i Resolv. von K_{i-1} und $K' \in \mathcal{K}^+ \cap \mathcal{K}_{min}$, dann ist auch $K, K_1, \dots, K_2, \dots, K_{i-1}, K_i$ lin. Res. in \mathcal{K}_{min}

Falls K_i Resolv. von K_{i-1} und $K' \in \mathcal{K}^+, K' \notin \mathcal{K}_{min}$ dann ist $K' \cup \{\bar{L}\} \in \mathcal{K}_{min}$

Dann ist $K, K_1, \dots, K_2, \dots, K_{i-1}, K_i \cup \{\bar{L}\}, K_i$ (K_i entsteht durch Res. von $K_i \cup \{\bar{L}\}$ mit $K = \{L\}$)

2. Vollständigkeit der Prädikatenlog. linearen Resolution

Analog zum Vollständigkeitsbeweis der vollen prädikatenlogischen Resolution.

\mathcal{K} unerfüllbar

\rightarrow es ex. endl. unerfüllbare Menge von Grundinstanzen von Klauseln aus \mathcal{K} (Satz 3.3.9 (a))

\rightarrow es ex. Herleitung von \square durch lin. Resolution aus der Menge der Grundinstanzen der Klauseln von \mathcal{K}

\rightarrow es ex. eine Herleitung von \square durch lineare Resolution aus \mathcal{K} (genauso wie im Beweis von Satz 3.4.10)

□

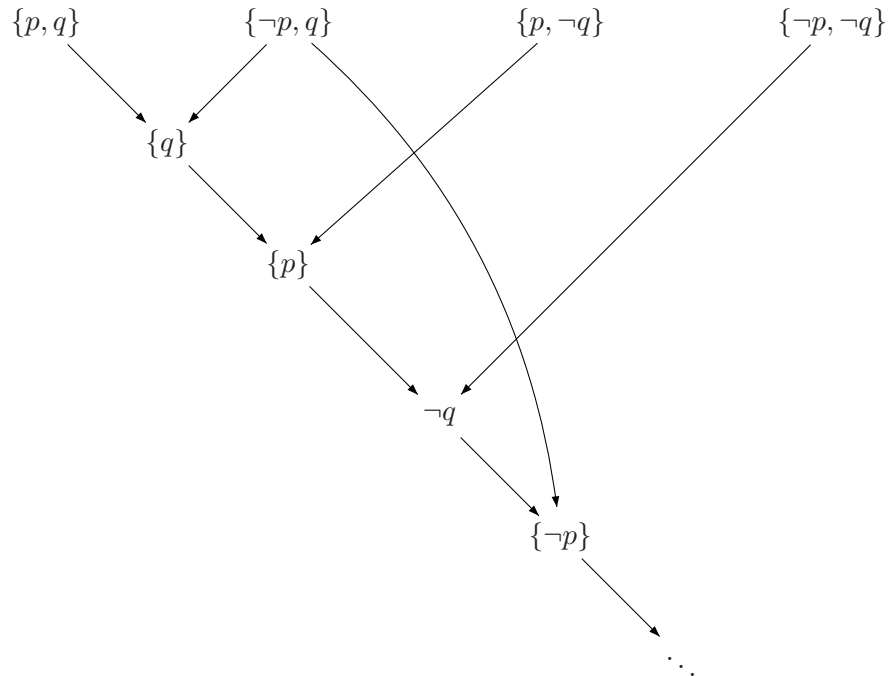
Bisher: bei jedem Resolutionsschritt liegt eine Elternklausel fest (zuletzt erzeugter Resolvent). Andere Elternklausel ist aus ursprünglicher Klauselmengung oder ein früher erzeugter Resolvent).

3.5.2 Input- und SLD-Resolution

Definition 3.5.4 (Input-Resolution)

Sei \mathcal{K} eine Klauselmenge. Die leere Klausel \square ist aus der Klausel $K \in \mathcal{K}$ durch **Input-Resolution** herleitbar gdw. es eine Folge von Klauseln K_1, \dots, K_m gibt, mit $K_1 = K, K_m = \square$ und so dass für alle $2 \leq i \leq m$ gilt: K_i ist Resolvent von K_{i-1} und einer Klausel aus \mathcal{K}
 \Rightarrow Input-Resolution ist Spezialfall der linearen Resolution.

Beispiel 3.5.5



\square ist nicht mit Input-Resolution herleitbar \Rightarrow Input-Resolution ist nicht vollständig!

Lösung: Einschränkung auf **Hornklauseln**(enthält höchstens ein positives Literal).

Auf Hornklauseln ist Input-Resolution vollständig.

\Rightarrow Durch Einschränkung der Sprache (d. Ausdrucksstärke) kann man enormen Effizienzgewinn durch Input-Resolution bekommen, ohne die Vollständigkeit zu verlieren.

Definition 3.5.6 (Hornklausel)

Eine Klausel K ist eine Hornklausel gdw. sie höchstens ein pos. Literal enthält (d.h. höchstens eine nicht-negierte atomare Formel).

Eine Hornklausel $\{\neg A_1, \dots, \neg A_k\}$ ohne pos. Literale heißt **negativ**.

Eine Hornklausel $\{B, \neg C_1, \dots, \neg C_n\}$ mit einem positiven Literal heißt **definit**

Beispiel 3.5.7 $\{\{p, \neg q\}, \{\neg r, \neg p, s\}, \{s\}\}$

bedeutet $(p \vee \neg q) \wedge (\neg r \vee \neg p \vee s) \wedge s$

bedeutet $(q \rightarrow p) \wedge (r \wedge p \neg s) \wedge s$

Hornklauselmenge $\hat{=}$ Konjunktion von Implikationen.

Zusammenhang zur Logikprogrammierung:

- Fakten $\hat{=}$ Hornklausel ohne negative Literale, z.B. $\{s\}, \{mutterVon(renate, susanne)\}$

Schreibweise: S .

$mutterVon(renate, susanne)$.

- Regeln $\hat{=}$ Hornklausel mit positiven und negativen Literalen, z.B. $\{\neg r, \neg p, s\}, \{\neg verheiratet(V, F), \neg mutterVon(V, K)\}$

Schreibweise: $S : \neg r, p$.

$vaterVon(V, K) : \neg verheiratet(V, F), mutterVon(F, K)$.

- Anfragen $\hat{=}$ Hornklauseln ohne positive Literale,

z.B. $\{\neg p, \neg q\}, \{\neg vaterVon(gerd, Y)\}$

Schreibweise: $? - p, q$.

$? - vaterVon(gerd, Y)$.

Fakten und Regeln sind definite Hornklauseln, Anfragen sind negative Hornklauseln.

Definite Hornklauseln $\hat{=}$ Klauseln des Logikprogramms.

negative Hornklauseln $\hat{=}$ Anfrage

Betrachtung von Hornklauseln schränkt die Ausdrucksstärke ein, aber Unerfüllbarkeitsnachweis ist viel effizienter.

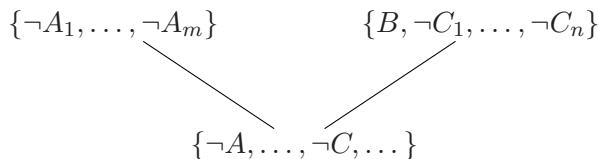
	volle Klauseln	Hornklauseln
Aussagenlogik	NP-vollständig (SAT)	Linearzeit
Prädikatenlogik	unentscheidbar	unentscheidbar
	aber semi-entscheidbar	aber semi-entscheidbar (effizienter)

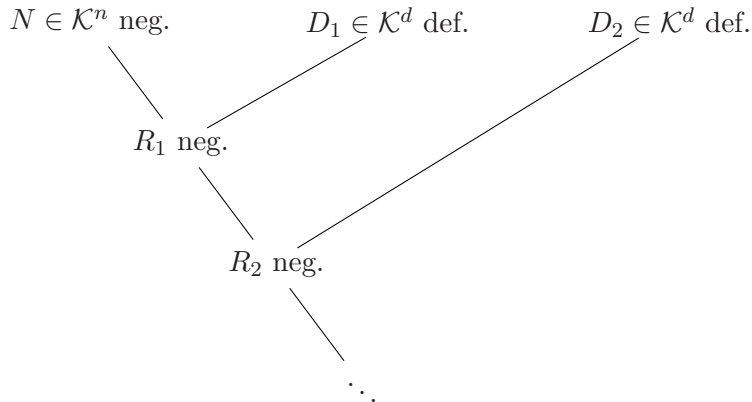
Statt Vollständigkeit der Input-Resolution auf Hornklauselmengen nachzuweisen, schränke Input-Resolution weiter ein zur SLD-Resolution. Dann zeigen wir, dass SLD-Resolution auf Hornklauselmengen vollständig ist.

Definition 3.5.8 (SLD-Resolution)

Sei \mathcal{K} eine Menge v. Hornklauseln mit $\mathcal{K} = \mathcal{K}^d \uplus \mathcal{K}^n$, wobei \mathcal{K}^d die definiten Klauseln und \mathcal{K}^n die neg. Klauseln von \mathcal{K} enthält. Die leere Klausel \square ist aus $K \in \mathcal{K}^n$ durch SLD-Resolution herleitbar gdw. es eine Folge K_1, \dots, K_m gibt mit $K_1 = K \in \mathcal{K}^n, K_m = \square$ für alle $1 \leq i \leq m$: K_i ist Resolvent von K_{i-1} und einer Klausel aus \mathcal{K} . (K_1, \dots, K_m sind negative Klauseln)

Negative Hornklauseln können nur mit definiten Hornklauseln resolviert werden.





SLD = Linearresolution with
 Selectionfunktion for
 Definite Clauses

Zwei Indeterminismen:

- Wahl des Literals aus der negativen Klausel, mit dem resolviert werden soll.
- Wahl der definiten Elternklausel

Selektionsfunktion "löst" diese Indeterminismen
 Ignoriere Selektionsfunktion \Rightarrow "LUSH-Resolution"

LUSH = Linear resolution with
 Unrestricted
 Selection for
 Horn clauses

Satz 3.5.9 (Korrektheit und Vollständigkeit der SLD-Resolution)

Sei \mathcal{K} eine Menge von Hornklauseln. Dann ist \mathcal{K} unerfüllbar gdw. \square aus einer negativen Klausel $N \in \mathcal{K}$ durch SLD-Resolution herleitbar ist.

Beweis

\Leftarrow *Korrektheit: Klar (da SLD-Resolution ein Spezialfall der vollen Resolution, die vollständig ist. (Satz 3.4.10))*

\Rightarrow *Vollständigkeit: \mathcal{K} enthält eine negative Klausel, denn jede Menge definiter Hornklauseln ist erfüllbar (Ein Modell ist die Struktur die alle atomaren Formeln erfüllt).*

Sei $\mathcal{K}_{min} \subseteq \mathcal{K}$ eine minimale unerfüllbare Teilmenge $\rightarrow \mathcal{K}_{min}$ enthält auch mindestens eine negative Klausel N . Wegen Vollständigkeit der linearen Resolution (Satz 3.5.3) ist \square aus jeder Klausel von \mathcal{K}_{min} durch lineare Resolution herleitbar. Also existiert auch ein linearer Resolutionsbeweis von \square der mit N startet.

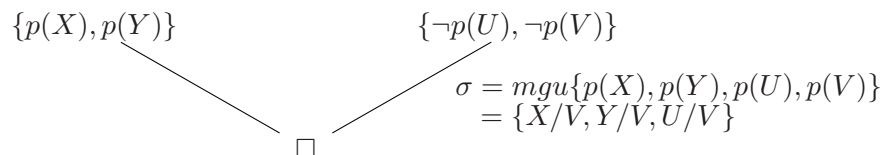
Dieser lineare Resolutionsbeweis ist auch ein SLD-Resolutionsbeweis:

- startet mit negativer Klausel
- da alle Resolventen negativ sind, können nie zwei Resolventen miteinander resolviert werden \Rightarrow ist Input-Resolutionsbeweis.

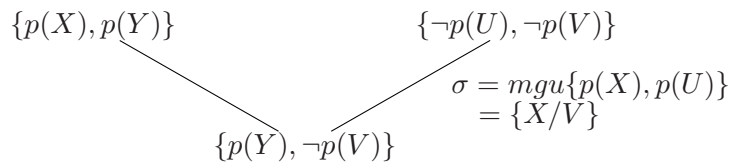
□

Weitere (letzte) Einschränkung Binäre Resolution: $m = n = 1$, d.h. in jedem Resolutionsschritt wird nur zwischen jeweils **einem** Literal der Elternklauseln resolviert. Aber binäre Resolution ist nicht vollständig!

Beispiel 3.5.10



Mit binärer Resolution



Aber binäre Resolution ist vollständig auf Hornklauselmengen

Satz 3.5.11 (Korrektheit und Vollständigkeit der binären SLD-Resolution)

Sei \mathcal{K} eine Hornklauselmenge. Dann ist \mathcal{K} unerfüllbar gdw. \square aus einer neg. Klausel N in \mathcal{K} durch binäre SLD-Resolution herleitbar ist.

Beweis

\Leftarrow Klar ✓

\Rightarrow Wegen der Vollständigkeit der vollen SLD-Resolution (Satz 3.5.8) ex ein SLD-Resolutions-Beweis von \square , der mit N startet.

Zeige: Jeder Resolutionsschritt mit voller SLD-Resolution kann durch eine Folge binärer SLD-Resolutionsschritte ersetzt werden.

SLD-Resolutionsschritt:

$$\begin{array}{ccc}
 N = \{\neg A_1, \dots, \neg A_m, \dots, \neg A_p\} & & K = \{B, \neg C_1, \dots, \neg C_n\} \\
 & \searrow & \nearrow \\
 & & \sigma = mgu\{A_1, \dots, A_m, B\} \\
 & & R = \sigma(\{\neg A_{m+1}, \dots, \neg A_p, \neg C_1, \dots, \neg C_n\})
 \end{array}$$

Durch Induktion über m zeigen wir, dass man diesen Schritt durch m binäre SLD-Resolutionsschritte ersetzen kann und den gleichen Resolventen R (bis auf Variablenumbenennung) erhält.

Ind-Anf.: $m = 1$ ✓ (dann ist das bereits binäre Resolution)

Ind-Schluss: $m > 1$

$$\begin{array}{ccc}
 N & & K & & K \\
 & \searrow & & \nearrow & \\
 & & \sigma = mgu\{A_1, B\} & & \\
 N' = \sigma'(\{\neg A_2, \dots, \neg A_m, \dots, \neg A_p, \neg C_1, \dots, \neg C_n\}) & & & & K \\
 & & & \searrow & \\
 & & & & \text{geht da } \{\sigma'(A_2), \dots, \sigma'(A_m), B\} \text{ unifizierbar.}
 \end{array}$$

\xrightarrow{IH} geht auch in $m - 1$ binären Resolutionsschritten.

Es bleibt zu zeigen: $R' = R$ bis auf Variablenumbenennung (Übung).

4 Logikprogramme

Zunächst "reine" Logikprogramme:

4.1 Syntax und Semantik

4.2 Universalität der Logikprogramme

4.3 Indeterminismen der Logikprogrammierung.

4.1 Syntax und Semantik von Logikprogrammen

Bisher:	Klausel = Menge von Literalen	Klauselmenge = Menge von Klauseln
Jetzt:	Klausel = Folge von Literalen.	Klauselmenge = Folge von Klauseln.

Also: Reihenfolge von Literalen/Klauseln spielt eine Rolle. Literale/Klauseln können mehrfach auftreten.

Definition 4.1.1 (Syntax von Logikprogrammen)

Eine nicht-leere Menge \mathcal{P} von definiten Hornklauseln über der Signatur (Σ, Δ) heißt **Logikprogramm** über (Σ, Δ) . Die Klauseln in \mathcal{P} heißen **Programmklauseln**. Man unterscheidet zwei Arten von Programmklauseln:

- **Fakten** sind Klauseln der Art $\{B\}$, B atomare Formel
- **Regeln** sind Klauseln der Art $\{B, \neg C_1, \dots, \neg C_n\}$, $n \geq 1$, B, C_1, \dots, C_n atomare Formeln.

Aufrufen eines Logikprogramms geschieht durch eine

- **Anfrage:** G der Art $\{\neg A_1, \dots, \neg A_k\}$, $k \geq 1$

In der Prädikatenlogik haben wir nur untersucht, ob eine Formel aus einer Formelmenge folgt. Bei Logikprogrammen will man auch noch wissen, wie die Variablen der Formel instanziiert werden müssen, damit die Formel aus der Formelmenge folgt \Rightarrow "Antwortsubstitution"

Wie bisher: Klausel $\hat{=}$ allg. Disjunktion ihrer Literale.

Aufruf des Logikprogramms \mathcal{P} mit Anfrage $G = \{\neg A_1, \dots, \neg A_k\}$: bedeutet, dass man untersuchen will, ob folgendes gilt: $\mathcal{P} \models \exists \underbrace{X_1, \dots, X_p}_{\text{Variablen in } A_1 \dots A_k} A_1 \wedge \dots \wedge A_k$ (*)

Variablen in \mathcal{P} sind implizit allquantifiziert

Variablen in Anfrage sind implizit existenzquantifiziert.

(*) ist äquivalent zu: $\mathcal{P} \cup \{G\}$ ist unerfüllbar.,

\Leftrightarrow es existiert eine endliche Menge von Grundinstanzen von $\mathcal{P} \cup \{G\}$, die unerfüllbar ist (nach Satz 3.3.9a). Diese muss auch Grundinstanz von G enthalten. Wegen der Vollständigkeit der SLD-Resolution reicht es jeweils genau eine Grundinstanz von G zu betrachten.

\Leftrightarrow es existieren Grundterme t_1, \dots, t_p so dass $\mathcal{P} \cup \{G[X_1/t_1, \dots, X_p/t_p]\}$ unerfüllbar ist.

\Leftrightarrow es existieren Grundterme t_1, \dots, t_p so dass $\mathcal{P} \models A_1 \wedge \dots \wedge A_k \underbrace{[X_1/t_1, \dots, X_p/t_p]}_{\text{Antwortsubstitution}}$

Falls (*) gilt, soll das Logikprogramm auch die Antwortsubstitution berechnen. Wir erlauben auch Antwortsubstitutionen, die X_1, \dots, X_p durch Terme mit Variablen ersetzen. Die verbleibenden Variablen können dann weiter durch beliebige Terme ersetzt werden.

Beispiel 4.1.2 Sei \mathcal{P} folgendes Logikprogramm:

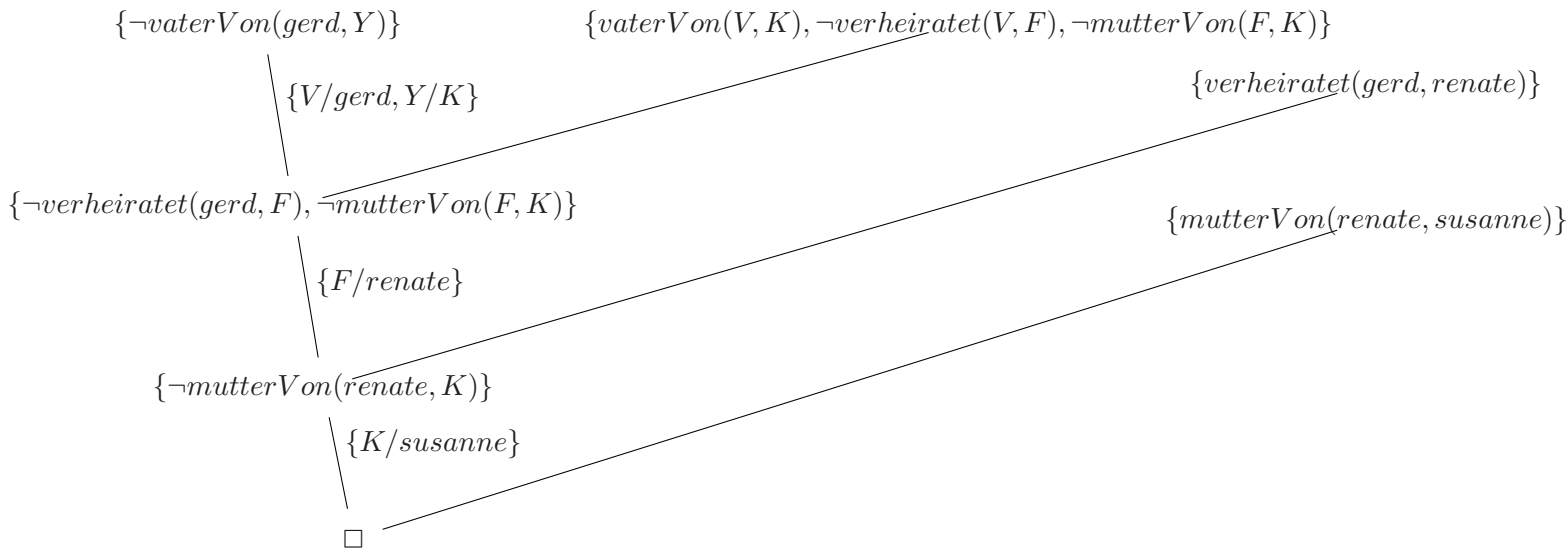
```
mutterVon(renate,susanne).
verheiratet(gerd,renate).
vaterVon(V,K):- verheiratet(V,F), mutterVon(F,K).
```

Schreibweise von \mathcal{P} als Klauselmenge:

```
{mutterVon(renate,susanne)},
{verheiratet(gerd,renate)},
{vaterVon(V,K), ¬verheiratet(V,F), ¬mutterVon(F,K)}
```

Anfrage: ? – $vaterVon(gerd,Y)$, d.h. $G = \{vaterVon(gerd,Y)\}$.

Statt $\mathcal{P} \models \exists Y vaterVon(gerd,Y)$ zu untersuchen, untersuche Unerfüllbarkeit von $\mathcal{P} \cup \{G\}$ durch binäre SLD-Resolution.



Antwortsubstitution:

- Komposition der mgu's $\{K/susanne\} \circ \{F/renate\} \circ \{V/gerd, Y/K\}$
 $= \{V/gerd, Y/susanne, F/renate, K/susanne\}$
- Einschränken aus Variable Y der Anfrage.

Es gibt drei äquivalente verschiedene Arten zur Definition der Semantik von Logikprogrammen

- deklarative Semantik
- prozedurale Semantik
- Fixpunkt-Semantik

4.1.1 Deklarative Semantik der Logikprogrammierung

Definiere die Semantik für ein Logikprogramm \mathcal{P} zusammen mit Anfrage G .

Deklariere Semantik von \mathcal{P} m $G =$ alle "wahren" Grundinstanzen von G , d.h. alle Grundinstanzen von G , die aus \mathcal{P} folgen.

Definition 4.1.3 (Deklarative Semantik)

Sei \mathcal{P} ein Logikprogramm und $G = \{\neg A_1, \dots, \neg A_k\}$ eine Anfrage.

Dann ist die deklarative Semantik von \mathcal{P} bezüglich G definiert als:

$D[\mathcal{P}, G] = \{\sigma(A_1 \wedge \dots \wedge A_k) \mid \mathcal{P} \models \sigma(A_1 \wedge \dots \wedge A_k), \sigma \text{ ist Grundsubstitution}\}$.

Jede Grundinstanz $\sigma(A_1 \wedge \dots \wedge A_k) \in D[\mathcal{P}, G]$ enthält als "Lösung" die entsprechende Instanz der Variablen aus A_1, \dots, A_k

Beispiel 4.1.4 \mathcal{P}, G wie in Bsp. 4.1.2

$\mathcal{P} \models \sigma(\text{vaterVon}(\text{gerd}, Y))$ gilt gdw. $\sigma(Y) = \text{susanne}$

$D[\mathcal{P}, G] = \{\text{vaterVon}(\text{gerd}, \text{susanne})\}$.

$D[\mathcal{P} \cup \{\text{mutterVon}(\text{renate}, \text{peter})\}, G] = \{\text{vaterVon}(\text{gerd}, \text{susanne}), \text{vaterVon}(\text{gerd}, \text{peter})\}$.

4.1.2 Prozedurale Semantik der Logikprogrammierung

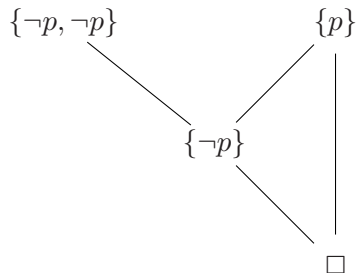
Operationelle Semantik durch Angabe eines Interpreters

Interpreter arbeitet auf **Konfigurationen**: Paar aus Anfrage und Substitution.

Startkonfiguration: (G, \emptyset) , gewünschte Zielkonfiguration (\square, σ) . Antwortsubstitution ist dann σ eingeschränkt auf die Variablen von G .

Man kann eine Konfiguration in die nächste durch **binäre SLD-Resolution** umformen. Zwei Unterschiede zur binären SLD-Resolution in Kapitel 3:

- Klauseln sind jetzt **Folgen** (statt **Mengen**) von Literalen
z.B.



Korrektheit und Vollständigkeit bleiben erhalten

- statt Variablenumbenennungen auf **beide** Elternklauseln anzuwenden, darf man sie jetzt nur noch auf die (definiten) Programm-Klauseln anwenden, nicht auf die anderen negativen Elternklauseln. ("standardisierte" SLD-Resolution).

Definition 4.1.5 (Prozedurale Semantik eines Logikprogramms)

Sei \mathcal{P} ein Logikprogramm

- Eine **Konfiguration** ist ein Paar (G, σ) aus einer Anfrage G und einer Substitution σ
- Es gibt einen **Rechenschritt** $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ gdw.
 - $G_1 = \{\neg A_1, \dots, A_k\}$ mit $k \geq 1$
 - es ex. Programmklausel $K \in \mathcal{P}$ und eine Variablenumbenennung ν mit $\nu(K) = \{B, \neg C_1, \dots, \neg C_n\}, n \geq 0$ so dass:
 - * G_1 und $\nu(K)$ keine gemeinsamen Variablen haben
 - * es ein $1 \leq i \leq k$ gibt, so dass A_i und B mit einem mgu σ unifizierbar sind.
 - $G_2 = \sigma(\{\neg A_1, \dots, \neg A_{i-1}, \neg C_1, \dots, \neg C_n, \neg A_{i+1}, \dots, \neg A_k\})$
 - $\sigma_2 = \sigma \circ \sigma_1$
- Eine **Berechnung** von \mathcal{P} bei Eingabe von $G = \{\neg A_1, \dots, \neg A_n\}$ ist eine (endl. oder unendl.) Folge von Konfigurationen der Form $(G, \emptyset) \vdash_{\mathcal{P}} (G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \dots$
- Eine Berechnung, die mit (G, \emptyset) startet heißt **erfolgreich**, falls sie mit (\square, σ) endet. Die **Antwortsubstitution** ist σ eingeschränkt auf die Variablen aus G
- Prozedurale Semantik von \mathcal{P} bezüglich G ist definiert als:
 $P[\mathcal{P}, G] = \{\sigma'(A_1 \wedge \dots \wedge A_k) \mid (G, \emptyset) \vdash_{\mathcal{P}}^+ (\square, \sigma), \sigma'(A_1 \wedge \dots \wedge A_k) \text{ ist die Grundinstanz von } \sigma(A_1 \wedge \dots \wedge A_k)\}$
 Hierbei ist " $\vdash_{\mathcal{P}}^+$ " die transitive Hülle von " $\vdash_{\mathcal{P}}$ ".
 (Es gilt: $(G, \emptyset) \vdash_{\mathcal{P}}^+ (\square, \sigma)$ gdw. $(G, \emptyset) \vdash_{\mathcal{P}} \dots \vdash_{\mathcal{P}} (\square, \sigma)$.)
 Analog dazu definieren wir $\vdash_{\mathcal{P}}^l$ für ein $l \in \mathbb{N}$ gdw. $(G, \emptyset) \vdash_{\mathcal{P}} \dots \vdash_{\mathcal{P}} (\square, \sigma)$
 $\underbrace{\hspace{10em}}_{l \text{ Schritte}}$

19.05.06

Beispiel 4.1.6 \mathcal{P}, G wie in Bsp. 4.1.2

$$\begin{aligned}
 (\{\neg \text{vaterVon}(\text{gerd}, Y), \emptyset\}) &\vdash_{\mathcal{P}} (\{\neg \text{verheiratet}(\text{gerd}, F), \neg \text{mutterVon}(F, K)\}, \{Y/K, V/\text{gerd}\}) \\
 &\vdash_{\mathcal{P}} (\{\neg \text{mutterVon}(\text{renate}, k)\}, \underbrace{\{F/\text{renate}\} \circ \{Y/K, V/\text{gerd}\}}_{\{F/\text{renate}, Y/K, V/\text{gerd}\}}) \\
 &\vdash_{\mathcal{P}} (\square, \underbrace{\{K/\text{susanne}\} \circ \{F/\text{renate}, Y/K, V/\text{gerd}\}}_{\{K/\text{susanne}, F/\text{renate}, Y/\text{susanne}, V/\text{gerd}\}})
 \end{aligned}$$

Da die ursprüngliche Anfrage nur die Variable Y enthält.:

Antwortsubstitution ist $\{Y/\text{susanne}\}$.

$P[\mathcal{P}, G, \square] = \{\text{vaterVon}(\text{gerd}, \text{susanne})\}$

$\vdash_{\mathcal{P}}$ hat 2 Indeterminismen:

- Wahl der Programm-Klausel K
- Wahl des Literals A_i

Beispiel 4.1.7 $\mathcal{P} = \{\{p(X, Z), \neg q(X, Y), \neg p(Y, Z)\}, \{p(U, U)\}, \{q(a, b)\}\}$
Anfrage: $G = \{\neg p(V, b)\}$

$$\begin{aligned} (\{\neg p(V, b)\}, \emptyset) &\vdash_{\mathcal{P}} (\{\neg q(V, Y), \neg p(Y, b)\}, \{X/V, Z/b\}) \\ &\vdash_{\mathcal{P}} (\{\neg p(b, b)\}, \{V/a, Y/b, X/a, Z/b\}) \\ &\vdash_{\mathcal{P}} (\{\neg q(B, Y'), \neg p(Y', b)\}, \{X'/b, Z'/b, V/a, Y/b, X/a, Z/b\}) \\ &\vdash_{\mathcal{P}} (\{\neg q(b, b)\}, \{U/b, Y'/b, X'/b, Z'/b, V/a, Y/b, X/a, Z/b\}) \end{aligned}$$

Die Herleitung ist endlich aber nicht erfolgreich.

Alternativ:

$$\begin{aligned} (\{\neg p(V, b)\}, \emptyset) &\vdash_{\mathcal{P}} (\{\neg q(V, Y), \neg p(Y, b)\}, \{X/V, Z/b\}) \\ &\vdash_{\mathcal{P}} (\{\neg p(b, b)\}, \{V/a, Y/b, X/a, Z/b\}) \\ &\vdash_{\mathcal{P}} (\{\square\}, \{U/b, V/a, Y/b, X/a, Z/b\}) \end{aligned}$$

Herleitung erfolgreich. Antwortsstitution: $\{V/a\}$

Andere Alternative

$$(\{\neg p(V, b)\}, \emptyset) \vdash_{\mathcal{P}} (\{\square\}, \{V/b\})$$

Herleitung erfolgreich, Antwortsstitution: $\{V/a\}$

Indeterminismen beeinflussen:

- Erfolg / Scheitern nach endlich vielen Schritten / unendliche Herleitung
- Unterschiedliche Antwortsstitutionen bei erfolgreichen Pfaden

Momentan: lasse Indeterminismen zu $\rightarrow p(a, b), p(b, b) \in P[\mathcal{P}, G]$

Satz 4.1.8 (Äquivalenz der deklarativen und prozeduralen Semantik (Clark))

Sei \mathcal{P} ein Logikprogramm und $G = \{\neg A_1, \dots, \neg A_k\}$ eine Anfrage.

Dann gilt: $D[\mathcal{P}, G] = P[\mathcal{P}, G]$

Im Prinzip: $\supseteq \hat{=}$ Korrektheit der binären SLD-Resolution

$\subseteq \hat{=}$ Vollständigkeit der binären SLD-Resolution

Aber: Die Antwortsstitution muss noch mit berücksichtigt werden.

Beweis

- Zeige erst $P[\mathcal{P}, G] \subseteq D[\mathcal{P}, G]$
 Sei $\sigma'(A_1 \wedge \dots \wedge A_k) \in P[\mathcal{P}, G]$
 $\rightarrow \underbrace{(G, \emptyset) \vdash_{\mathcal{P}} \dots \vdash_{\mathcal{P}} (\square, \sigma)}_{l \text{ Schritte}}, \text{ wobei } \sigma'(A_1 \wedge \dots \wedge A_k) \text{ Grundinstanz von } \sigma(A_1 \wedge \dots \wedge A_k) \text{ ist.}$

Induktion über l

$(G, \emptyset) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash_{\mathcal{P}} (G_2, \delta_2 \circ \delta_1) \dots \vdash_{\mathcal{P}} (\square, \delta_l \circ \dots \delta_2 \circ \delta_1)$

Es existiert also ein A_i , ein $K \in \mathcal{P}$ mit $\nu(K) = \{B, \neg C_1, \dots, \neg C_n\}$

$\delta_1 = \text{mgu}(A_1, B)$, wobei $G_1 = \delta_1(\{\neg A_1, \dots, \neg A_{i-1}, \neg C_1, \dots, \neg C_n, \neg A_{i+1}, \dots, \neg A_K\})$

Induktions Anfang: $l = 1 \rightarrow G_1 = \square$

$\rightarrow i = K = 1, n = 0$ (K ist Faktum).

$\sigma = \delta_1$

Zu zeigen ist: jede Grundinstanz von $\sigma(A_1)$ folgt aus \mathcal{P}

Da $K \in \mathcal{P}, \nu(K) = \{B\}$ gilt $\mathcal{P} \models B$

$\rightarrow \mathcal{P} \models \underbrace{\delta_1(B)}_{=\delta_1(A)=\sigma(A)}$

Induktionsschluss: $l > 1$

$(G_1, \emptyset) \vdash_{\mathcal{P}} (G_2, \delta_2) \vdash_{\mathcal{P}} \dots \vdash_{\mathcal{P}} (\square, \delta_l \circ \dots \delta_2)$ hat Länge $l - 1$

Induktions-Hypothese: Alle Grundinstanzen der Atome in $\delta_l \circ \dots \delta_2(G_1)$ folgen aus \mathcal{P}

\rightarrow Jede Grundinstanz von $\delta_l(\dots \delta_2(\delta_1(\{A_1 \wedge \dots \wedge A_{i-1} \wedge C_1, \dots \wedge C_n \wedge A_{i+1} \wedge \dots \wedge A_k\}))) \dots$ folgt aus \mathcal{P} .

Zu zeigen ist: Jede Grundinstanz von $\delta_l(\dots \delta_2(\delta_1(\{A_1 \wedge \dots \wedge A_k\}))) \dots$ folgt aus \mathcal{P} . Ist klar für alle A_j mit $j \neq i$

Da alle Grundinstanzen von $\delta_l(\dots \delta_1(C_1 \wedge \dots \wedge C_n))$ aus \mathcal{P} folgen, folgen auch alle Grundinstanzen von $\delta_l(\dots \underbrace{\delta_1(B)}_{=\delta_1(A_i)} \dots)$ aus \mathcal{P} (Denn $\{B, \neg C_1, \dots, \neg C_n\}$ ist Programm-Klausel).

- $D[\mathcal{P}, G] \subseteq P[\mathcal{P}, G]$ Verwende die Vollständigkeit der binären SLD-Resolution und eine Variante des Lifting-Lemmas.

4.1.3 Fixpunkt-Semantik der Logikprogrammierung

Nicht modelltheoretisch definiert (wie die deklarative Semantik), sondern man versucht (ähnlich zur prozeduralen Semantik) durch resolutionsähnliche Schritte wahre Aussagen herzuleiten.

Im Unterschied zur prozeduralen Semantik, gebe keine Anfrage vor, sondern gehe nur vom Programm aus und berechne alle Anfragen (ohne Variablen), die man in 0,1,2,3,... Schritten herleiten könnte.

Nach höchstens 0 Beweisschritten: keine Aussage beweisbar.

1 Beweisschritten: alle Grundinstanzen von Fakten beweisbar (M_1)

2 Beweisschritten: alle Aussagen, die durch höchstens 1 Regelanwendung + Faktum beweisbar sind (M_2)

Die Funktion $trans_{\mathcal{P}}$:

$trans_{\mathcal{P}}(M) =$ alle Aussagen, die ausgehend von M mit höchstens einem weiteren Resolutionsschritt beweisbar sind.

M ist eine Menge von Aussagen (atomare Formeln ohne Variablen).

$$M_0 = \emptyset$$

$$M_1 = trans_{\mathcal{P}}(M_0)$$

$$M_2 = trans_{\mathcal{P}}(M_1) \quad M_i = trans_{\mathcal{P}}^i(\emptyset)$$

Definition 4.1.9 ($trans_{\mathcal{P}}$)

Sei \mathcal{P} ein Logikprogramm über (Σ, Δ) . Dann definieren wir $trans_{\mathcal{P}} : Pot(At(\Sigma, \Delta, \emptyset)) \rightarrow Pot(At(\Sigma, \Delta, \emptyset))$ mit

$$trans_{\mathcal{P}}(M) = M \cup \{A' \mid \{A', \neg B'_1, \dots, \neg B'_n\} \text{ ist Grundinst. von } \{A, \neg B_1, \dots, \neg B_n\} \in \mathcal{P}, B'_1, \dots, B'_n \in M\}$$

Beispiel 4.1.10 Sei \mathcal{P} wie in Bsp. 4.1.2

$$trans_{\mathcal{P}}(\emptyset) = \{mutterVon(renate, susanne), verheiratet(gerd, rena)\}$$

$$trans_{\mathcal{P}}^2(\emptyset) = \{mutterVon(renate, susanne), verheiratet(gerd, rena)\} \cup \{vaterVon(gerd, susanne)\}$$

$$trans_{\mathcal{P}}^3(\emptyset) = trans_{\mathcal{P}}^2(\emptyset)$$

23.05.06

$$\text{Fixpunkt Semantik: } M_{\mathcal{P}} = \emptyset \cup trans_{\mathcal{P}}(\emptyset) \cup trans_{\mathcal{P}}^2(\emptyset) \cup \dots = \bigcup_{i \in \mathbb{N}} \underbrace{trans_{\mathcal{P}}^i(\emptyset)}$$

*: alle Aussagen (variablen-frei), die man in höchstens i Schritten herleiten kann.

Beispiel 4.1.11 Logikprogramm:

$p(a)$.

$p(f(x)) :- p(x)$.

$$trans_{\mathcal{P}}(\emptyset) = \{p(a)\}.$$

$$trans_{\mathcal{P}}^2(\emptyset) = \{p(a), p(f(a))\}$$

$$trans_{\mathcal{P}}^i(\emptyset) = \{p(a), p(f(a)), \dots, p(f^{i-1}(a))\}$$

$$M_{\mathcal{P}} = \{p(f^i(a)) \mid i \geq 0\}$$

$M_{\mathcal{P}}$ ist ein Fixpunkt von $trans_{\mathcal{P}} : trans_{\mathcal{P}}(M_{\mathcal{P}}) = M_{\mathcal{P}}$.

Es ist sogar der **kleinste** Fixpunkt (bezüglich \subseteq).

$$\Sigma_0 = \{a, b\}, \Sigma_1 = \{f\}, \Delta_1 = \{p\}$$

$$trans_{\mathcal{P}}(At(\Sigma, \Delta, \emptyset)) = At(\Sigma, \Delta, \emptyset)$$

$At(\Sigma, \Delta, \emptyset)$ ist immer ein Fixpunkt. Aber wir definieren die Semantik als **kleinsten** Fixpunkt, da er nur die Formeln enthalten soll, die man wirklich aus \emptyset herleiten kann.

Frage: Führt $M_{\mathcal{P}}$ immer zum kleinsten Fixpunkt?

Antwort: Ja $\rightarrow trans_{\mathcal{P}}$ ist eine stetige Funktion auf einer vollständigen Ordnung.

Ordnung: transitive und antisymmetrische Relation.

\subseteq ist eine **reflexive Ordnung**

- Reflexivität: $M \subseteq M$
- Transitivität: $M_1 \subseteq M_2, M_2 \subseteq M_3 \rightarrow M_1 \subseteq M_3$

- Antisymmetrie: $M_1 \subseteq M_2, M_2 \subseteq M_1 \rightarrow M_1 = M_2$

Lemma 4.1.12 (Vollständigkeit von \subseteq) Die Relation \subseteq ist **vollständig**, d.h.

- es ex. ein kleinstes Element bezüglich \subseteq
- für jede **Kette** $M_0 \subseteq M_1 \subseteq M_2 \subseteq \dots$ existiert eine **kleinste obere Schranke** (least upper bound, lub) $M = \bigcup_{i \geq 0} M_i$

Beweis

- Das kleinste Element ist \emptyset ($\emptyset \subseteq M$ für alle Mengen M).
- M ist obere Schranke: $M_i \subseteq M = \bigcup_{i \geq 0} M_i$
- M ist **kleinste** obere Schranke: Sei m' weitere obere Schranke von $M_0, M_1, \dots \rightarrow M_i \subseteq M'$ für alle i .

$$\rightarrow \underbrace{\bigcup_{i \geq 0} M_i}_M \subseteq M'$$

□

Vollständige Ordnungen bezeichnet man auch als **complete partial order (cpo)**

Wir betrachten die Kette $\emptyset \subseteq \text{trans}_{\mathcal{P}}(\emptyset) \subseteq \text{trans}_{\mathcal{P}}^2(\emptyset) \subseteq \dots$

$M_{\mathcal{P}} = \bigcup_{i \geq 0} \text{trans}_{\mathcal{P}}^i(\emptyset)$ ist also die kleinste obere Schranke dieser Kette.

Lemma 4.1.13 (Monotonie und Stetigkeit)

- Die Funktion $\text{trans}_{\mathcal{P}}$ ist **monoton**, d.h. falls $M_1 \subseteq M_2$, dann ist auch $\text{trans}_{\mathcal{P}}(M_1) \subseteq \text{trans}_{\mathcal{P}}(M_2)$.
(Aus größeren Mengen von Aussagen lassen sich auch mehr Aussagen ableiten)
- Die Funktion $\text{trans}_{\mathcal{P}}$ ist **stetig**, d.h. für jede Kette $M_0 \subseteq M_1 \subseteq \dots$ gilt:
 $\text{trans}_{\mathcal{P}}(\bigcup_{i \geq 0} M_i) = \bigcup_{i \geq 0} \text{trans}_{\mathcal{P}}(M_i)$

Beweis

a) folgt sofort aus der Definition von $\text{trans}_{\mathcal{P}}$

- $\text{trans}_{\mathcal{P}}(\bigcup_{i \geq 0} M_i) \supseteq \bigcup_{i \geq 0} \text{trans}_{\mathcal{P}}(M_i)$, dann $\text{trans}_{\mathcal{P}}(\bigcup_{i \geq 0} M_i) \supseteq \text{trans}_{\mathcal{P}}(M_i)$ wegen Monotonie
(a): $\bigcup_{i \geq 0} M_i \supseteq M_i$
 $\text{trans}_{\mathcal{P}}(\bigcup_{i \geq 0} M_i) \subseteq \bigcup_{i \geq 0} \text{trans}_{\mathcal{P}}(M_i)$
Sei $A' \in \text{trans}_{\mathcal{P}}(\bigcup_{i \geq 0} M_i)$ Dann ist $\{A', \neg B'_1, \dots, \neg B'_n\}$ Grundinstanz einer Klausel aus \mathcal{P} und $B'_1, \dots, B'_n \in \bigcup_{i \geq 0} M_i$.
Da $M_0 \subseteq M_1 \subseteq \dots \rightarrow$ es existiert M_j mit $B'_1, \dots, B'_n \in M_j$
Dann $A' \in \text{trans}_{\mathcal{P}}(M_j) \subseteq \bigcup_{i \geq 0} \text{trans}_{\mathcal{P}}(M_i)$

□

Fixpunktsatz [Tarski/Klenne] Jede stetige Funktion f aus einer vollst. Ordnung hat einen kleinsten Fixpunkt. Diesen erhält man als lub der Kette: kleinstes El. , $f(Kl.El.)$, $f^2(Kl.El.)$, ... Hier: spezielle Formulierung des Fixpunktsatzes für die stetige Funktion $trans_{\mathcal{P}}$ und für die vollständige Ordnung \subseteq

Satz 4.1.14 (Fixpunktsatz)

Für jedes Logikprogramm \mathcal{P} hat $trans_{\mathcal{P}}$ einen kleinsten Fixpunkt (least fixpoint, lfp).

Es gilt $lfp(trans_{\mathcal{P}}) = \bigcup_{i \geq 0} trans_{\mathcal{P}}^i(\emptyset)$

Beweis

1. Zeige dass $\bigcup_{i \geq 0} trans_{\mathcal{P}}^i(\emptyset)$ Fixpunkt von $trans_{\mathcal{P}}$ ist.

$$trans_{\mathcal{P}}(\bigcup_{i \geq 0} trans_{\mathcal{P}}^i(\emptyset)) = \bigcup_{i \geq 0} \underbrace{(trans_{\mathcal{P}}(trans_{\mathcal{P}}^i(\emptyset)))}_{trans_{\mathcal{P}}^{i+1}(\emptyset)} \quad (\text{stetigkeit von } trans_{\mathcal{P}}, \text{ Lemma 4.1.13})$$

(b))

$$= \emptyset \cup \bigcup_{i \geq 0} trans_{\mathcal{P}}^{i+1}(\emptyset)$$

$$= \bigcup_{i \geq 0} trans_{\mathcal{P}}^i(\emptyset)$$

2. Zeige, dass $\bigcup_{i \geq 0} trans_{\mathcal{P}}^i(\emptyset) \subseteq M$ für jeden Fixpunkt M von $trans_{\mathcal{P}}$

Wir zeigen $trans_{\mathcal{P}}^i(\emptyset) \subseteq M$ für alle i durch Ind. über i .

Ind. Anf. $i = 0$ $\emptyset \subseteq M \checkmark$

Ind. Schluss $i > 0$:

Ind. Hypothese: $trans_{\mathcal{P}}^{i-1}(\emptyset) \subseteq M$

$$\rightarrow \underbrace{trans_{\mathcal{P}}(trans_{\mathcal{P}}^{i-1}(\emptyset))}_{trans_{\mathcal{P}}^i(\emptyset)} \subseteq \underbrace{trans_{\mathcal{P}}(M)}_{=M \text{ da } M \text{ ein Fixpunkt ist}} \quad \text{wg. Monotonie von } trans_{\mathcal{P}} \quad (\text{Lemma 4.1.13})$$

a))

□

Definition 4.1.15 (Fixpunkt-Semantik)

Sei \mathcal{P} ein Logikprogramm, $G = \{\neg A_1, \dots, \neg A_k\}$ eine Anfrage. Dann ist die Fixpunkt-Semantik von \mathcal{P} bezüglich G definiert als:

$$F[\mathcal{P}, G] = \{\sigma(A_1 \wedge \dots \wedge A_k) \mid \sigma(A_i) \in lfp(trans_{\mathcal{P}}) \text{ für alle } i\}$$

Satz 4.1.16 (Äquivalenz der Fixpunkt-Semantik zu den anderen Semantiken)

Sei \mathcal{P} ein Logikprogramm, G eine Anfrage. Dann gilt:

$$D[\mathcal{P}, G] = P[\mathcal{P}, G] = F[\mathcal{P}, G]$$

Beweis

1. $P[\mathcal{P}, G] \subseteq F[\mathcal{P}, G]$

Sei $\sigma'(A_1 \wedge \dots \wedge A_k) \in P[\mathcal{P}, G]$

Dann ex. $(G, \emptyset) \vdash_{\mathcal{P}} \dots \vdash_{\mathcal{P}} (\Box, \sigma)$ und $\sigma'(A_1 \wedge \dots \wedge A_k)$ ist Grundinstanz von $\sigma(A_1 \wedge \dots \wedge A_k)$

Zu zeigen ist: $\sigma'(A_i) \in lfp(trans_{\mathcal{P}}) = \bigcup_{i \geq 0} trans_{\mathcal{P}}^i(\emptyset)$

Es reicht zu zeigen: für alle A_i ex. ein $j \geq 0$, so dass $trans_{\mathcal{P}}^j(\emptyset)$ alle Grundinstanzen von $\sigma(A_i)$ enthält.

(geht analog zum Beweis $P[\mathcal{P}, G] \subseteq D[\mathcal{P}, G]$)

2. $F[\mathcal{P}, G] \subseteq D[\mathcal{P}, G]$

Sei $\sigma(A_1 \wedge \dots \wedge A_k) \in F[\mathcal{P}, G] \rightarrow \sigma(A_i) \in lfp(trans_{\mathcal{P}})$ für alle i .

Zu zeigen: $\mathcal{P} \models \sigma(A_i)$

Zeige stattdessen: Für alle $A' \in trans_{\mathcal{P}}^j(\emptyset)$ gilt $\mathcal{P} \models A'$ mit Induktion über j .

Ind. Anfj $= 0 \checkmark$ ($trans_{\mathcal{P}}^0(\emptyset) = \emptyset$)

Ind. Schluss $j > 0$

$A' \in trans_{\mathcal{P}}(trans_{\mathcal{P}}^{j-1}(\emptyset)) \rightarrow A' \in trans_{\mathcal{P}}^{j-1}(\emptyset)$ ($\rightarrow \mathcal{P} \models A'$ wegen Ind.Hyp.)

oder es ex. eine Grundinstanz $\{A', \neg B'_1, \dots, \neg B'_n\}$ einer Klausel aus \mathcal{P} mit $B'_1, \dots, B'_n \in trans_{\mathcal{P}}^{j-1}(\emptyset)$

I.H. $\mathcal{P} \models B'_1, \dots, \mathcal{P} \models B'_n \rightarrow \mathcal{P} \models A'$

□

26.05.06

4.2 Universalität der Logikprogrammierung

Logikprogrammierung ist eine vollwertige Programmiersprache, man kann jede berechenbare Funktion auch durch ein Logikprogramm berechnen. (Die Logikprogrammierung ist Turing-vollständig Einschränkung auf arithmetische Funktionen $f : \mathbb{N}^n \rightarrow \mathbb{N}$. Andere Datenstrukturen kann man durch Abbildungen in \mathbb{N} codieren.

Was sind berechenbare Funktionen?:

Turing: alles, was mit Turing-Maschinen berechenbar ist.

Church: alles, was mit dem dem λ -Kalkül berechenbar ist

Kleene: alle μ -rekursiven Funktionen.

Diese 3 Aussagen sind äquivalent

Church'sche These: Dieses ist die Menge der im intuitiven Sinne berechenbaren Funktionen

Definition 4.2.1 (μ -rekursive Funktionen)

Die Klasse der μ -rekursiven Funktionen ist die kleinste Klasse arithmetischer Funktionen mit:

1. Für jedes $n \in \mathbb{N}$ ist die Funktion $null_n : \mathbb{N}^n \rightarrow \mathbb{N}$ mit $null_n(k_1, \dots, k_n) = 0$ μ -rekursiv.
2. Die Nachfolgerfunktion $succ : \mathbb{N} \rightarrow \mathbb{N}$ mit $succ(k) = k + 1$ ist μ -rek.
3. Für jedes $n \geq 1$ und jedes $1 \leq i \leq n$ ist die Projektionsfunktion $proj_{n,i}(k_1, \dots, k_n) = k_i$ μ -rek.
4. μ -rekursive Funktionen sind unter **Komposition** abgeschlossen.
Falls $f : \mathbb{N}^m \rightarrow \mathbb{N}$, $f_1 : \mathbb{N}^n \rightarrow \mathbb{N}, \dots, f_m : \mathbb{N}^n \rightarrow \mathbb{N}$ μ -rekursiv sind, dann ist auch $g : \mathbb{N}^n \rightarrow \mathbb{N}$ μ -rekursiv mit:
$$g(k_1, \dots, k_n) = f(f_1(k_1, \dots, k_n), \dots, f_m(k_1, \dots, k_n))$$
5. μ -rekursive Funktionen sind unter **primitiver Rekursion** abgeschlossen:
Falls $f : \mathbb{N}^n \rightarrow \mathbb{N}$, $g : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ μ -rekursiv, dann ist auch $h : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ μ -rekursiv mit:
$$h(k_1, \dots, k_n, 0) = f(k_1, \dots, k_n)$$

$$h(k_1, \dots, k_n, k + 1) = g(k_1, \dots, k_n, h(k_1, \dots, k_n, k))$$

6. μ -rekursive Funktionen sind unter **Minimalisierung** abgeschlossen:
 Falls $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ μ -rekursiv ist, dann ist auch $g : \mathbb{N}^n \rightarrow \mathbb{N}$ μ -rekursiv mit:
 $g(k_1, \dots, k_n) = k$ gdw. $f(k_1, \dots, k_n, k) = 0$ und für alle $0 \leq k' < k$ ist $f(k_1, \dots, k_n, k')$ definiert,
 aber $f(k_1, \dots, k_n, k') > 0$
 (wenn $f(k_1, \dots, k_n, k) \neq 0$ für alle k , dann ist $g(k_1, \dots, k_n)$ nicht definiert)

Funktionen mit 1-5: **primitiv rekursive** Funktionen

Es existieren berechenbare Funktionen, die nicht primitiv-rekursiv sind:

- alle partiellen berechenbaren Funktionen
- es ex. auch totale berechenbare Funktionen die nicht primitiv rekursiv sind (Ackermann-Funktion)

Beispiel 4.2.2 • Addition

$plus : \mathbb{N}^2 \rightarrow \mathbb{N}$ ist primitiv rekursiv:

$$plus(x, 0) = proj_{1,1}(x) \leftarrow x$$

$$plus(x, y + 1) = f(x, y, plus(x, y)) \leftarrow (x, y) + 1$$

$$f(x, y, z) = succ(proj_{3,3}(x, y, z))$$

- Multiplikation

$$times(x, 0) = null_1(x) \leftarrow 0$$

$$times(x, y + 1) = g(x, y, times(x, y)) \ plus(x, times(x, y))$$

$$g(x, y, z) = plus(proj_{3,1}(x, y, z), proj_{3,3}(x, y, z)) \ plus(x, z)$$

- Predecessor:

$$p(0) = null_0$$

$$p(x + 1) = proj_{2,1}(x, p(x))$$

- Subtraktion $minus(x, y) = \begin{cases} 0 & x \leq y \\ x - y & \text{sonst} \end{cases}$

$$minus(x, 0) = proj_{1,1}(x)$$

$$minus(x, y + 1) = h(x, y, minus(x, y)) \leftarrow p(minus(x, y))$$

$$h(x, y, z) = p(proj_{3,3}(x, y, z))$$

- Division:

$$div(0, 0) = 0$$

$$div(x + 1, 0) = undef.$$

$$div(x, y) = \lceil \frac{x}{y} \rceil, \text{sonst}$$

$$div(x, y) = z \text{ gdw. } i(x, y, z) = 0 \text{ und für alle } 0 \leq z' < z \text{ ist } i(x, y, z') \text{ definiert und } i(x, y, z') > 0$$

$$i(x, y, z) = minus(proj_{3,1}(x, y, z), j(x, y, z)) \leftarrow x - y \cdot z$$

$$j(x, y, z) = times(proj_{3,2}(x, y, z), proj_{3,3}(x, y, z)) \leftarrow y \cdot z$$

Darstellung von \mathbb{N} in Logikprogrammen? \leftarrow Logikprogramm arbeitet auf **Termen**

Berechnung von Funktionen in Logikprogrammen? \leftarrow Logikprogramm definiert nur **Relationen**

Definition 4.2.3 (Berechnung arithmetischer Funktionen durch Logikprogramme)

- Jede Zahl $k \in \mathbb{N}$ wird durch den Term $\underline{k} \in \mathcal{T}(\Sigma, \mathcal{V})$ mit $\underline{k} = \underbrace{s(\dots s(0))}_{k \text{ Stück}}$ dargestellt, wobei

$$0 \in \Sigma_0, s \in \Sigma_1.$$

$$(\underline{0} = 0, \underline{1} = s(0) \dots)$$

- Ein Logikprogramm \mathcal{P} über (Σ, Δ) **berechnet** $f : \mathbb{N}^n \rightarrow \mathbb{N}$ gdw. es eine Prädikatssymbol $\underline{f} \in \Delta_{n+1}$ gibt, so dass $f(k_1, \dots, k_n) = k$ gdw. $\mathcal{P} \models \underline{f}(\underline{k}_1, \dots, \underline{k}_n)$
Grund: Dann kann man f durch Anfragen an \mathcal{P} berechnen lassen:
Um $f(k_1, \dots, k_n)$ zu berechnen, stelle folgende Anfrage: $? - \underline{f}(\underline{k}_1, \dots, \underline{k}_n, X)$

Beispiel 4.2.4

$\underline{plus}(X, 0, X).$

$\underline{plus}(X, s(Y), s(Z)) : \neg \underline{plus}(X, Y, Z).$

$\underline{times}(X, 0, 0).$

$\underline{times}(X, s(Y), Z) : \neg \underline{times}(X, Y, U), \underline{plus}(X, U, Z).$

Satz 4.2.5 (Universalität der Logikprogrammierung)

Jede μ -rekursive Funktion ist durch ein Logikprogramm berechenbar.

Beweis

Induktion über Aufbau der Klasse der μ -rekursiven Funktion.

1. $\underline{null}_n(X_1, \dots, X_n, 0)$
2. $\underline{succ}(X, s(X)).$
3. $\underline{proj}_{n,i}(X_1, \dots, X_n, X_i).$
4. *Es ex. $\underline{f}, \underline{f}_1, \dots, \underline{f}_m$ - Klauseln. Ergänze Logikprogramm um*

$$\underline{g}(X_1, \dots, X_n, Z) : \neg \underline{f}_1(X_1, \dots, X_n, Z_1), \dots, \underline{f}_n(X_1, \dots, X_n, Z_m),$$

$$\underline{f}(Z_1, \dots, Z_m, Z).$$

5. *Es ex. $\underline{f}, \underline{g}$ -Klauseln.*
Ergänze Logikprogramm um:
 $\underline{h}(X_1, \dots, X_n, 0, Z) : \neg \underline{f}(X_1, \dots, X_n, Z).$
 $\underline{h}(X_1, \dots, X_n, s(X), Z) : \neg \underline{h}(X_1, \dots, X_n, X, Y), \underline{g}(X_1, \dots, X_n, X, Y, Z).$

6. *Es ex. \underline{f} -Klauseln.*
Ergänze Logikprogramm um:
 $\underline{g}(X_1, \dots, X_n, Z) : \neg \underline{f}(X_1, \dots, X_n, Z, 0, \underline{f}'(X_1, \dots, X_n, Z)). \leftarrow \text{wahr, falls für alle } 0 \leq Z' < Z$
 $\underline{f}(X_1, \dots, X_n, Z')$ def. ist und $f(X_1, \dots, X_n, Z') > 0$
 $\underline{f}'(X_1, \dots, X_n, 0).$
 $\underline{f}'(X_1, \dots, X_n, s(X)) : \neg \underline{f}'(X_1, \dots, X_n, X), \underline{f}(X_1, \dots, X_n, X, s(Y)).$

Beispiel 4.2.6 $\underline{proj}_{3,3}(X, Y, Z, Z)$.

$\underline{succ}(X, s(X))$.

$\underline{f}(X, Y, Z,) : -\underline{proj}_{3,3}(X, Y, Z, Z_1), \underline{succ}(Z_1, Z_2)$.

$\underline{proj}_{1,1}(X, X)$.

$\underline{plus}(X, 0, Z) : -\underline{proj}_{1,1}(X, Z)$.

$\underline{plus}(X, s(Y), Z_2) : -\underline{plus}(X, Y, Z_1), \underline{f}(X, Y, Z_1, Z_2)$

$\underline{div}(X, Y, Z) : -\underline{i}(X, Y, Z, 0), \underbrace{\underline{i}'(X, Y, Z)}_{(*)}$.

$\underline{i}'(X, Y, 0)$.

$\underline{i}'(X, Y, s(Z)) : -\underline{i}'(X, Y, Z), \underline{i}(X, Y, Z), \underline{i}(X, Y, Z, s(U))$.

(*) wahr, falls für alle $Z < Z'$ $\underline{i}(X, Y, Z') > 0$ d.h. $\underline{i}(X, Y, Z', s(U))$ ist wahr.

4.3 Indeterminismus und Auswertungsstrategien

Ausführung von Logikprogrammen funktioniert wie in der prozeduralen Semantik. Falls and das Logikprogramm \mathcal{P} die Anfrage G gestellt wird, dann versucht man eine erfolgreiche Berechnung $(G, \emptyset) \vdash_{\mathcal{P}}^+ (\square, \sigma)$ zu finden. Die Ausgabe ist die Antwortsubstitution σ eingeschränkt aus Variablen aus G .

$\vdash_{\mathcal{P}}$ hat in jedem Schritt $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ zwei Arten von Indeterminismen.

Indeterminismus 1. Art: Wahl der Programm-Klausel, mit der resolviert wird

Indeterminismus 2. Art: Wahl des Literals aus G_1 , das zu Resolution verwendet wird.

Beispiel 4.3.1 Sei \mathcal{P} folgendes Logikprogramm

`mutterVon(renate,susanne).`

`mutterVon(susanne,aline).`

`vorfahre(V,X) :- mutterVon(V,X).`

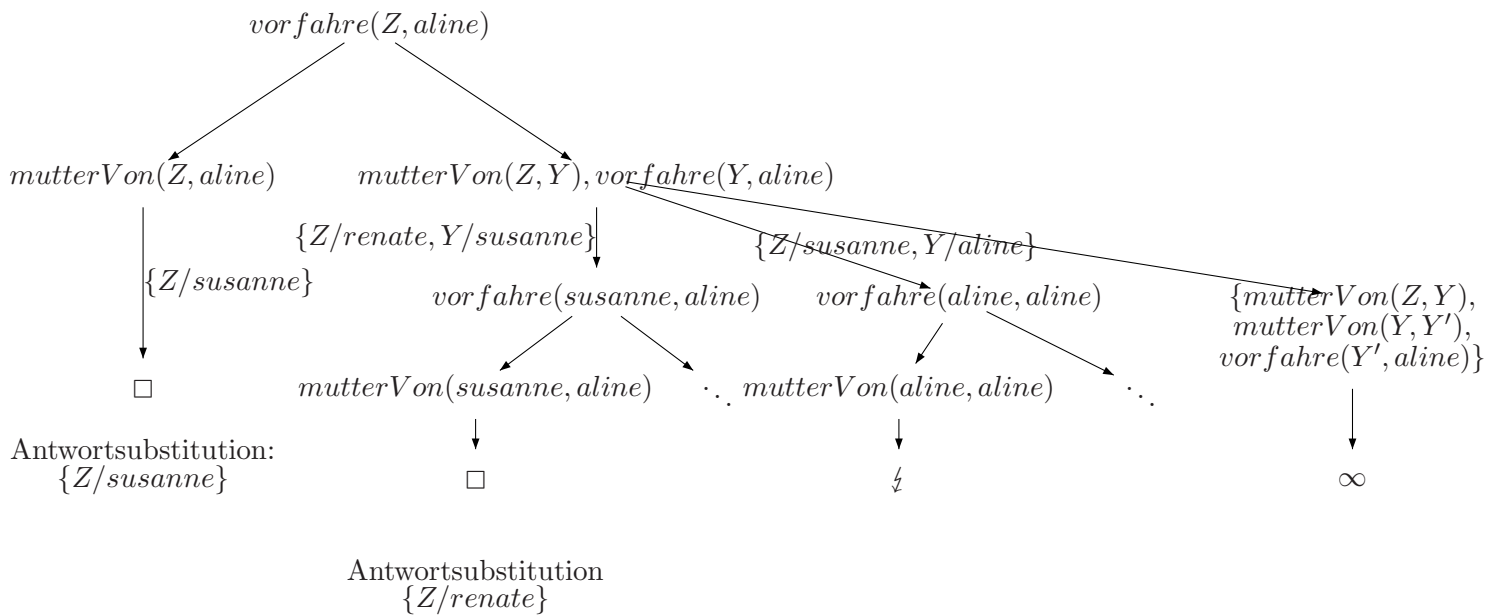
`vorfahre(V,X) :- muterVon(V,Y), vorfahre(Y,X).`

Anfrage: ? – `vorfahre(Z,aline)`.

$(\{\neg \text{vorfahre}(Z, \text{aline})\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg \text{mutterVon}(Z, \text{aline})\}, \{V/Z, X/\text{aline}\})$

$(\{\neg \text{vorfahre}(Z, \text{aline})\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg \text{mutterVon}(Z, Y), \neg \text{vorfahre}(Y, \text{aline})\}, \{V/Z, X/\text{aline}\})$

\Rightarrow Indeterminismus 1. Art.



Dieser Baum entsteht wie folgt:

- statt $(\{\neg A_1, \dots, \neg A_k\}, \sigma)$ schreibe A_1, \dots, A_k in die Knoten.
- markiere Kanten mit den in der Resolution verwendeten mgu's, eingeschränkt auf die Variablen der Anfrage.

Dieser Baum hat

- erfolgreiche Pfade(\square) z.T. mit unterschiedlichen Antwortsstitutionen
- im Endlichen erfolglose Pfade(ζ)
- unendliche Pfade

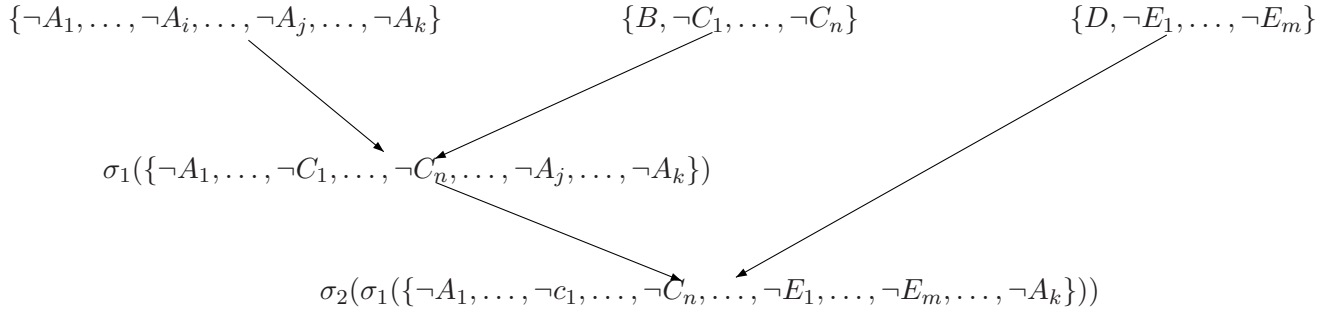
Auflösung der Indeterminismen beeinflusst, welcher Pfad im Baum zuerst betrachtet wird \Rightarrow beeinflusst das Programmverhalten

Indeterminismus 2. Art ist "harmlos": Wenn man ihn auflöst (indem man z.B. nur Resolutionen mit dem 1. Literal einer Anfrage zulässt), dann entsteht der sogenannte SLD-Baum.

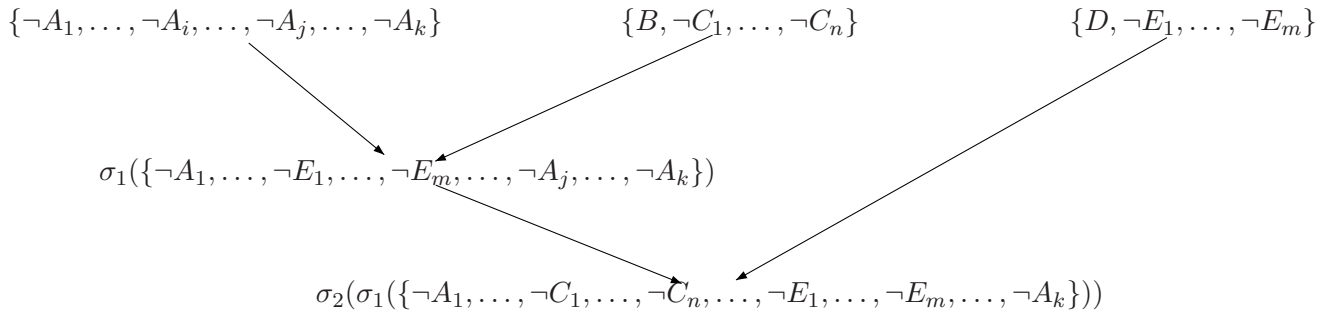
Dieser Baum enthält immer noch jede erfolgreiche Berechnung mit jeder Antwortsstitution, d.h. falls es eine Herleitung von \square mit einer Antwortsstitution σ gibt, dann gibt es auch eine Herleitung von \square mit der Antwortsstitution σ' im SLD-Baum, σ und σ' unterscheiden sich nur durch Variablenumbenennung. Grund: Anfrage $G = \{\neg A_1, \dots, \neg A_n\}$ kann man Resolutionsschritte die erst mit $\neg A_i$ und dann mit $\neg A_j$ resolvieren, vertauschen.

Lemma 4.3.2 (Vertrauschungslemma) Seien $\{\neg A_1, \dots, \neg A_k\}, \{B, \neg C_1, \dots, \neg C_n\}, \{D, \neg E_1, \dots, \neg E_m\}$ paarweise variablendisjunkt.

Sei σ_1 mgu von A_i und B , sei σ_2 mgu von $\sigma_1(A_j)$ und D . Dann sind folgende SLD-Resolutionsschritte möglich:



Dann existiert auch ein mgu σ'_1 von A_j und D und ein mgu σ'_2 mgu von $\sigma'_1(A_i)$ und B , so dass:



Dabei unterscheiden sich $\sigma_1 \circ \sigma_1$ und $\sigma'_2 \circ \sigma'_1$ nur durch Variablenumbenennungen, d.h. $\sigma'_2 \circ \sigma'_1 = \nu \circ \sigma_2 \circ \sigma_1$ für eine Variablenumbenennung ν .

Beweis

$\sigma_1(A_j)$ und D haben mgu σ_2 ($D = \sigma_1(D)$, da Klauseln variablendisjunkt).

d.h. $\sigma_2 \circ \sigma_1$ ist Unifikator von A_j und D . Dann haben A_j und D auch einen mgu σ'_1 und es ex. eine Substitution σ mit $\sigma_2 \circ \sigma_1 = \sigma \circ \sigma_1$ (*).

Jetzt muss man zeigen, dass $\sigma'_1(A_i)$ und B unifizierbar sind.

Dies Gilt, denn σ ist ihr Unifikator:

$$\begin{aligned}
 \sigma(\sigma'_1(A_i)) &= \sigma_2(\underbrace{\sigma_1(A_i)}_{\sigma_1(B)}) \text{ wegen } (*) \\
 &= \sigma_2(\sigma_1(B)) \\
 &= \sigma(\underbrace{\sigma'_1(B)}_B) \text{ wegen } (*) \\
 &= \sigma(B)
 \end{aligned}$$

Dann haben $\sigma'_1(A_i)$ und B auch einen mgu σ'_2

Noch zu zeigen $\sigma_2 \circ \sigma_1$ und $\sigma'_2 \circ \sigma'_1$ sind bis auf Variablenumbenennung gleich.

Hierfür reicht es zu zeigen: (a) $\sigma_2 \circ \sigma_1 = \delta \circ \sigma'_2 \circ \sigma'_1$

(b) $\sigma'_2 \circ \sigma'_1 = \delta' \circ \sigma_2 \circ \sigma_1$

d.h. die beiden Substitutionen sind jeweils Instanzen voneinander.

δ' kann man so erweitern, dass δ' eine Variablenumbenennung ist.

Zeige nur (a), (b) ist analog.

$$\begin{aligned} \sigma_2 \circ \sigma_1 &= \sigma \circ \sigma'_1 \text{ wegen } (*) \\ &= \delta \circ \sigma'_2 \circ \sigma'_1 \end{aligned}$$

2.6.05

Beispiel 4.3.3 (Illustration des Vertauschungslemmas)

$p(Z, Z) :- r(Z).$

$q(W).$

Anfrage: ? - $p(X, Y), q(X).$

$$\begin{aligned} (\{\neg p(X, Y), \neg q(X)\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg r(Z), \neg q(Z)\}, \{X/Z, Y/Z\}) \vdash_{\mathcal{P}} (\{\neg r(Z)\}, \underbrace{\{W/Z, X/Z, Y/Z\}}_{\sigma_2}) \\ (\{\neg p(X, Y), \neg q(X)\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg p(W, Y)\}, \{X/W\}) \vdash_{\mathcal{P}} (\{\neg r(Y)\}, \underbrace{\underbrace{\{W/Y, Z/Y\}}_{\sigma'_2} \circ \underbrace{\{X/\bar{W}\}}_{\sigma'_1}}_{\sigma_2 \circ \sigma_1}) \\ \underbrace{\qquad\qquad\qquad}_{\{W/Y, Z/Y, X/Y\}} \end{aligned}$$

$$\begin{aligned} \text{Ergebnis ist: } \sigma_2 \circ \sigma_1(r(Z)) \\ \sigma'_2 \circ \sigma'_1(r(Z)) \end{aligned}$$

Antwortsubstitutionen sind gleich bis auf Variablenumbenennung ν :

$$\sigma'_2 \circ \sigma'_1 = \nu \circ \sigma_2 \circ \sigma_1 \text{ wobei } \nu = \{Y/Z, Z/Y\}$$

Vertauschungslemma: Man kann beliebige Ordnung der Literale wählen und bei der Resolution immer das "erste" Literal in der Anfrage bearbeiten. \Rightarrow Man kann beliebige Selektionsfunktionen (SLD) zur Auswahl des Literals verwenden.

\Rightarrow Betrachte Klauseln als **Folgen** von Literalen und verwende Selektionsfunktion, die das linkeste Literal der Anfrage auswählt.

Definition 4.3.4 (Kanonische Berechnung)

Eine Berechnung $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \dots$ heißt **kanonisch**, wenn in jedem Resolutionsschritt mit dem ersten Literal der jeweiligen Anfrage G_i resolviert wird.

Um zu zeigen, dass man sich auf Kanonische Berechnungen einschränken kann, brauchen wir folgendes Lemma: Wenn sich zwei Anfragen nur durch Variablenumbenennung unterscheiden, dann kann man mit ihnen die gleichen Berechnungen durchführen und erhält die gleiche Antwortsubstitution (bis auf Variablenumbenennung).

Lemma 4.3.5 Sei $l \in \mathbb{N}$ Falls $(G, \sigma) \vdash_{\mathcal{P}}^l (G', \sigma')$ und ν eine Variablenumbenennung ist, dann $(\nu(G), \nu \circ \sigma) \vdash_{\mathcal{P}}^l (\nu(G'), \nu \circ \sigma')$

Beweis

Induktion über l :

Induktions-Anfang: $l = 0$. Trivial ($G' = G, \sigma' = \sigma$)

Induktions-Schluss: $l \geq 1$

$(G, \sigma) \vdash_{\mathcal{P}} (H, \delta \circ \sigma) \vdash_{\mathcal{P}}^{l-1} (G', \sigma')$

Induktions-Hypothese: $(\nu(H), \nu \circ \delta \circ \sigma) \vdash_{\mathcal{P}}^{l-1} (\nu(G'), \nu \circ \sigma)$

Zu Zeigen: $(\nu(G), \nu \circ \sigma) \vdash_{\mathcal{P}} (\nu(H), \nu \circ \delta \circ \sigma)$

Wir haben: $G = \{\neg A_1, \dots, \neg A_k\}$

Es existiert eine Variablenumbenannte Programmklauseel (Variablen-disjunkt mit G und $\nu(G)$).
und $H = \delta(\{\neg A_1, \dots, \neg C_1, \dots, \neg C_n, \dots, \neg A_k\})$ mit $\delta = \text{mgu}(A_i, B)$

Es genügt zu zeigen dass $\nu \circ \delta \circ \nu^{-1} = \text{mgu}(\nu(A_i), B)$ (*)

Denn dann ergibt die Resolution von $\nu(G)$ und $\{B, \neg C_1, \dots, \neg C_n\}$:

$\nu(\delta(\nu^{-1}(\{\nu(\neg A_1), \dots, \neg C_1, \dots, \neg C_n, \dots, \nu(\neg A_k)\}))) = \nu(\underbrace{\delta(\{\neg A_1, \dots, \neg C_1, \dots, \neg C_n, \dots, \neg A_k\})}_H) =$

$\nu(H)$

$\rightarrow (\nu(G), \nu \circ \sigma) \vdash_{\mathcal{P}} (\nu(H), \nu \circ \delta \circ \nu^{-1} \circ \nu \circ \sigma)$

((*) ist nicht schwer zu zeigen (Übung)).

Beispiel 4.3.6 (Illustration von Lemma 4.3.5)

$p(Z, Z) \text{ .- } r(Z)$.

$q(W)$.

$(\{\neg p(X, Y), \neg q(X)\}, \sigma) \vdash_{\mathcal{P}} (\{\neg r(Y), \neg q(Y)\}, \{X/Y, Z/Y\} \circ \sigma)$

Sei $\nu = \{X/Y, Y/U, U/X\}$

$(\underbrace{\nu(\{\neg p(X, Y), \neg q(X)\})}_{\{\neg p(Y, U), \neg q(Y)\}}, \nu \circ \sigma) \vdash_{\mathcal{P}} (\underbrace{\nu(\{\neg r(Y), \neg q(Y)\})}_{\{\neg r(U), \neg q(U)\}}, \underbrace{\nu(\{X/Y, Z/Y\} \circ \sigma)}_{\{X/U, Y/U, Z/U, U/X\} \circ \sigma})$

Satz 4.3.7 (Auflösung des Indeterminismus 2. Art)

Sei \mathcal{P} ein Logikprogramm und G eine Anfrage. Dann existiert zu jeder Berechnung $(G, \emptyset) \vdash_{\mathcal{P}}^+$ (\square, σ) eine kanonische Berechnung $(G, \emptyset) \vdash_{\mathcal{P}}^+ (\square, \sigma')$ gleicher Länge, wobei sich σ und σ' nur durch Variablenumbenennung unterscheiden.

Beweis

Wir haben eine Berechnung $(G, \emptyset) \vdash_{\mathcal{P}} (G_1, \delta_1) \vdash_{\mathcal{P}} (G_2, \delta_2 \circ \delta_1) \vdash_{\mathcal{P}} \dots \vdash_{\mathcal{P}} (\square (= G_l), \delta_l \circ \dots \circ \delta_1)$

Induktion über die Länge j der ersten nicht-kanonischen Teilberechnungsfolge.

Induktions-Anfang: $j = 0$. \rightarrow Berechnung ist bereits kanonisch \checkmark

Induktions-Schluss: $j \geq 1$ Erster nicht-kanonischer Schritt sei $(G_i, \delta_i \circ \dots \circ \delta_1) \vdash_{\mathcal{P}} (G_{i+1}, \delta_{i+1} \circ \delta_i \circ \dots \circ \delta_1)$

Da man das erste Literal irgendwann durch Resolution eliminieren muss, um \square zu bekommen, muss später noch einmal ein kanonischer Schritt kommen.

$$i + j < l$$

nicht kanonisch: $\vdash_{\mathcal{P}}^{j-1} (G_{i+j-1}, \delta_{i+j-1} \circ \dots \circ \delta_1)$

$$\begin{aligned}
\text{nicht kanonisch: } & \vdash_{\mathcal{P}}^{\langle} G_{i+j}, \delta_{i+j} \circ \dots \circ \delta_1 \rangle (*) \\
\text{kanonisch: } & \vdash_{\mathcal{P}}^{\langle} G_{i+j+1}, \delta_{i+j+1} \circ \dots \circ \delta_1 \rangle (**) \\
& \vdash_{\mathcal{P}}^{l-i-j-1} (\square, \sigma)
\end{aligned}$$

Wende Vertauschungslemma 4.3.2 an, um Schritte (*) und (**) zu vertauschen.

$\rightarrow (G_{i+j-1}, \delta_{i+j-1} \circ \dots \circ \delta_1) \vdash_{\mathcal{P}}^2 (\nu(G_{i+j+1}), \nu \circ \delta_{i+j+1} \circ \dots \circ \delta_1)$ wobei der erste Berechnungsschritt kanonisch ist.

Nach Lemma 4.3.5: $(\nu(G_{i+j+1}), \nu \circ \delta_{i+j+1} \circ \dots \circ \delta_1) \vdash_{\mathcal{P}}^{l-i-j-1} (\square, \nu \circ \delta)$

D.h.

$$\begin{aligned}
\text{kanonisch: } & \vdash_{\mathcal{P}}^i (G_i, \delta_i \circ \dots \circ \delta_1) \\
\text{nicht kanonisch: } & \vdash_{\mathcal{P}}^{j-1} (G_{i+j-1}, \delta_{i+j-1} \circ \dots \circ \delta_1) \\
\text{nicht kanonisch: } & \vdash_{\mathcal{P}}^2 (\nu(G_{i+k+1}), \nu \circ \delta_{i+j+1} \circ \dots \circ \delta_1) \\
& \vdash_{\mathcal{P}}^{l-i-j-1} (\square, \nu \circ \sigma)
\end{aligned}$$

Lemma folgt jetzt aus der Induktions-Hypothese. Für den Fall $j = 1$ ist eine extra Betrachtung notwendig. Zeige durch Induktion, dass man den nicht-kanonischen Schritt immer weiter nach hinten verschieben kann.

Besser wäre eine Induktion über die Länge der ersten kanonischen Teilfolge.

Beispiel 4.3.8 (Einschränkung von Beispiel 4.3.1 auf kanonische Berechnungen) \rightarrow alle Lösungen werden immer noch gefunden, aber unendliche Pfade fehlen.

\Rightarrow der Indeterminismus 2. Art beeinflusst das Terminierungsverhalten.

Der Baum der durch Einschränkung auf kanonische Berechnungen entsteht, heißt **SLD-Baum**

Die Reihenfolge der Kinder entspricht der Reihenfolge der Programmklauseln.

3.6.06

Beispiel 4.3.9

$p :- p.$
 $q(a).$

Anfrage: $? - q(b), p.$ Terminiert mit Fehlschlag, wenn man mit dem linken Literal beginnt.

Terminiert nicht, wenn man mit dem rechten Literal beginnt.

\rightarrow der Indeterminismus 1. Art beeinflusst das Terminierungsverhalten, aber nicht die Vollständigkeit des Ableitungsbaumes.

Definition 4.3.10 (SLD-Baum)

Sei \mathcal{P} ein Logikprogramm, G eine Anfrage. Dann ist der SLD-Baum von \mathcal{P} bei Anfrage G ein endlicher oder unendlicher Baum, dessen Knoten mit Folgen von atomaren Formeln markiert sind und dessen Kanten mit Substitutionen markiert sind. Der SLD-Baum ist der kleinste Baum mit

- Falls $G = \{\neg A_1, \dots, \neg A_k\}$, dann ist die Wurzel mit A_1, \dots, A_k markiert.

- Sei ein Knoten mit B_1, \dots, B_n markiert, sei B_1 mit den positiven Literalen von k Programmklauseln K_1, \dots, K_k unifizierbar, wobei die Klauseln in dieser Reihenfolge im Programm sind. Dann hat der Knoten k Nachfolger. Der i -te Nachfolger ist mit den Atomen markiert, die sich bei einem kanonischen Berechnungsschritt durch Resolution mit der Klausel K_i ergeben. Falls die Berechnung also die Gestalt $(\{\neg B_1, \dots, \neg B_n\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg C_1, \dots, \neg C_m\}, \sigma)$ hat, so ist der i -te Nachfolgerknoten mit C_1, \dots, C_m markiert und die Kante ist mit σ eingeschränkt auf die Variablen in B_1, \dots, B_n markiert.

Ablesen von Antwortsubstitution aus erfolgreichen Pfaden (enden mit \square): Wenn Pfad mit $\delta_1, \delta_2, \dots, \delta_l$ beschriftet ist, dann ist die Antwortsubstitution $\delta_l \circ \dots \circ \delta_1$ eingeschränkt auf Variablen aus G .

Neben Erfolgreichen Pfaden kann es noch

- Pfade mit endlichem Fehlschlag (enden mit Anfragen deren erstes Atom mit keiner Programmklausele resolvierbar ist).
- unendliche Pfade.

Der SLD-Baum löst den Indeterminismus 2. Art auf.

repräsentiert den Indeterminismus 1. Art: Reihenfolge der Kinder $\hat{=}$ Reihenfolge der Program

Auflösung des Indeterminismus 1. Art: Auswertungsstrategie, die angibt, wie der SLD-Baum zu durchsuchen/aufzubauen ist.

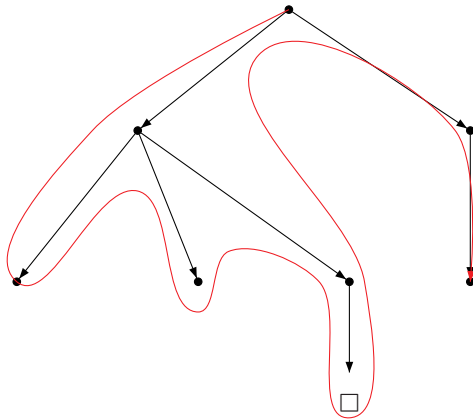
Unterscheide dann, ob man an **einer** oder an **allen** Lösungen interessiert ist.

Breitensuche ist vollständig: jede erfolgreiche Berechnung wird irgendwann gefunden.

Nachteil: ineffizient.

Tiefensuche ist unvollständig: nicht jede erfolgreiche Berechnung wird gefunden, wenn es unendliche Pfade gibt.

Vorteil: kann effizienter sein.



Prolog verwendet Tiefensuche wobei der linkeste Pfad zuerst betrachtet wird.

Bei Eingabe von ; oder bei Fehlschlag (Blatt $\neq \square$) findet Backtracking statt.

\Rightarrow Programmierer sollte Anordnung von Literalen in Klauseln und von Klauseln im Programm "geschickt" wählen.

Vertauschen von Literalen kann Effizienz und Terminierung beeinflussen.

Beispiel 4.3.11 Vertauschung der Literale in rekursiver *vorfahre*-Klausel: \rightarrow findet immer noch beide Lösungen aber wenn man danach ";" eingibt \rightarrow nicht-Terminierung.

Beispiel 4.3.12 Vertauschung der letzten beiden Klauseln: \rightarrow Erste Anfrage terminiert nicht.

\Rightarrow nicht-rekursive Klauseln eines Prädikats sollten vor den rekursiven Klauseln des Prädikats kommen.

- Indeterminismus 1. Art: Auflösung in Prolog: bearbeite Programm-Klauseln von oben nach unten
- Indeterminismus 2. Art: Auflösung in Prolog: bearbeite Literale von links nach rechts.

Dies ist kein ganz reine deklarative Programmierung; Programmierer sollte dies beachten (wg. Effizienz und Terminierung).

5 Die Programmiersprache Prolog

Bekannteste Sprache die auf Logikprogrammierung beruht (erste Hälfte der 70er Jahre, Kowalski + Colmeraner).

Popularität in der Künstlichen Intelligenz: Hauptsprache des japanischen "Fifth Generation Project" (1981).

Syntax von (einfachem) Prolog: Syntax von Logikprogrammen mit ":-", "?-".

Signatur ergibt sich aus den darin auftretenden Funktions- und Prädikatssymbolen. (beginnen mit Kleinbuchstaben oder Strings aus Sonderzeichen (< -- >) oder Strings in Apostrophen 'X').

Variablen beginnen mit Großbuchstaben oder _

Besonderheit _ anonyme Variable

- mehrfache Vorkommen von _ dürfen unterschiedlich belegt werden.
- Belegung von _ werden bei der Antwortsstitution nicht ausgegeben.

Bsp: Programm:

$p(a,b,c).$

Anfrage: $? - p(_, _, X).$ → Antwortsstitution: $X = c$

Überladen von Funktions- und Prädikatssymbolen möglich:

$p(a,b,c).$

$p(a,p(b,c)).$

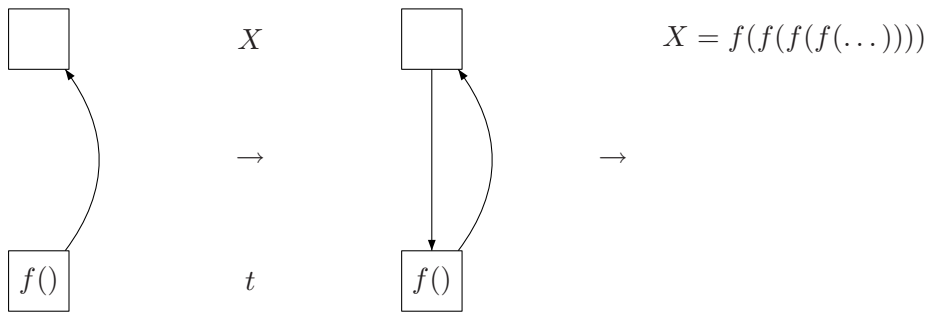
$p/3$ und $p/2$ haben nichts miteinander zu tun.

Semantik von (einfachem) Prolog: Semantik von Logikprogrammen, wobei der SLD-Baum in Tiefensuche von links nach rechts durchlaufen wird.

Aber: keine Korrekte Unifikation, sondern Unifikation **ohne Occur Check** (aus Effizienzgründen).

Wenn X mit t unifiziert werden sollen ($X \neq T$), dann überprüft Prolog nicht, ob X in t auftritt.

Speicherzelle für X zeigt dann auf Speicherzelle für t :



X und $f(X)$ hat mgu $\{X/f(f(f(f(\dots))))\}$ wenn man ∞ viele Terme zulässt.

Programm: $p(X, f(X))$.

Anfrage: $? - p(X, X)$.

Antwortsubst: $X = f(f(\dots))$

Es existiert ein vordefiniertes Prädikat `unify_with_occur_check/2`:

? - `unify_with_occur_check(X, f(X))` → No

? - `unify_with_occur_check(X, f(Y))` → $X = f(Z), Y = Z$

Folgende Abschnitte: Eigenschaften von Prolog, die über reine Logikprogrammierung hinausgehen.

5.1 Arithmetik

Logikprogramme arbeiten auf Termen → Datenobjekte müssen als Terme dargestellt werden.
Natürliche Zahlen z.B. darstellbar über

- $0 \in \Sigma_0, s \in \Sigma_1$
 $0 \hat{=} 0, s(0) \hat{=} 1 \dots$
 Addition: $add(X, Y, Z) \hat{=} X + Y = Z$
 2-stellige Funktion wird durch 3-stelliges Prädikat dargestellt.
 $add(X, 0, X)$.
 $add(X, s(Y), s(Z)) : -add(X, Y, Z)$.

Nachteil:

- Lesbarkeit ($s(s(\dots(s(0))\dots))$)
- Effizienz

⇒ Prolog bietet Unterstützung für Zahlen und Listen.

Vorteile: Bidirektionalität (Ein- und Ausgabe ist nicht festgelegt):

? - $add(s(0), s(s(0)), X)$ → $X = s(s(s(0)))$ Berechnet 1 + 2

? - $add(X, s(s(0)), s(s(s(0))))$ → $X = s(0)$ Berechnet 3-2.

? - $add(X, Y, s(s(s(0))))$. → 4 Lösungen

? - $add(X, s(s(0)), Z)$. \rightarrow Antwortsubstitution: $X = U, Z = s(s(U))$

? - $add(s(0), Y, Z)$. $\rightarrow \infty$ viele Antwortsubstitutionen.

\Rightarrow Fast jedes Prolog-Programm kann zu einem unendlichem SLD-Baum führen (hängt von der Anfrage ab).

- Eingebaute nat. Zahlen

Arithmetischer Ausdruck: Term aus Zahlen, Variablen, binären Infix-Funktionen ($+$, $-$, $*$, $//$ (*ganzzahl-Division*), $**$ (*Potenz*), \dots).

Präfix-Funktion: $-$

Die Funktionssymbole $+$, $-$, \dots kann man wie bisher mit syntaktischer Unifikation behandeln.

Programm:

`equal(X,X).`

? - $equal(3, 1 + 2)$. \Rightarrow No

? - $equal(X, 1 + 2)$. $\Rightarrow X = 1 + 2$

Auswertung von $+$, $-$, \dots nur durch vordef. Prädikate.

Vordefinierte Prädikatssymbole zum **Vergleich** arithmetischer Ausdrücke: $op \in \{<, >, =, <=, >=, =:=, = \setminus =\}$

? - $t_1 op t_2$. ist erfolgreich, wenn t_1, t_2 voll instantiierte arithmetische Ausdrücke sind und wenn das Resultat z_1 der Auswertung von t_1 und das Resultat z_2 der Auswertung von t_2 in der Relation op stehen.

Programm bricht mit Fehler ab, wenn t_1 oder t_2 kein voll instatiiertes arithmetischer Ausdruck ist.

$\Rightarrow op$ erzwingen Auswertung ihrer Argumente.

? - $1 < 2 \Rightarrow$ Yes.

? - $-1 < -1 \Rightarrow$ Yes

? - $1 * 1 < 1 + 1 \Rightarrow$ Yes

? - $2 < 1 \Rightarrow$ No

? - $6 // 3 < 5 - 4 \Rightarrow$ No

? - $a < 1 \Rightarrow$ Programmabbruch

? - $X < 1 \Rightarrow$ Programmabbruch

? - $X =:= 2 \Rightarrow$ Programmabbruch (führt nicht zur Antwortsubstitution $X = 2$)

\Rightarrow weiteres vordef. Prädikatssymbol "*is*"

? - $t_1 is t_2$.

ist erfolgreich, wenn t_2 voll inst. arithm. Ausdruck ist, der zu Wert z_2 -auswertet und wenn t_1 mit t_2 unifiziert. Programm bricht ab, wenn t_2 kein vollständig instantiiertes arithmetischer Ausdruck ist.

? - $2 is 1 + 1$. \Rightarrow Yes

? - $2 is 2$. \Rightarrow Yes

? - $1 + 1 is 2$. \Rightarrow No

? - $1 + 1 is 1 + 1$. \Rightarrow No

? - $X + 1 is 1 + 1$. \Rightarrow No

? - X is 2. \Rightarrow Antwortsubstitution $X = 2$
 ? - X is $1 + 1$. \Rightarrow Antwortsubstitution $X = 2$
 ? - X is $3 + 4$, Y is $X + 1$. \Rightarrow Antwortsubstitution: $X = 7, Y = 8$.
 ? - Y is $X + 1$, X is $3 + 4$. \Rightarrow Programmabbruch
 ? - X is X , ? - Z is X , X is a \Rightarrow Programmabbruch

Weitere vordefinierte Gleichheit "=" durch Faktum $X = X$. (keine Auswertung von $+$, $-$, \dots
 \rightarrow reine syntaktische Unifikation)

? - $a = a$, ? - $2 = 2$, ? - $1 + 1 = 1 + 1$ \Rightarrow Yes
 ? - $2 = 1 + 1$, ? - $1 + 1 = 2$ \Rightarrow No
 ? - $X + 1 = 1 + 1$, ? - $1 = X$ \Rightarrow Antwortsubstitution $X = 1$
 ? - $X = 1 + 1$ \Rightarrow Antwortsubstitution $X = 1 + 1$
 ? - $X = X$ \Rightarrow Antwortsubstitution $X = Y$ für neue Variable Y
 ? - $1 + X = Y + 1$ \Rightarrow Antwortsubstitution $X = 1, Y = 1$
 ? - $X = 3 + 4$, Y is $X + 1$ \Rightarrow $X = 3 + 4, Y = 8$
 ? - $X = f(X)$ \Rightarrow $X = f(f(\dots)) =$, unify_with_occurs_check: nur Termgleichheit
 ==: Wertgleichheit
 is Wertzuweisung (Auswertung nur auf der rechten Seite)

"add" mit vordefinierten Zahlen.

$add(X, 0, X)$.
 $add(X, Y, Z) : -Y > 0, Y1$ is $Y - 1, add(X, Y1, Z1), Z$ is $Z1 + 1$.
 (alternativ: $add(X, Y, Z) : -Z$ is $X + Y$).
 ? - $add(1, 2, X)$. \Rightarrow $X = 3$
 ? - $add(X, 2, 3)$. \Rightarrow Programmabbruch (in rekursivem Aufruf ist $Z1$ nicht voll instantiiert ist).
 \Rightarrow Bidirektionalität kann verloren gehen.

$add(X, 0, X)$.
 $add(X, Y + 1, Z + 1) : -add(X, Y, Z)$.
 ? - $add(1, 2, X)$. \Rightarrow No
 ? - $add(1, 0 + 1, X)$. \Rightarrow Antwortsubstitution: $X = 1 + 1$.
 \Rightarrow nicht vorteilhaft.

Beispiel 5.1.1 (Fakultät) $fakt(0, 1)$.

$fakt(X, Y) :- X > 0, X1$ is $X - 1, fakt(X1, Y1), Y$ is $Y1 * X$.

$ggT(0, X, X)$.
 $ggT(X, 0, X)$.
 $ggT(X, Y, Z) :- X < Y, X > 0, Y1$ is $Y - X, ggT(X, Y1, Z)$.
 $ggT(X, Y, Z) :- X > Y, X < 0, X1$ is $X - Y, ggT(X1, Y, Z)$.
 ? - $fakt(3, X)$ \Rightarrow $X = 6$
 ? - $ggT(28, 36, X)$ \Rightarrow $X = 4$

Weitere vordefinierte Prädikate, um "Typen" von Termen zu überprüfen, z.B. $number/1$
 $number(t)$ ist wahr, falls t eine **Zahl** ist.

? - $number(2).$ \Rightarrow Yes

? - $Xis1 + 1, number(X).$ \Rightarrow Yes

? - $number(1 + 1)$ \Rightarrow No

? - $number(X)$ \Rightarrow No

(Es ex. auch vordefinierte Gleitkommazahlen in Prolog).

5.2 Listen

Darstellung von Listen als Terme:

$nil \in \Sigma_0$ (leere Liste)

$cons \in \Sigma_2$ (vorne in Liste Einfügen) ($cons(1, cons(2, nil)) \hat{=} [1, 2]$)

$len(nil, 0).$

$len(cons(X, Xs, Y) : -len(Xs, Y1), YisY1 + 1.$

? - $len(cons(7, cons(3, nil)), X)$ $\Rightarrow X = 2$

Prolog bietet lesbarere Kurzschreibweise für Listen, falls man statt "nil" das Funktionssymbol $[]$ benutzt

statt "cons" das Funktionssymbol $.$ benutzt

$len([], 0).$

$len(., (X, Xs), Y) : -len(Xs.Y1), YisY1 + 1.$

Folgende Kurzschreibweisen sind möglich:

- $.(t_1, t_2) = [t_1|t_2]$
- $.(t, []) = [t]$
- $(t_1, .(t_2, .(t_3, t))) = [t_1, t_2, t_3|t]$
- $.(t_1, .(t_2, .(t_3, []))) = [t_1, t_2, t_3] = [t_1, t_2][t_3|[]] = [t_1|[t_2, t_3|[]]]$

Kurzschreibweisen werden als **identisch** zu dem Original-Term aufgefasst:

? - $[1, 2] = [1|[2]]$ \Rightarrow Yes

? - $(1, .(2, [])) = [1, 2]$ \Rightarrow Yes

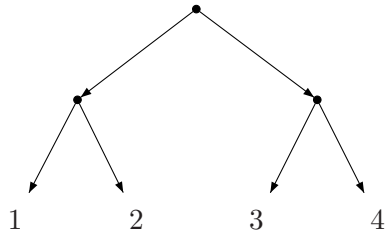
? - $(1, [2]) = [1, 2]$ \Rightarrow Yes

? - $(1, 2) = [1|2]$ \Rightarrow Yes

? - $(1, X) = [1, 2, 3]$ $\Rightarrow X = [2, 3]$

? - $[X, [1|X]] = [[2], Y]$ $\Rightarrow X = [2], Y = [1, 2]$

„.” is t ein rein syntaktisches Funktionssymbol. Kann man auch für Binärbäume benutzen:



$\cdot((1, 2), \cdot(3, 4)) = [[1|2]||[3|4]]$

20.06.05

Beispiel 5.2.1

```
member(X, [X|_]).
member(X, [_|Ys]) :- member(X, Ys).
```

? – $member(X, [[a, b], 1, []])$.

$X = [a, b]$;

$X = 1$;

$X = []$

No

? – $member(b, Xs)$. Welche listen enthalten b ?

$Xs = [b|Ys]$; Alle Listen mit 1. Element b

$Xs = [Y, b|Ys]$; Alle Listen mit 2. Element b ...

Es gibt also unendlich viele Antworten.

```
app([], Ys, Ys).
```

```
app([X|Xs], Ys, [X|Zs]) :- app(Xs, Ys, Zs).
```

? – $app([1, 2], [3, 4], Xs)$.

$Xs = [1, 2, 3, 4]$

? – $app(Xs, Ys, [1, 2, 3])$

$Xs = [], Ys = [1, 2, 3]$

$Xs = [1], Ys = [2, 3]$

$Xs = [1, 2], Ys = [3]$

$Xs = [1, 2, 3], Ys = []$

? – $app(Xs, [], Zs)$.

$Xs = [], Zs = []$;

$Xs = [X], Zs = [X]$;

⋮

5.3 Operatoren

Bisher: Schreibe Terme und atomare Formeln in Präfix-Notation mit Klammern: $p(X, f(a))$ mit $p \in \Delta_2, f \in \Sigma_1, a \in \Sigma_0$

p, f, a : **Funktoren**

Jetzt: Prädikat und Funktionssymbole in Infix-, Präfix-, Postfix-Schreibweise ohne Klammern (**Operatoren**).

Grund: bessere Lesbarkeit ("Programmieren in natürlicher Sprache").

Beispiel 5.3.1 $+, -, *, \text{etc.}$ sind bereits als Operatoren vordefiniert.

" $2 + 3$ " wird von Prolog in $+(2, 3)$ umgewandelt.

? $- 2 + 3 = +(2, 3)$.

Yes.

Benutzer kann selbst neue Operatoren deklarieren:

Direktive: Programmklauseln ohne Kopf (Anfragen). Beim laden versucht Prolog diese Anfragen zu beweisen. `:-op(Präzedenz,typ,Name(n) op` ist ein vordefiniertes Prädikat von Operatoren.

Neu deklarierte Operatoren sind **nach** der entsprechenden Direktive verwendbar..

Vordefinierte Direktiven:

`:-op(500,yfx,[+,-]).`

`:-op(400yfx,*)`

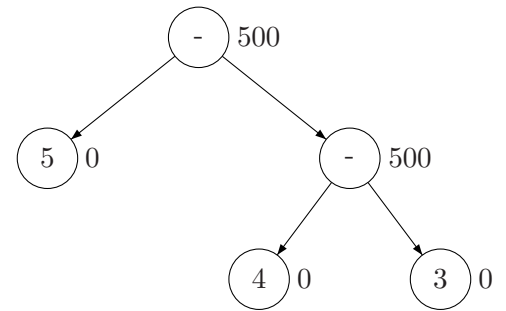
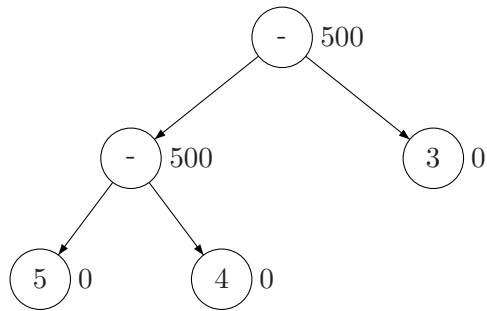
1. Argument Erstes Argument (Präzedenz): Zahl zwischen 0 und 1200 gibt an, wie stark der Operator bindet. **Kleinere** Präzedenzen $\hat{=}$ **stärkere** Bindung
2. Argument Zweites Argument (Typ): bestimmt die Reihenfolge von Operator und Argument(en).
 $f \hat{=}$ Operator, $x, y \hat{=}$ Argumente.
 xfx, yfx, xfy : Binäre Infix-Operatoren
 fx, fy : Präfix-Operatoren
 xf, yf : Postfix-Operatoren
3. Argument Drittes Argument: Name, Liste von Funktions- oder Prädikatssymbolen

$$\begin{aligned} 5 - 4 - 3 &= (5 - 4) - 3 = -2 \\ &= 5 - (4 - 3) = 4 \end{aligned}$$

x = Argumente mit Präzedenz, die echt kleiner als die Präzedenz von f ist

y = Argumente mit Präzedenz, die kleiner oder gleich der Präzedenz von f ist

Präzedenz eines Arguments = Präzedenz des führenden Operators des Arguments, sonst 0 (außenstehender Funktor, oder Klammern)



”yfx” bedeutet **Linksassoziativität**

? $- 1 + 2 + 3 = 1 + (2 + 3)$.

No

? $- 1 + 2 + 3 = (1 + 2) + 3$.

Yes

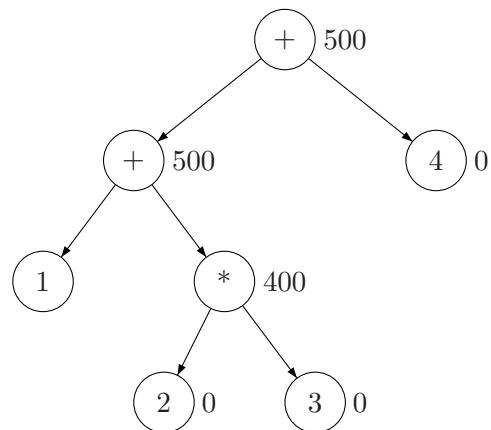
”xfy” bedeutet Rechtsassoziativität

”xfx” bedeutet keine Assoziativität: z.B.:

`:- op(500,xfx,+)`

$1+2+3 \Rightarrow$ Programmfehler

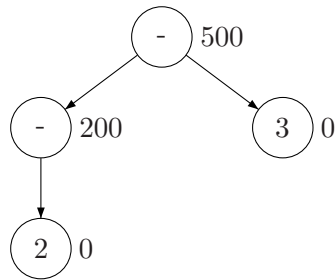
$1+2 * 3+4$: steht für $(1 + (2*3)) + 4$



Operatoren dürfen auch überladen werden:

`:- op(200 fy,-)`

$-2 - 3$ steht für $(-2) - 3$



Definiere eigene Operatoren für einfache Sprachverarbeitung:

- Verb "was", 2-stellig, Infix-Schreibweise. 'Laura was young' $\hat{=}$ was(laura,young).
Keine Assoziativität: (xfx) ("laura was young was beautiful" ist sinnlos).
- "of", 2-stellig, Infix. "secretary of john"
Rechtsassoziativ: (xfy) ("secrerary of (son of john)").
niedrigere Präzedenz als "was": "laura was (secretary of john)"
- "the", 1-stellig, Präfix Keine Assoziativität: (fx) ("the secretary the son" ist sinnlos).
Niedrigere Präzedenz als "of" ("(the secretary) of (the son)")

Programm:

```

:- op(300, fxf, was).
:- op(250, xfy, of).
:- op(200, fx, the).
  
```

laura was the secretary of the head of the department.

?- Who was the secretary of the head of the department
Who = laura

?- laura was What
What = the secretary of the head of the department

?-Who was the secretary of the head of What
Who= laura
What= department

5.4 Das Cut-Prädikat und Negation

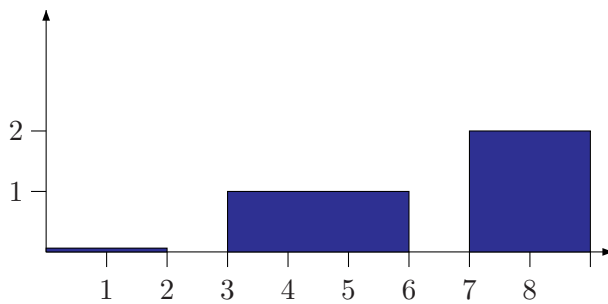
5.4.1 Das Cut-Prädikat

Prolog führt automatisch Backtracking durch wenn es einen endlichen Fehlschlag erkennt (Blatt + □). Dies kann nachteilig sein, da dies zeit- und speicherintensiv ist (Frühere Knoten mit Alternativen, müssen im Speicher gehalten werden) und auch zur Nichtterminierung führen kann.

⇒ Beschneide den SLD-Baum, so dass bestimmte Kanten beim zurücksetzen nicht mehr durchlaufen werden dürfen.

23.06.06

$$\text{Beispiel 5.4.1 } f(x) = \begin{cases} 0 & x < 3 \\ 1 & 3 \leq x < 6 \\ 2 & x \geq 6 \end{cases}$$



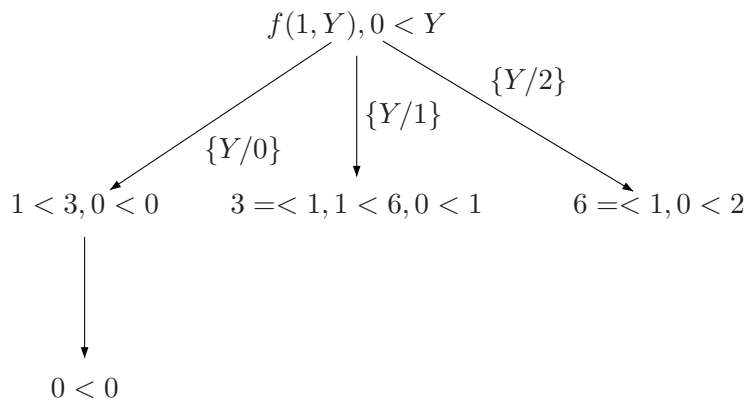
Prolog-Programm:

$f(X,0) :- X < 3.$

$f(X,1) :- 3 \leq X, X < 6.$

$f(X,2) :- 6 \leq X.$

? - $f(1,Y), 0 < Y.$



Ergebnis: No. Das Ergebnis kann erst ausgegeben werden, wenn der gesamte SLD-Baum aufgebaut wurde. Dies kann sehr ineffizient sein (bzw. nicht-terminierend).

Ziel: Verbessere Programm: Untersuche, welche Bedingungen um Programm sich ausschliessen → falls manche Teilziele bewiesen werden konnten, muss man manche anderen Teilziele gar nicht mehr untersuchen.

⇒ Falls $X < 3$ zutrifft, dann braucht man die zweite Klausel nicht mehr betrachten ($3 \leq X$),

und man braucht die dritte Klausel auch nicht mehr zu betrachten ($6 = < X$)

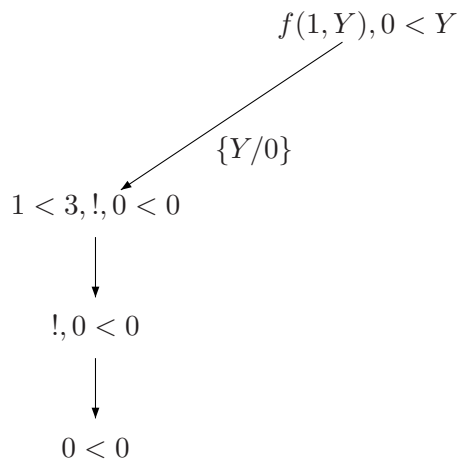
\Rightarrow Falls $X < 6$ zutrifft, dann braucht man die dritte Klausel nicht mehr zu betrachten ($6 = < X$).
Im Bsp: da "1 < 3" bewiesen wurde, sollte man den mittleren und den rechten Pfad des SLD-Baums gar nicht aufbauen.

In Prolog gibt es dazu das 0-stellige Prädikats-Symbol Cut: "!". Dieses darf in rechten Seiten von Regeln und in Anfragen auftreten. Es ist immer beweisbar, aber es schneidet Pfade im SLD-Baum ab.

$f(X, 0) : -X < 3, !$. (Falls Beweis von $X < 3$ gelingt, dann wird durch ! das Rücksetzen verhindert. \Rightarrow keine Betrachtung der anderen f -Klauseln.)

$f(X, 1) : -3 = < X, X < 6!$.

$f(X, 2) : -6 = < X$.

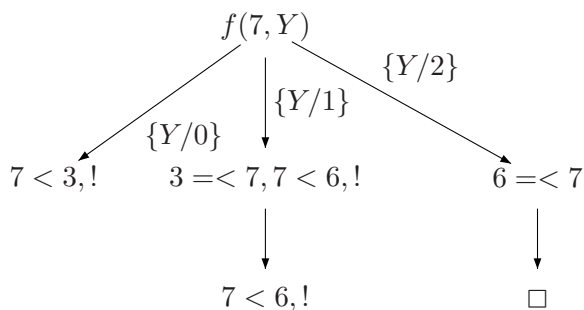


Nach dem Gelingen von $X < 3$ wird mit keiner anderen f -Klausel beim Rücksetzen resolviert.

"grüne Cuts": beeinflussen die Effizienz aber nicht das Ergebnis.

"rote Cuts": Weglassen der Cuts ändert nicht nur Effizienz sondern auch das Ergebnis.

Beispiel 5.4.2 ? - $f(/, Y)$.



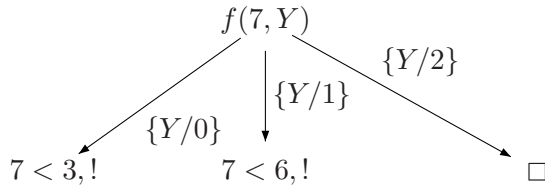
Antwort: $Y = 2$

Ineffizient: Falls $X < 3$ scheitert, muss $3 = < X$ gelingen
 Falls $X < 6$ scheitert, muss $6 = < X$ gelingen

$f(X, 0) : -X < 3, !$.

$f(X, 1) : -X < 6, !$.

$f(X, 2)$.



? - $f(1, Y)$. Antwort: $Y = 0; \underbrace{Y = 1; Y = 2}_{\text{ohne Cuts}}$

Bei Verwendung von Cuts denkt man an eine 'bestimmte Verwendung' des Prädikats (best. Positionen von Ein- und Ausgabe).

Im Beispiel: 1. Argument $\hat{=}$ Eingabe, 2. Argument $\hat{=}$ Ausgabe.

Bei anderer Verwendung können Cuts zu "unerwarteten" Ergebnissen führen.

? - $f(0, Y)$. $\rightarrow Y = 0$

? - $f(0, 2)$. $\rightarrow \text{Yes}$

Beispiel 5.4.3

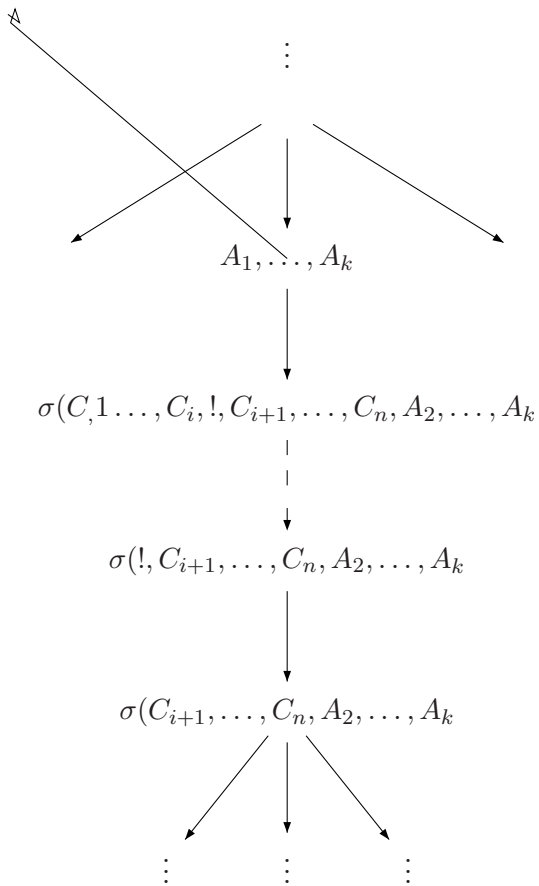
$p(0) : -!$.

$p(1) : -!$.

? - $p(X)$. Antwort: $X = 0$.

? - $p(1)$. Antwort: Yes

Genauere Bedeutung des Cuts: Falls Anfrage $?-A_1, \dots, A_k$ mit Programmklausel $B : -C_1, \dots, C_i, !, C_{i+1}, \dots$ resolviert wird und der Beweis von inst. Teilzielen C_1, \dots, C_i gelingt, dann entsteht folgender SLD-Baum:



Beispiel 5.4.4 $a(X) :- b(X)$.

$a(5)$.

$b(1) :- e(1)$.

$b(X) :- c(Y), d(X, Y)$.

$b(4)$.

$c(1) :- e(1)$.

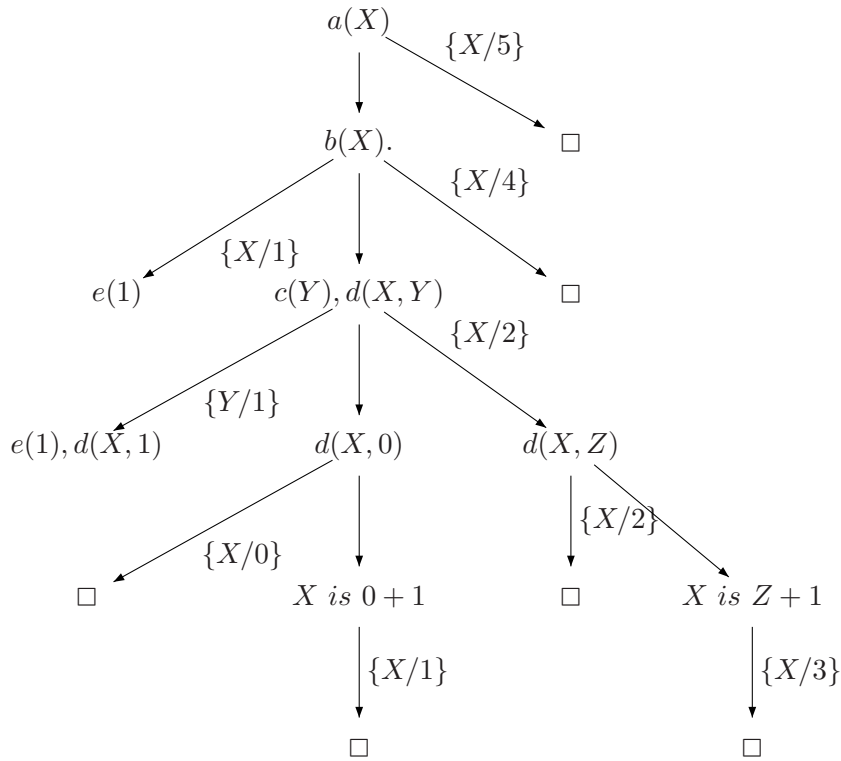
$c(0)$.

$c(2)$.

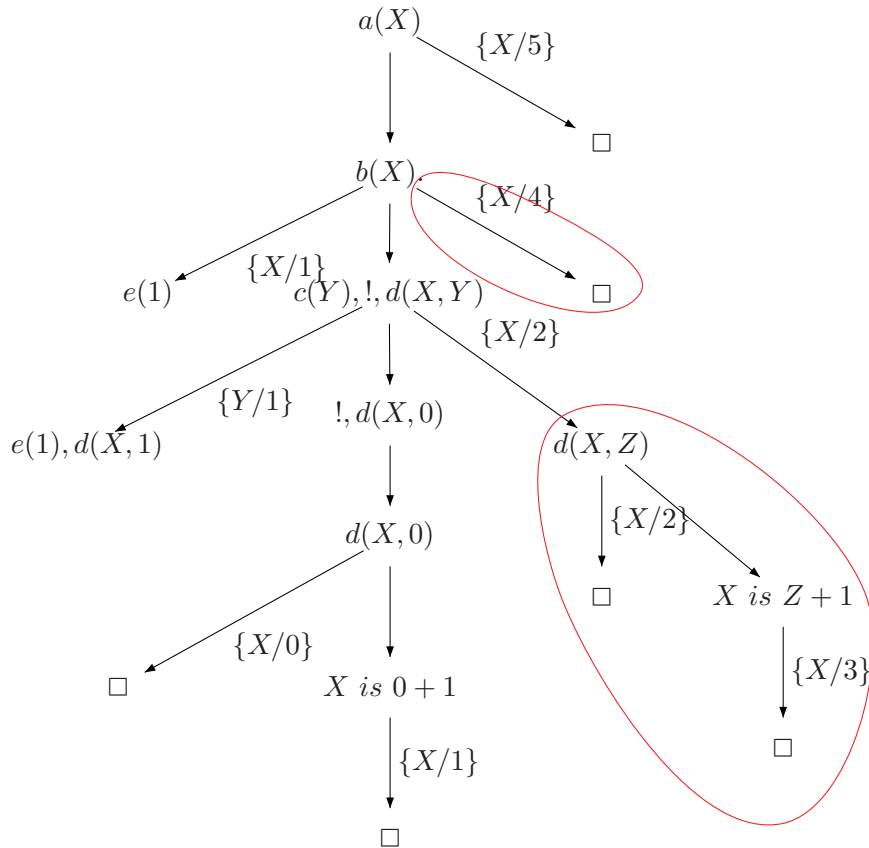
$d(X, X)$.

$d(X, Y) :- X \text{ is } Y+1$.

$e(0)$.



Antworten: X=0;X=1,X=2;X=3;X=4;X=5



Cut beeinflusst nur den

Beweis der Atome, die vor dem ! in der Klausel stehen (d.h. von $b(X)$ und $c(Y)$).

Die Beweise von $a(X)$, $d(X, Y)$ werden nicht beeinflusst.

Antworten: $X=0; X=1; X=5$

Beispiele zur Verwendung des Cuts

$ggT(X, 0, X)$.

$ggT(0, X, X)$.

$ggT(X, Y, Z) : -X < Y, X > 0, Y1 \text{ is } Y - X, ggT(X, Y1, Z)$.

$ggT(X, Y, Z) : -Y < X, Y > 0, X1 \text{ is } X - Y, ggT(X1, Y, Z)$.

Besser:

$ggT(X, 0, X) : -!$.

$ggT(0, X, X) : -!$.

$ggT(X, Y, Z) : -X < Y, !, Y1 \text{ is } Y - X, ggT(X, Y1, Z)$.

$ggT(X, Y, Z) : -X1 \text{ is } X - Y, ggT(X1, Y, Z)$

- Falls eine der ersten beiden Klauseln verwendet wird, sollte man die anderen ggT -Klauseln nicht mehr betrachten

- Falls in der 3. Klausel $X = < Y$ gelingt, dann betrachte die 4. Klausel nicht mehr.
- Da wir nur $X, Y \geq 0$ betrachten, erreicht man die 3. und 4. Klausel nur bei $X > 0, Y > 0$ (wegen ! in Klauseln 1 und 2) \Rightarrow kasse $X > 0, Y > 0$ weg.
- lasse $Y < X$ in Klausel 4 weg (wegen ! in Klausel 3)

`remove(X,Xs,Ys)` wahr falls Liste `Ys` aus `Xs` entsteht, indem man alle Vorkommen von `X` gelöscht werden. $? - \text{remove}(1, [0, 1, 2, 1], Ys) \rightarrow Ys = [0, 1]$

`remove(_, [], [])`.

`remove(X, [X|Xs], Ys) :- !, remove(X, Xs, Ys)`.

`remove(X, [Y|Xs], [Y|Ys]) :- remove(X, Xs, Ys)`.

Ohne Cut: $? - \text{remove}(1, [0, 1, 2, 1], Ys) \rightarrow Ys = [0, 2]; Ys = [0, 2, 1]; Ys = [0, 1, 2]; Ys = [0, 1, 2, 1]$

5.4.2 Meta-Variablen und Negation

Bislang: Terme $f(t_1, \dots, t_n), X$ mit f : Funktionssymbol, t_i : Terme, X : Variable

Atomare Formeln: $p(t_1, \dots, t_n)$ mit p : Prädikatssymbol.

Jetzt: keine Trennung mehr zwischen Funktions-/Prädikatssymbolen, d.h. keine Trennung zwischen Termen/atomaren Formeln.

Definition 5.4.5 (Meta-Variablen)

Variablen, die für **Formeln** statt für **Terme** stehen

Definition 5.4.6 (Meta-Prädikate:)

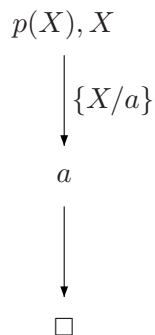
Prädikatssymbole, die **Formeln** (statt **Terme** als Argumente haben

Beispiel 5.4.7

$p(a)$. $\leftarrow p$ ist ein 1-stelliges Meta-Prädikatssymbol

a . $\leftarrow a$ ist ein 0-stelliges Prädikatssymbol.

Anfrage: $? - p(X), X$. $\leftarrow X$ ist Meta-Variable, kann mit Formel inst. werden.



Antwortsubst. $X = a$

Resolution und Unifikation wie bisher.

Aber: Meta-Variablen müssen instantiiert werden, bevor sie zur Resolution verwendet werden:

? - $p(X), X, Y$

Programm-Fehler: denn man müsste " $? - Y$ " beweisen (mit uninstantiierter Meta-Var. Y).

Meta-Präd. sind z.B. nützlich, um logische Junktoren zu programmieren.

$\text{or}(X, Y) :- X.$

$\text{or}(X, Y) :- Y.$

Ist Vordefiniert in Prolog:

; $(X, Y) :- X$

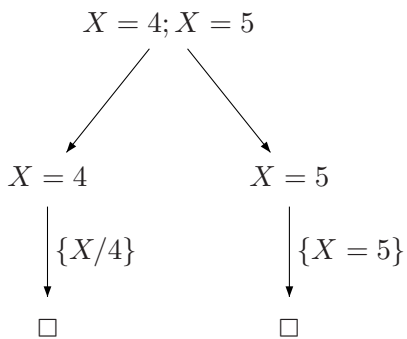
; $(X, Y) :- Y$

Rechtsassoziative Infix-Operatoren:

: $-op(1100, xfy, ;).$

: $-op(1000, xfy, ,).$

? - $X = 4; X = 5.$



Antwortsubst: $X = 4; X = 5$

$p(X, Y) :- X = 1, Y = 1; X = 2, Y = 2$

? - $p(X, Y).$

Antwortsubst: $(X = 1, Y = 1); (X = 2, Y = 2)$

Beispiel 5.4.8 $\text{if}(A, B, C) \hat{=} \text{if } A \text{ then } B \text{ else } C$

$\text{if}(A, B, C) :- A, !, B.$

$\text{if}(A, B, C) :- C.$

? - $\text{if}(A, B, C)$

beweise zunächst A : Wenn der Beweis von A gelingt, dann muss B bewiesen werden. Wenn B scheitert, dann scheitert auch $\text{if}(A, B, C)$, denn wegen des Cuts, kann die zweite Klausel nicht

verwendet werden.

Ohne Cut wäre $if(A, B, C)$ wahr, falls $(A$ und $B)$ wahr oder C wahr.

Wenn der Beweis von A nicht gelingt, dann muss C bewiesen werden.

Vordefiniert in Prolog: "if(A,B,C)" = "A \rightarrow B; C"

Beispiel 5.4.9 (Negation) Aus Programm jetzt nicht nur existenzquantifizierte Konjunktionen $A_1, \wedge \dots, A_k$ (A_i atomare Formel) herleiten, sondern auch Konjunktionen, die negierte atomare Formeln $\neg A$ enthalten.

$\mathcal{P} \models \neg A$ gilt **nie!** (mit \mathcal{P} : Menge von definiten Hornklauseln)

$$\left\{ \begin{array}{l} \text{mutterVon(renate, susanne),} \\ \text{mutterVon(susanne, aline)} \end{array} \right\} \models \neg \text{mutterVon(renate, klaus)}?$$

Die Struktur, die alle atomaren Formeln wahr macht, ist Modell von \mathcal{P} aber nicht von $\neg A$

\Rightarrow Prädikat "not" kann nicht die Semantik der normalen Negation haben,

Bei der Realisierung der Negation werden zwei Annahmen getroffen:

a) Aus dem Programm sind alle wahren Aussagen über die Welt herleitbar (Closed World Assumption)

Falls A nicht herleitbar ist, dann ist A auch nicht wahr $\rightarrow \neg A$ wahr.

b) Falls eine Aussage aus dem Programm nicht herleitbar ist, dann wird das in endlicher Zeit festgestellt.

\Rightarrow Interpretiere Negation als "endlichen Fehlschlag" (Negation as Failure)

Um $\neg A$ zu beweisen, versuche A zu beweisen. Falls dies in endlicher Zeit fehlschlägt, dann ist der Beweis von $\neg A$ erfolgreich. (Entspricht der wahren Negation, falls a) und b) zutreffen).

`not(A) :- !, fail.`

`not(A).`

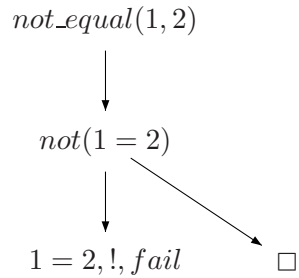
`fail/0` ist vordefiniert und schlägt immer fehl.

Der Cut ist nötig damit man nicht zurücksetzt, wenn A beweisbar ist.

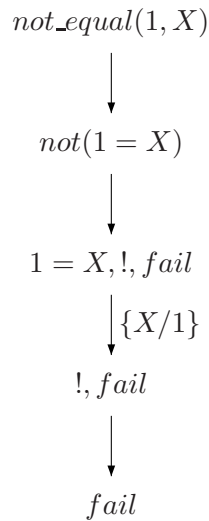
`not/1` ist vordefiniert, auch als Präfix-Schreibweise: `\ + A $\hat{=}$ not(A)`

Beispiel 5.4.10 `not_equal(X, Y) :- not(X=Y).`

? - `not_equal(1, 2).` \Rightarrow Yes



- ? - $\text{not_equal}(1,1)$. \Rightarrow No
 ? - $X=2, \text{not_equal}(1,X)$. \Rightarrow Yes
 ? - $\text{not_equal}(1,X)$. \Rightarrow No



Anfragen sind **allquantifiziert** (gilt $1 \neq X$ für alle X ?)

Problem wenn b) nicht zutrifft:

$\text{even}(0)$.
 $\text{even}(X) :- X1 \text{ is } X-2, \text{even}(X1)$.

? - $\text{not}(\text{even}(1))$. terminiert nicht. $\text{even}(1)$ folgt **nicht** aus dem Programm, aber Fehlschlag wird nicht in endlicher Zeit festgestellt.

Problem, wenn a) nicht zutrifft:

$\text{even}(0)$.
 $\text{even}(X) :- X >= 2, X1 \text{ is } X-2, \text{even}(X1)$.

? – $\text{not}(\text{even}(1)) \Rightarrow \text{Yes}$.

? – $\text{not}(\text{even}(-2)) \Rightarrow \text{Yes}$. Das Programm enthält nicht alles Wissen über die Welt. Closed world assumption trifft hier nicht zu.

Korrekte Version:

```
even(0):-!.
even(X):-X>0,!,X1 is X-1,not(even(X1)).
even(X):-X1 is X+1,not(even(X1)).
```

5.5 Ein- und Ausgabe

Bisher:

Eingaben: Anfragen an das Programm

Ausgaben: Antwortsubstitutionen, Yes/No

Jetzt: vordefinierte Prädikate, die Seiteneffekte mit Ein-/Ausgabe durchführen.

5.5.1 Ausgabe

`write/1`: schreibt den Argument-Term in den aktuellen Ausgabe-Stream (standardmäßig ist das der Bildschirm des Benutzers).

? – $\text{write}(t)$. gelingt immer, als Seiteneffekt wird "t" ausgegeben.

? – $X \text{ is } 2 + 3, \text{write}(X)$. Antwortsubstitution: $X = 5$, als Seiteneffekt wird zusätzlich 5 auf dem Bildschirm ausgegeben.

? – $\text{write}(\underbrace{\text{'Dies ist eine Konstante'}}_{0\text{-stelliges Fktssymbol}})$.

Seiteneffekt: Gibt "Dies ist eine Konstante" aus
Yes

```
mult(X,Y) :- Ergebnis is X*Y, write(X*Y), write(' = '), write(Ergebnis)
```

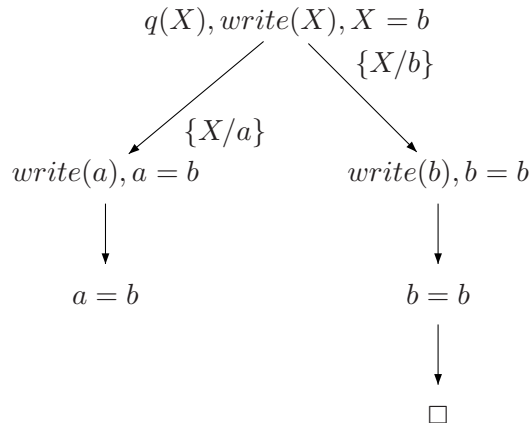
? – $\text{mult}(3,4) \Rightarrow 3*4 = 12$

Achtung: beim Rücksetzen werden Seiteneffekte nicht rückgängig gemacht,

q(a).

q(b).

p:-q(X),write(X),X=b.



1.7.06

Weitere vordefinierte Prädikate: für die Ausgabe, z.B. *nl/0* (new line).

? – *write(a), nl, write(b), nl, write(c)*

Ausgabe:

a
b
c

5.5.2 Eingabe

read/1 read(t): Liest einen Term *s* vom aktuellen Eingabestream. und versucht, *t* und *s* zu unifizieren. Dies schlägt fehl, falls *s* und *t* nicht unifizierbar sind. Um das Ende des eingelesenen Terms *s* zu markieren, muss Eingabe mit "." enden.

? – *read(X)*.

Benutzereingabe: "5."

Antwortsubstitution: X=5

? – *read(f(X, Y))*.

Benutzereingabe: "f(2,3)"

Antwortsubstitution: X=2, Y=3

Benutzereingabe: 5

Antwortsubstitution: No.

*sqr(X, Y) :- Y is X*X*

sqr :- nl,

write('Bitte geben Sie eine Zahl oder "stop" ein:'),

```
read(X),
proc(X).
```

```
proc(stop) :-!.
proc(X) :-  sqr(X,Y),
write('Das Quadrat von '), write(X), write(' ist '),write(Y), sqr.
```

Start des Beispielprogramms:

```
?-sqr.
```

```
Bitte geben Sie eine Zahl oder "stop" ein: 3.
Das Quadrat von 3 ist 9
Bitte geben Sie eine Zahl oder "stop" ein: -4.
Das Quadrat von -4 ist 16
Bitte geben Sie eine Zahl oder "stop" ein: stop.
```

Yes

5.5.3 Ein-/Ausgabe mit Files:

Änderung des aktuellen Ein-/Ausgabestreams mit *see/1,tell/1*.

see(t) setzt den Eingabestream auf t. Der standard-Ausgabestream ist "user" (*tell(user)*.)

see(t) setzt den Eingabestream auf t. Der standard-Eingabestream ist "user" (*see(user)*.)

seen/0,told/0 schließen den aktuellen Ein-/Ausgabestream und setzen ihn zurück auf "user".
end_of_file/0 wird von Prolog gelesen, wenn das Ende der Datei erreicht wurde.

Modifiziertes Programm:

```
sqr(X,Y) :- Y is X*X
```

```
start :- nl,
    write('bitte geben Sie den Namen eines Eingabefiles ein: '),
    read(Eingabefile),
    write(Bitte geben Sie den Namen eines Ausgabefiles ein: '),
    read(Ausgabefile),
    see(Eingabefile),
    tell(Ausgabefile),
    sqr,
    seen,
    told.
```

```
sqr :- read(X),
    proc(X).
```

```
proc(end_of_file) :-!.
```

```

proc(X) :-  sqr(X,Y),
write('Das Quadrat von '), write(X), write(' ist '),write(Y),
nl,
sqr.

```

Beispielfrage:

?-start.

Bitte geben Sie den Namen eines Eingabefiles ein: ein.

Bitte geben Sie den Namen eines Ausgabefiles ein: aus.

ein: 3. -4.

aus: Das Quadrat von 3 ist 9

Das Quadrat von -4 ist 16

5.6 Meta-Programmierung

Manipulation von Termen und Programmen.

5.6.1 Verarbeitung von Termen und atomaren Formeln

Prädikate um bestimmte Arten von Termen zu erkennen: Erkennung bestimmter Arten von Termen:

- number/1 (Abschnitt 5.1): *number(t)* wahr falls *t* eine Zahl ist.
 ? - *number(2)*. / *number(-2)*. \Rightarrow Yes
 ? - *number(a)*. / *number(X)*. / *number(2 + 5)*. \Rightarrow No
- var/1 : *var(t)* ist wahr, falls *t* eine Variable ist.
 ? - *var(X)*. Antwort: $X = X1$
 ? - $X = 2$, *var(X)*. Antwort: No.
 ? - *var(X)*, $X = 2$. Antwort: $X = 2$
- nonvar/1: *nonvar(t)* ist wahr, falls *t* keine Variable ist.
 ? - *nonvar(a)*. / $X = 2$, *nonvar(X)*. \Rightarrow Yes
 ? - *nonvar(X)* \Rightarrow No.
- atomic/1: *atomic(t)* ist wahr, falls *t* 0-stelliges Funktions- oder Prädikatssymbol oder eine Zahl ist. ? - *atomic(a)*. / ? - *atomic(-)*. / ? - *atomic(-2)* \Rightarrow Yes
 ?*atomic(X)*. / ? - *atomic(a(a))* \Rightarrow No
- atom/1: Wie atomic, jedoch nicht wahr wenn *t* eine Zahl ist
- compound: *compound(t)* ist wahr, falls *t* ein Term/atomare Formel ist, die nicht nur aus 0-stelligen Funktions-/Prädikatssymbolen oder eine Zahl besteht.
 ? - *compound(a)*. / ? - *compound(X)*. / *compund(-2)* \Rightarrow No
 ? - *compound(a(a))*. / ? - *compound(1 + 2)* \Rightarrow Yes

Prädikate, um Bestandteile von Termen zu extrahieren und um neue Terme zu konstruieren:

Idee: Man könnte Terme als **Listen** schreiben.

Term: $f(a, b) \rightarrow$ Liste: $[f, a, b]$

Vordefiniertes Prädikat: ' $= ..$ ': $t = ..l$ ist wahr, falls l die Listendarstellung des Terms t ist.

Die Umwandlung von Termen/Listen findet nur auf der **obersten** Ebene statt.

anfrage	Antwort
$? - f(a, b) = ..L$	$L = [f, a, b]$
$? - 1 + 2 = ..L$	$L = [+ , 1, 2]$
$? - T = ..[f, a, b].$	$T = f(a, b)$
$f(a(b)) = ..L$	$L = [f, a(b)]$

$? - X = ..Y / ? - X = ..[Y, a, b] / ? - X = ..[f|L]$: Programmfehler: das äußerste Funktionssymbol und seine Stelligkeit sind nicht eindeutig.

Beispiel 5.6.1 (Einsatz von = ..) Betrachte ein Programm zur Vergrößerung geometrischer Figuren. Verschiedene Prädikatsymbole für verschiedene Figurenarten.

```
square(Side).
rectangle(Side1,Side2).
triangle(Side1,Side2,Side3).
circle(Radius).
```

Ziel: $enlarge/3$. $enlarge(Fig, Factor, NewFig)$ ist wahr, falls $NewFig$ aus Fig durch Vergrößerung um $Factor$ entsteht.

Naive Lösung:

```
enlarge(square(Side),Factor,square(NewSide)):-
    NewSide is Factor * Side.
enlarge(rectangle(Side1,Side2),Factor,rectangle(NewSide1,NewSide2)) :-
    NewSide1 is Factor * Side1,
    NewSide2 is Factor * Side2.
enlarge(triangle(Side1,Side2,Side3),Factor,triangle(NewSide1,NewSide2,NewSide3)) :-
    NewSide1 is Factor * Side1,
    NewSide2 is Factor * Side2,
    NewSide3 is Factor * Side3.
enlarge(circle(Radius),Factor,circle(NewRadius)):-
    NewRadius is Factor * Radius.
```

Eine $enlarge$ -Klausel für jeden Figurentyp. nachteil:

- großer Programmieraufwand
- alle Figurentypen müssen bereits bekannt sein.

Bessere Lösung:

```
enlarge(Fig,Factor,NewFig) :-
    Fig =..[Type | Param],
    multiplyList(Param,Factor,NewParam),
    NewFig =..[Type|NewParam].
```

```
multiplyList([],_[]).
```

```
multiplyList([X|Xs],Factor,[Y|Ys]) :- Y is X*Factor, multiplyList(Xs,Factor,Ys).
```

Type $\hat{=}$ äußerestes Funktionssymbol von Fig (square,rectangle,..).

Idee:

- überführe Term in Liste
- manipulierte Liste
- überführe Liste in Term

Weitere vordefinierte Prädikate zum Zugriff und zur Manipulation von Termen: functor/3,arg/3.

- *functor*(*t*, *f*, *n*) ist wahr, falls *f* das äußerste Funktionssymbol von *t* ist und *n* die Stelligkeit von *f* ist.
 $? - \text{functor}(g(f(X), X, g), F, N). \rightarrow X = X1, F = g, N = 3$
 $? - \text{functor}(T, g, 3). \rightarrow g(X, Y, Z).$
- *arg*(*n*, *t*, *a*) ist wahr, falls *a* das *n*-te Argument im Term *t* ist. (Nummerierung beginnt mit 1).
 $? - \text{arg}(3, g(f(X), X, g), A) \rightarrow A = g, X = X1.$

```
?-functor(D,date,3),
    arg(1,D,4),
    arg(2,D,F),
    arg(3,D,2006).
```

```
D = date(4,7,2006)
```

Beispiel 5.6.2 (ground/1) *ground*(*t*) ist wahr, falls *t* Grundterm (ohne Variable) ist.

```
ground(T) :- nonvar(T),
    T=..[Functor|Args],
    groundList(Args).
```

```
groundList([]).
```

```
groundList([T|Ts]) :- ground(T), groundList(Ts).
```

```
a = ..L → L = [a]
```

5.6.2 Verarbeitung von Programmen

Ein Prolog Programm ist eine Datenbank, die aus Klauseln (Fakten und Regeln) besteht.

Lesen der Datenbank während der Lauzeit über das vordefinierte Prädikat *clause/2* möglich.
 $? - clause(\underbrace{h, b}_{\text{Terme/atom. Formeln}})$ ist wahr, falls es eine Programmklausel " $B : -C_1, \dots, C_k$ " gibt, so

dass " $clause(h, b)$ " und " $clause(B, (C_1, \dots, C_k))$ " unifzieren.

Die Antwortsustitution ist der *mgu*, eingeschränkt auf die Variablen aus h, b .

Beispiel 5.6.3

```
times(_0,0).
times(X,Y,Z):-Y>0,Y1 is Y-1, times(X,Y1,Z1), Z is Z1 + X.
```

Aufruf: $? - clause(times(X, Y, Z), Body)$.

Antwort: $X = XX, Y = 0, Z = 0, Body = true$ ($true/0$ ist vordefiniert und immer wahr).

$X = X, Y = YY, Z = ZZ, Body = (YY > 0, YY1 is YY - 1,)$

clause ermöglicht das Auslesen des gerade laufenden Programms (sowohl für benutzerdefinierte wie auch für vordefinierte Prädikatssymbole).

Schreiben der Datenbank während der Laufzeit.

Es gibt dazu zwei vordefinierte Prädikate: *assert/1*, *retract/1*

Beweis von "*assert(t)*" gelingt stets, und als Seiteneffekt wird die Klausel *t* ans Ende des Programms hinzugefügt.

Beispiel 5.6.4 Betrachte ein Prolog-Programm mit 2 times-Klauseln.

$? - assert(p(0))$. gelingt, als Seiteneffekt, wird das Faktum " $p(0)$." an das Ende des Programms hinzugefügt.

$? - p(X)$. liefert $X = 0$.

Auslesen des momentanen Programms mit *clause*:

$? - clause(p(X), B)$. liefert die Antwortsustitution: $X = 0, B = true$.

$? - -assert(square(X, Y)) : -times(X, X, Y)$.

$? - square(3, Y)$. Antwort: $Y = 9$

Zum Einfügen am Anfang des Programms: *asserta/1* (am Ende *assertz/1* $\hat{=}$ *assert/1*). Das Verändern von Programmklauseeln ist nur möglich für Prädikatssymbole, die als "dynamisch" vereinbart wurden .

- Alle durch *assert(a/z)* eingeführten neuen Prädikatssymbole sind dynamisch.
- Alle Prädikatssymbole, die im Programm als "dynamisch" deklariert wurde: *dynamic/1* bekommt als Argument: Name des Prädikatssymbols / Stelligkeit des Prädikatssymbols. Die Stelligkeit ist zur eindeutigen Identifizierung des Symbols nötig. *dynamic* ist Präfix-Operator \Rightarrow keine Klammern um Argument nötig.

Beispiel 5.6.5

```
:-dynamic times/3
times(_,0,0).
times(X,Y,Z) :- ...
```

Direktive \Rightarrow wird beim Laden bewiesen. Als Seiteneffekt wird *times/3* dynamisch.

? - *asserta(times(X,1,X))*. \Rightarrow fügt das Faktum "*times(X,1,X)*." vorne in das Programm ein.

? - *clause(times(X,Y,Z),B)* ergibt $X = XX, Y = Y1, Z = ZZ, B = true$.

Löschen von Programmklauseln

? - *retract(t)*. gelingt, falls es eine Programmklauseln gibt, die mit der Klausel *t* unifiziert. Als Seiteneffekt wird die erste solche Regel gelöscht. Nach eingabe von ";" wird die nächste solche Klausel gelöscht.

? - *retract(times(X,Y,Z) :- Body)*. Gelingt und *times(X,1,X)* wird gelöscht.

Eine Anfrage zum Löschen aller *times*-Klauseln: ? - *retract(times(X,Y,Z) :- Body)*, *fail*.

Das Löschen aller "*times*"-Fakten geht über die Anfrage ? - *retract(times(X,Y,Z))*, *fail*.

Vorsicht bei der Verwendung von *assert* und *retract*. Dies kann zu völlig unverständlichen Programmen führen.

Beispiel 5.6.6 Hier benutzen wir *assert* zum Abspeichern von häufig benötigten Berechnungsergebnissen zu Effizienzsteigerung.

Erzeuge eine Tabelle mit allen Multiplikationen " $X \cdot Y$ " für $X, Y \in \{0, \dots, 9\} \Rightarrow$ Programm aus 100 Fakten.

Schreibe das Programm nicht selbst, sondern erzeuge es automatisch:

```
maketable :- L=[0,1,2,3,4,5,6,7,8,9],
    member(X,L),
    member(Y,L),
    Z is X * Y,
    assert(times(X,Y,Z)),
    fail.
```

? - *maketable*. \rightarrow No.

? - *times(X,Y,8)*. $\rightarrow X = 1, Y = 8 ; X = 2, Y = 4 ; X = 4, Y = 2 ; X = 8, Y = 1$

Beispiel 5.6.7

```
weiblich(monika).
weiblich(karin).
weiblich(renate).
weiblich(susanne).
weiblich(aline).
```

```

maennlich(werner).
maennlich(klaus),
maennlich(gerd).
maennlich(peter).
maennlich(dominique).

verheiratet(werner, monika).
verheiratet(gerd, reate).
verheiratet(klaus, susanne).

mutterVon(monika, karin).
mutterVon(monika, klaus).
mutterVon(reate, susanne).
mutterVon(reate, peter).
mutterVon(susanne, aline).
mutterVon(susanne, dominique).

```

Verwandschafts-Programm +

```
vaterVon(V,K) :- verheiratet(V,F), mutterVon(F,K).
```

? – $vaterVon(gerd, K)$. $\rightarrow K = susanne ; K = peter$

Ziel: Berechne Liste **aller** Kinder von gerd. (d.h. Aufsammeln **aller** Lösungen)

Ziel *findall*/3

findall(t, g, l) ist wahr falls:

- Die Anfrage g hat die Antwortsubstitution $\sigma_1, \dots, \sigma_k$. (Tiefensuche von links nach rechts).
- l ist die Liste $[\sigma_1(t), \dots, \sigma_k(t)]$.

? – $findall(K, vaterVon(gerd, K), L)$. $\rightarrow L = [susanne, peter]$, $K = KK$

? – $findall(vaterVon(gerd, K), vaterVon(gerd, K), L)$. \rightarrow

$L = [vaterVon(gerd, susanne), vaterVon(gerd, peter)]$, $K = KK$

Programmieren *findall* selbst

```

findall(X,Anfrage,Liste):- Anfrage,
    assert(loesung(X)),
    fail;
    sammleLoesungen(Liste).

```

```

sammleLoesungen([X|Rest]):-retract(loesung(X)),
    !,
    sammleLoesungen(Rest).
sammleLoesungen([]).

```

Baue mit den ersten drei Zeilen den kompletten SLD-Baum für *Anfrage* aus. Füge *loesung*($\sigma_1(X)$), ..., *loesung* dem Programm hinzu. Der Cut in *sammleLoesungen* sorgt dafür, dass falls nach findall-Anfrage wieder zurückgesetzt wird, keine fehlerhaften Lösungen ausgegeben werden.

11.7.06

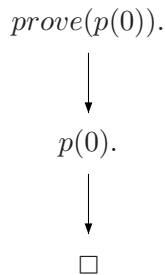
5.6.3 Meta-Interpreter

Ein Meta-Interpreter ist ein Interpreter der in der Sprache geschrieben ist, die er interpretiert, z.B. ein Prolog-Interpreter der in Prolog geschrieben ist. Diese sind z.B. für Rapid Prototyping geeignet.

Einfachste Meta-Interpreter:

```
prove(Goal):-Goal.
```

Fals das Programm das Faktum "p(0)." enthält, dann führt die Anfrage "prove(p(0))" zu folgendem SLD-Baum:



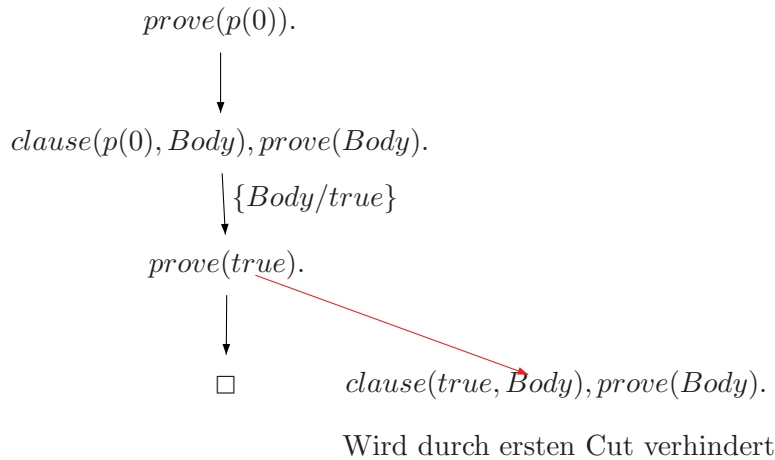
Dieser Meta-Interpreter ist ziemlich nutzlos, da er die Anfrage an den ursprünglichen Prolog-Interpreter delegiert.

Meta-Interpreter 1

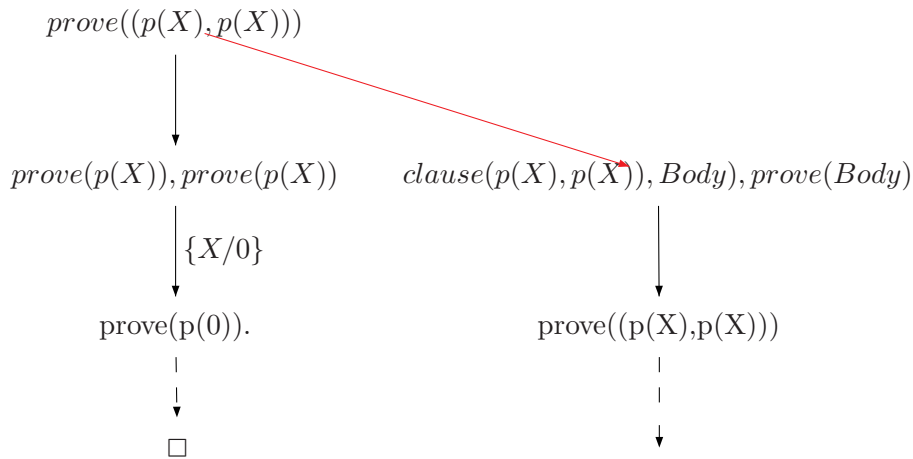
```

prove(true):-!.
prove((Goal1,Goal2)) :- !, prove(Goal1), prove(Goal2).
prove(Goal) :- clause(Goal,Body), prove(Body).

```



Dieser Interpreter funktioniert allerdings nur für reine Logikprogramme. Die Cuts sind notwendig um den SLD-Baum endlich zu halten:



Der Meta-Interpreter 1 kann als Ausgangspunkt für alternative Interpreter verwendet werden.

Meta-Interpreter 2:

```

prove(true) :- !.
prove((Goal1, Goal2)) :- !, prove(Goal2), prove(Goal1).
prove(Goal) :- clause(Goal, Body), prove(Body).

```

Meta-Interpreter 3: wieder von links nach rechts, aber zusätzlich soll die Länge des Beweises mit ausgegeben werden. ? – $prove(p(0), N)$. Antwort: $N = 3$

Meta-Interpreter 1

```

prove(true, 1) :- !.

```

prove((Goal1,Goal2),N) :- !, prove(Goal1,N1), prove(Goal2,N2), N is N1+N2.
 prove(Goal,N) :- clause(Goal,Body), prove(Body,N1), N is N1+1.

5.7 Differenzlistenes und definite Klauselgrammatiken

5.7.1 Differenzlisten

Alternative Listendarstellung \Rightarrow viele Listenoperationen dadurch wesentlich effizienter Programmierbar. **Klassisches Append**

append([],Ys,Ys).
 append([X|Xs],Ys,[X,Zs]) :- append(Xs,Ys,Zs).

Aufwand zum Konkatenieren von 2 Listen: $O(n)$, n Länge der ersten Liste. ?-append([1,2,3],[4,5],Zs).
 $\Rightarrow Zs = [1,2,3,4,5]$

Alternative Darstellung: Repräsentiere eine Liste als die **Differenz** von zwei anderen Listen:
 $[1,2,3,4,5] - [4,5] = [1,2,3]$
 $[1,2,3] - [] = [1,2,3]$
 $[1,2,3,4,5|Ys] - [4,5|Ys] = [1,2,3]$
 $[1,2,3|Ys] - Ys = [1,2,3] \leftarrow$ allgemeinste Darstellung als Differenzliste.

Alternative Implementierung:

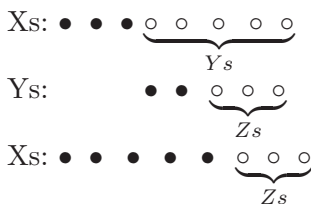
app(Xs-Ys,Ys,Xs).

?-app(t_1, t_2, Xs). Falls t_1 eine Liste in Differenzlisten-Darstellung ist (z.B. $t_1 = [1,2,3|Ys] - Ys$), dann wird in einem Resolutions-Schritt Ys mit t_2 instantiiert und die Antwortsubstition ist $Xs = [1,2,3|t_2]$.

$$\begin{array}{c} \text{app}([1,2,3|Ys] - Ys, [4,5], Xs). \\ \downarrow \\ \{Ys/[4,5], Xs/[1,2,3,4,5]\} \\ \downarrow \\ \square \end{array}$$

Nachteil: das erste Argument ist in Differenzlisten-Darstellung, aber das zweite und dritte Argument nicht, dies führt zu Problemen bei der Mehrfachanwendung von append.

Alternative: app(Xs - Ys,)



$$\begin{array}{c}
 \text{app}([1, 2, 3|Ys] - Ys, [4, 5|Zs] - Zs, \text{Erg}) \\
 \downarrow \\
 \{Ys/[4, 5|Zs], \text{Erg}/[1, 2, 3, 4, 5|Zs] - Zs\} \\
 \downarrow \\
 \square
 \end{array}$$

Antwort: $\text{Erg} = [1, 2, 3, 4, 5|Zs] - Zs$.

app ist nur dann verwendbar, wenn die Differenzlisten-Darstellung der ersten beiden Argumente kompatibel ist. $? - \text{app}([1, 2, 3, 6] - [6], [4, 5|Zs] - Zs, \text{Erg}) \Rightarrow \text{No}$.

$? - L = [1|Ys] - Ys, \text{app}(L, [2|Zs] - Zs, \text{Erg1}), \text{app}(L, [3, |Ws] - Ws, \text{Erg2}). \Rightarrow \text{No}$.

5.7.2 Definite Klauselgrammatiken

Ziel ist die Repräsentation von kontextfreien Grammatiken in Prolog. \rightarrow automatisch wird ein Prolog-Programm generiert, um herauszufinden, ob ein Wort in der Sprache liegt oder nicht (Wortproblem).

Grammatik: $G = (N, T, S, P)$ mit

N Nichtterminalsymbole

T Terminalsymbole

S Startsymbol

P Regeln der Art $A \rightarrow \alpha$ mit $A \in N, \alpha \in (N \cup T)$

$\beta \Rightarrow_G \gamma$, falls $\beta = \beta_1 A \beta_2, \gamma = \beta_1 \alpha \beta_2, A \rightarrow \alpha \in P$

$L(G) = \{w \in T^* | S \Rightarrow_G^* w\}$.

Beispielgrammatik $G = (N, T, S, P)$.

- $N = \{ \text{Satz, Nominalphrase, Verbalphrase, Artikel, Nomen, Verb} \}$
- $T = \{a, the, cat, mouse, scares, hates\}$
- $S = \text{Satz}$
- P besteht aus folgenden Regeln

Satz	→	Nominalphrase Verbalphrase
Nominalphrase	→	Artikel Nomen
Verbalphrase	→	Verb
Verbalphrase	→	Verb Nominalphrase
Artikel	→	a
Artikel	→	the
Nomen	→	cat
Nomen	→	mouse
Verb	→	scares
Verb	→	hates

Darstellung von kontextfreien Grammatiken in Prolog:

- Nicht-Terminale werden als Konstanten dargestellt (0-stellige Prädikatssymbole), z.B. `satz`, `nominalphrase` etc.
- Terminalsymbole: ein-elementige Liste mit Konstante, z.b. `[a]`, `[the]`, `[cat]`, ...
- worte aus T^* : Listen von Konstanten, z.B. `[a, cat, scares, the, mouse]`, `[]`
- Worte aus $(N \cup T)^*$: durch Kommas getrennte Sequenzen aus Konstanten und Listen von Konstanten: `[a, mouse]`, `verbalphrase`
- `to` wird zu `-- >`

```
satz --> nominalphrase, verbalphrase.
nominalphrase --> artikel, nomen.
verbalphrase --> verb.
verbalphrase --> verb, nominalphrase.
artikel --> [a].
artikel --> [the].
nomen --> [cat].
nomen --> [mouse].
verb --> [scares].
verb --> [hates].
```

Prolog übersetzt solche Grammatiken in entsprechende Klauseln:

1. Idee: Generiere zu jedem Nicht-Terminalsymbol ein 1-stelliges Prädikatssymbol.

`satz([a, cat, scares, the, mouse]).` wahr gdw. $Satz \Rightarrow_G^* a \text{ cat scares the mouse}$

Nachteil: ineffizient, da dabei klassisches `append` benutzt werden würde.

Besser ist die Differenzlistendarstellung.

2. Idee Ordne jedem Nicht-Terminalsymbol a ein Prädikatssymbol a zu wobei $a(A - B)$ wahr ist, falls man aus a das Wort A ohne seinen Rest B herleiten kann, d.h. falls man den Anfang von A durch Nicht-Terminalsymbol a herleiten kann und dann noch den Rest B des Wortes akzeptieren muss.

Darstellung: $a(A, B)$

Überführe Regel $a \text{ -- } > a_1$ in Klausel $a(A, B) \rightarrow a_1(A, B)$.

$a \text{ -- } > a_1, a_2$ in Klausel $a(A, B) \rightarrow app(Xs - Ys, Vs - Ws, A - B)$,
 $a_1(Xs, Ys), a_2(Vs, Ws)$.

oder einfacher $a(A, B) \rightarrow a_1(A, C), a_2(C, B)$.

Beispielprogramm als Prolog-Klauseln:

```
satz(S,R):-nominalphrase(S,VP),verbalphrase(VP,R)
nominalphrase(NP,R):-artikel(NP,N),nomen(N,R).
verbalphrase(VP,R):-verb(VP,R).
verbalphrase(VP,R):-verb(VP,NP),nominalphrase(NP,R).
artikel([a|R],R).
artikel([the|R],R).
nomen([cat|R],R).
nomen([mouse|R],R).
verb([scares|R],R).
verb([hates|R],R).
```

? - *satz([the, cat, scares, the, mouse], []).* Antwort: Yes
 ? - *satz(X, []).* X=[a,cat,scares]; [a cat hates]; ...

Index

- $DOM(\sigma)$, 11
- \mathcal{V} , 9
- μ -rekursiv, 40
- transp*, 37

- Anfrage, 31
- arg/3, 75
- assert, 76
- asserta, 76
- assertz, 76
- atomaren Formeln, 10
- atomic, 73

- Berechnung, 34
- Berechnung, kanonische, 46

- clause, 76

- definit, 26
- Domain, 11
- dynamic, 76
- dynamisch, 76

- erfolgreich, 34

- fail, 68
- Fakten, 31
- Fixpunkt, 37
- Fixpunkt Semantik, 39
- Fixpunktsatz, 39
- Folgerbarkeit, 14
- Formeln, 10
- Formeln, atomar-, 10
- frei, 10
- functor/3, 75
- Funktionssymbol, 9
- Funktionssymbol, Deutung-, 12
- Funktoren, 57

- geschlossen, 10
- Gilmore, Algorithmus von, 20
- Grundinstanz, 11
- Grundsubstitution, 11
- Grundterme, 9

- Herbrand-Expansion, 19
- Herbrand-Modell, 18
- Herbrand-Strukturen, 18
- Hornklausel, 26

- input-Resolution, 26
- Instanz, 11
- Interpretation, 11

- kanonische Berechnung, 46
- Kette, 38
- Klausel, 21
- Klauseln, 4
- kleinste obere Schranke, 38
- KNF, 21
- Kompaktheitssatz, 20
- Konfiguration, 34
- Konfigurationen, 33
- Konstante, 9
- Kopf, 6

- least fixpoint, 39
- least upper bound, 38
- lfp, 39
- lineare Resolution, 23
- Listen, 55
- Literale, 21
- Logikprogramm, 31
- lub, 38

- Meta-Interpreter, 79

monoton, 38

Negat, 21
Negation as failure, 68
negative Hornklausel, 26
not, 68
number, 73

Operatoren, 57
Ordnung, 37

Prädikatensymbol, 9
Prädikatssymbol, Deutung-, 12
Pränex-Normalform, 16
primitiv rekursiv, 41
Programmklausel, 31

quantorfrei, 10

read/1, 71
Rechenschritt, 34
reflexive Ordnung, 37
Regeln, 6, 31
Resolution, lineare, 23
Resolvent, 22
retract, 76
Rumpf, 6

Signatur, 9
Skolem-Normalform, 17
SLD-Baum, 48
stetig, 38
Strukturelle Induktion, 13
Substitution, 11
Substitutionslemma, 13

Terme, 9
Träger, 12
Turing-vollständig, 40

var, 73
Variablen, 5
Variablenumbenennung, 11
vollständig, 38

write, 70