Vorlesung

Logikprogrammierung

Prof. Klaus Indermark

WS 2001/2002

Inhaltsverzeichnis

Kapitel 1

Einleitung

Programmiersprachen kann man grob folgendermaßen einteilen:

- imperativ
- deklarativ
  - ◇ funktional
  - ◇ logisch

Deklarativ bedeutet dabei die Beschreibung eines Problems durch

- Funktionen (deterministisch)
- Relationen (nicht-deterministisch)

Logikprogrammierung stellt dann die Beschreibung eines Problemwissens durch logische Formeln dar:

- **Problemwissen:** Eigenschaften und Beziehungen von Objekten.
- **Formeln:** Definite *Horn*-Formeln (*Fakten* und *Regeln*)

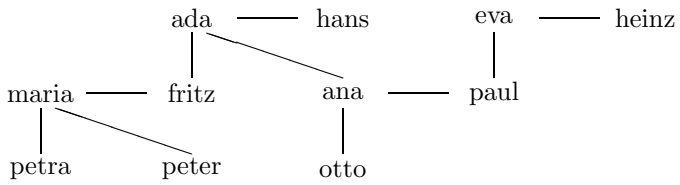
Ein Logikprogramm soll dann Anfragen (negativen Horn-Formeln) beantworten. Die Berechnung von Antworten erfolgt durch Resolution mit Unifikation.

Anwendung der Logikprogrammierung

- Deduktive Datenbanken
- Künstliche Intelligenz
- Expertensysteme
- Rapid Prototyping

Beispiel:

Personen:	ada, eva, maria, ana, petra, hans, heinz, fritz, paul, peter, otto	
Eigenschaften:	maennlich, weiblich	(1-stellig)
Beziehungen:	verheiratet (—), istMutterVon(|)	(2-stellig)




```
weiblich(ana).          verheiratet(maria, fritz).
⋮                       ⋮

maennlich(hans).        istMutterVon(eva, paul).
⋮                       ⋮
```

Beschreibung weiterer Beziehungen durch *Regeln*. Hilfsmittel sind *Variablen* und *logische Verknüpfungen*.

V ist Vater von K, wenn V und M verheiratet sind und M Mutter von K ist.

formal:

$$\text{verheiratet(V, M)} \wedge \text{istMutterVon(M, K)} \rightarrow \text{istVaterVon(V, K)}$$

PROLOG:

```
istVaterVon(V, K) :- verheiratet(M, V),
                     istMutterVon(M, K).

istGrossvaterVon(G, E) :- istVaterVon(G, V),
                          istVaterVon(V, E).

istGrossvaterVon(G, E) :- istVaterVon(G, M),
                          istMutterVon(M, E).
```

Anfragen in PROLOG

- `?- weiblich(eva).` ⇝ `Yes.`
- `?- weiblich(adam).` ⇝ `No.`
- `?- istVaterVon(paul, otto).` ⇝ `Yes.`
- `?- istGrossvaterVon(hans, E).` ⇝ `E = petra;`
  ⇝ `E = peter;`
  ⇝ `E = otto;`
  ⇝ `No.`

Kapitel 2

Prädikatenlogik, Resolution, Unifikation

Die Grundlage dieses Kapitels ist die formale Sprache FO[1], die Aussagen über Strukturen macht.

Struktur

- Menge von Objekten ("Universum")
- Beziehungen (auch Eigenschaften) zwischen (von) Objekten
  - ◇ funktional (deterministisch)
  - ◇ relational (nicht-deterministisch)

2.1 Syntax der Prädikatenlogik

Signatur: $\tau = \bigcup_{n\in\mathbb{N}} R^n(\tau) \cup \bigcup_{n\in\mathbb{N}} F^n(\tau)$

- $R^n(\tau)$: Menge der n-stelligen *Relationssymbole*
- $F^n(\tau)$: Menge der n-stelligen *Funktionssymbole*
- $F^0(\tau)$: Menge der *Konstantensymbole*
- $VAR := \{v_0, v_1, \ldots\}$: Menge der *Variablen* (abzählbar unendlich)

Definition 2.1 *Sei $X \subseteq VAR$. Dann ist die Menge $T(\tau, X)$ der* ***τ-Terme über*** *X induktiv definiert durch*

(1) $X \subseteq T(\tau, X)$

(2) $f \in F^n(\tau), t_1, \ldots, t_n \in T(\tau, X) \curvearrowright ft_1 \ldots t_n \in T(\tau, X)$

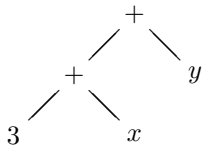
Beachte:

$F^0(\tau) \subseteq T(\tau, X)$
$T(\tau, \emptyset)$: Menge der **τ-Grundterme**

[1] first order logic

Notation von Termen

- klammerfreie Präfixnotation: $++3xy$
- Präfixnotation mit Klammern und Kommata: $+(+(3,x),y)$
- Infixnotation: $((3+x)+y)$
- Baumdarstellung:



Definition 2.2 *Sei τ eine Signatur und $X \subseteq VAR$. Dann ist die Menge $FO(\tau, X)$ der τ-**Formeln der Prädikatenlogik über** X induktiv definiert durch*

(1) $P \in R^n(\tau), t_1, \ldots, t_n \in T(\tau, X) \curvearrowright Pt_1 \ldots t_n \in FO(\tau, X)$

(2) $\varphi, \psi \in FO(\tau, X) \curvearrowright \neg\varphi, (\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \to \psi) \in FO(\tau, X)$

(3) $x \in X, \varphi \in FO(\tau, X) \curvearrowright \exists x\varphi, \forall x\varphi \in FO(\tau, X)$

Freie und gebundene Variablen

Ein Vorkommen von x in φ heißt **gebunden**, falls x in einer Teilformel der Form $\forall x\varphi, \exists x\varphi$, andernfalls **frei**.

Bezeichnung:

$Var(t), Var(\varphi)$: Menge der in t bzw. φ auftretenden Variablen
$Frei(\varphi)$: Menge der in φ frei vorkommenden Variablen
Ist $Frei(\varphi) = \emptyset$, so heißt φ ein τ-Satz.

2.2 Semantik der Prädikatenlogik

τ-Struktur $\mathfrak{A} = \langle A; P_1^{\mathfrak{A}}, \ldots, f_1^{\mathfrak{A}}, \ldots \rangle$

- A Menge: Universum
- $P \in R^n(\tau) \curvearrowright P^{\mathfrak{A}} \subseteq A^n$ (n-stellige Relation über A)
- $f \in F^n(\tau) \curvearrowright f^{\mathfrak{A}} : A^n \longrightarrow A$ (n-stellige Funktion/Operation über A)

Definition 2.3 *Eine τ-Struktur $\mathfrak{T} = \langle T; P_1^{\mathfrak{T}}, \ldots, f_1^{\mathfrak{T}}, \ldots \rangle$ heißt **frei** oder **Herbrand-Struktur**, falls*

(1) $T = T(\tau, \emptyset)$

(2) $f \in F^n(\tau) \curvearrowright f^{\mathfrak{T}} : T^n \longrightarrow T$, *mit* $f^{\mathfrak{T}}(t_1, \ldots, t_n) := ft_1 \ldots t_n$

*$f^{\mathfrak{T}}$ heißt auch **Konstruktorfunktion**.*

Zur Interpretation von τ-Formeln über X benötigt man neben einer τ-Struktur $\mathfrak{A}$ wegen frei auftretender Variablen noch eine Variablenbelegung $\beta : X \longrightarrow A$.

$\mathfrak{I} := (\mathfrak{A}, \beta)$ heißt (τ, X)-Interpretation.
$\mathfrak{I}$ bestimmt für jedes $t \in T(\tau, X)$ einen Wert $[\![t]\!]^{\mathfrak{I}} \in A$ und für jedes $\varphi \in FO(\tau, X)$ einen Wahrheitswert $[\![\varphi]\!]^{\mathfrak{I}} \in \{0, 1\}$.

Interpretation von Termen

- $[\![x]\!]^{\mathfrak{I}} := \beta(x)$
- $[\![ft_1 \dots t_n]\!]^{\mathfrak{I}} := f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{I}}, \dots, [\![t_n]\!]^{\mathfrak{I}})$

Interpretation von Formeln

- $[\![Pt_1 \dots t_n]\!]^{\mathfrak{I}} := \begin{cases} 1, & falls\ ([\![t_1]\!]^{\mathfrak{I}}, \dots, [\![t_n]\!]^{\mathfrak{I}}) \in P^{\mathfrak{A}} \\ 0, & sonst \end{cases}$
- Junktoren und Quantoren wie üblich

Sprech- und Schreibweisen

Sei $\varphi \in FO(\tau, X)$ und $\mathfrak{I} = (\mathfrak{A}, \beta)$ eine (τ, X)-Interpretation. Wenn $[\![\varphi]\!]^{\mathfrak{I}} = 1$, so sagt man

- $\mathfrak{I}$ ist ein **Modell** von φ
- $\mathfrak{I}$ **erfüllt** φ
- φ gilt/ist gültig in $\mathfrak{I}$

und man schreibt: $\boxed{\mathfrak{I} \models \varphi}$, $\mathfrak{A} \models \varphi$ für $Frei(\varphi) = \emptyset$

Definition 2.4 *Sei $\Phi \subseteq FO(\tau, X)$. Φ heißt **erfüllbar**, wenn Φ ein Modell beitzt, d. h. es existiert eine (τ, X)-Interpretation $\mathfrak{I}$, sodass für alle $\varphi \in \Phi$ gilt: $\mathfrak{I} \models \varphi$ (kurz: $\mathfrak{I} \models \Phi$); andernfalls heißt Φ **unerfüllbar**.*
*Φ heißt **allgemeingültig**, wenn jede (τ, X)-Interpretation Φ erfüllt.*

Definition 2.5 (Semantische Folgerungsbeziehung) *Sei $\Phi \subseteq FO(\tau, X)$ und $\psi \in FO(\tau, X)$. Man sagt: ψ **folgt aus** Φ (kurz: $\Phi \models \psi$), falls für alle (τ, X)-Interpretation $\mathfrak{I}$ gilt: $\mathfrak{I} \models \Phi \curvearrowright \mathfrak{I} \models \psi$.*
*$Cons(\Phi) := \{\psi \in FO(\tau, X) \mid \Phi \models \psi\}$ heißt **Folgerungsmenge** von Φ.*

Anwendung auf die Logikprogrammierung

$\Phi = \{\varphi_1, \dots, \varphi_k\}$ ist Logikprogramm mit $Cons(\Phi)$ als Semantik.
ψ Anfrage an Φ: Gilt ψ in Φ?
Formal: Gilt $\Phi \models \psi$? (Ist $\psi \in Cons(\Phi)$?)

Algorithmische Behandlung dieser Fragestellung

(1) Reduktion des Folgerungsproblems auf ein Unerfüllbarkeitsproblem

(2) Bestimmung der Unerfüllbarkeit durch Resolution

Lemma 2.1 *Für* $\{\varphi_1, \ldots, \varphi_k, \psi\} \subseteq FO(\tau, X)$ *gilt:* $\{\varphi_1, \ldots, \varphi_k\} \models \psi \curvearrowleft\!\curvearrowright \varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\psi$ *unerfüllbar*

Beweis:

$$\begin{aligned}\{\varphi_1, \ldots, \varphi_k\} \models \psi &\curvearrowleft\!\curvearrowright \text{für alle } \mathfrak{I} : \mathfrak{I} \models \{\varphi_1, \ldots, \varphi_k\} \curvearrowright \mathfrak{I} \models \psi \\ &\curvearrowleft\!\curvearrowright \text{für alle } \mathfrak{I} : \mathfrak{I} \models \varphi_1 \wedge \ldots \wedge \varphi_k \to \psi \\ &\curvearrowleft\!\curvearrowright \neg(\varphi_1 \wedge \ldots \wedge \varphi_k \to \psi) \text{ unerfüllbar} \\ &\curvearrowleft\!\curvearrowright (\varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\psi) \text{ unerfüllbar}\end{aligned}$$

q.e.d.

Aufgabe:

Übertragung der aussagenlogischen Resolution auf die Prädikatenlogik (FO-Resolution)

Problem:

Es gibt unendlich viele Interpretationsmöglichkeiten von (τ, X) durch $(\mathfrak{A}, \beta)$

Folgerung:

- Unerfüllbarkeit wird unentscheidbar.
- Unerfüllbarkeit bleibt rekursiv aufzählbar, d. h. semi-entscheidbar.

Idee:

(1) $\varphi \in FO(\tau, X)$ in Skolemnormalform bringen

(2) (τ, X)-Interpretation auf Herbrand-Interpretation beschränken

(3) aussagenlogische Resolution mit Unifikation zu der FO-Resolution erweitern

2.3 Skolemnormalform und Herbrand-Expansion

Ziel: Normalisierung bezüglich der Quantoren

Definition 2.6 *Sei* $\varphi \in FO(\tau, X)$. *Dann heißt* φ *in* ***Skolemnormalform (SNF)*** *(kurz:* $\varphi \in SNF(\tau, X)$*), wenn gilt:*

(1) $Frei(\varphi) = \emptyset$

(2) $\varphi = \forall x_1 \ldots x_n \varphi'$

(3) φ' *ist quantorenfrei*

Beachte:

$\varphi \in FO(\tau, \{x_1, \ldots, x_n\})$

Satz 2.1 *Jedes* $\varphi \in FO(\tau, X)$ *lässt sich in ein erfüllbarkeitsäquivalentes* $\psi \in SNF(\sigma, X)$ *transformieren, wobei* σ *eine Erweiterung von* τ *um geeignete Funktionssymbole ist.*

Bemerkung:

φ und ψ heißen **erfüllbarkeitsäquivalent** :$\curvearrowright\!\!\curvearrowleft$ (φ erfüllbar $\curvearrowright\!\!\curvearrowleft$ ψ erfüllbar)

Beweisidee:

(1) Pränexnormalform: Herausziehen innerer Quantoren

(2) Skolemnormalform: Eliminierung von $\exists$-Quantoren durch neue Fuktionssymbole ($\rightsquigarrow$ erfüllbarkeitsäquivalent)

(3) Ersetzen freier Variablen durch neue Konstantensymbole

Problem:

Im Unterschied zur Aussagenlogik gibt es unendlich viele Interpretationen

Vereinfachung:

Reduktion auf Herbrand-Strukturen

Substitution

Sei $s : X \longrightarrow T(\tau, X)$ eine Variablenbelegung durch Terme. Diese lässt sich fortsetzen zu einer Substitution auf Formeln:
$\widehat{s} : FO(\tau, X) \longrightarrow FO(\tau, X)$: jedes freie Vorkommen eines $x \in X$ in φ wird durch $s(x)$ ersetzt. Dabei: geeignete Umbenennungen gebundener Variablen von φ zur Vermeidung von Variablenkonflikten.

Spezialfall:

Seien $x_1, \ldots, x_n \in X$ paarweise verschieden und $t_1, \ldots, t_n \in T(\tau, X)$, sodass $s(x_i) = t_i$ und $s(x) = x$ für $x \notin \{x_1, \ldots, x_n\}$, dann schreibt man $s = [x_1/t_1, \ldots, x_n/t_n]$ und $\varphi[x_1/t_1, \ldots, x_n/t_n]$ statt $\widehat{s}(\varphi)$.

Lemma 2.2 (Substitutionslemma) *Syntaktische und semantische Substitution können wie folgt vertauscht werden: Sei $\mathfrak{I} = (\mathfrak{A}, \beta)$ eine (τ, X)-Interpretation und $s : X \longrightarrow T(\tau, X)$. Dann bezeichne $\mathfrak{I} \circ s$ die (τ, X)-Interpretation $(\mathfrak{A}, \beta_s)$ mit $\beta_s(x) = [\![s(x)]\!]^{\mathfrak{I}}$ und es gilt für jedes $t \in T(\tau, X)$ und $\varphi \in FO(\tau, X)$:*

(1) $[\![t]\!]^{\mathfrak{I} \circ s} = [\![\widehat{s}(t)]\!]^{\mathfrak{I}}$

(2) $\mathfrak{I} \circ s \models \varphi \curvearrowright\!\!\curvearrowleft \mathfrak{I} \models \widehat{s}(\varphi)$

Definition 2.7 *Eine τ-Struktur $\mathfrak{A}$ bestimmt eine **Herbrand-Struktur***

$$\mathfrak{T}_{\mathfrak{A}} = \langle T, P_1^{\mathfrak{T}_{\mathfrak{A}}}, \ldots, f_1^{\mathfrak{T}_{\mathfrak{A}}}, \ldots \rangle$$

durch

$$(t_1, \ldots, t_n) \in P_n^{\mathfrak{T}_{\mathfrak{A}}} \curvearrowright\!\!\curvearrowleft ([\![t_1]\!]^{\mathfrak{A}}, \ldots, [\![t_n]\!]^{\mathfrak{A}}) \in P_n^{\mathfrak{A}} \qquad (*)$$

für alle Terme und Relationssymbole.

Satz 2.2 *Für eine Formel $\varphi \in SNF(\tau, X)$ und eine Struktur $\mathfrak{A}$ gilt:*

$$\mathfrak{A} \models \varphi \curvearrowright \mathfrak{T}_{\mathfrak{A}} \models \varphi$$

Beweis:

durch Induktion über $n \in \mathbb{N}$: $\varphi = \forall x_1 \ldots \forall x_n \varphi'$

I.A.: $n = 0$: $\varphi = \varphi'$ quantorenfrei ✓
$\mathfrak{A} \models \varphi \curvearrowright \mathfrak{A} \models \varphi' \curvearrowright \mathfrak{T}_\mathfrak{A} \models \varphi'$ wegen $(*)$

I.S.: $\varphi = \forall x_1 \ldots \forall x_n \forall x_{n+1} \varphi'$
$\varphi_n := \forall x_1 \ldots \forall x_n \varphi'$ kann x_{n+1} frei enthalten
Sei $\mathfrak{A} \models \varphi$
$\curvearrowright$ für alle $a \in A$ gilt: $(\mathfrak{A}, \beta[x_{n+1}/a]) \models \varphi_n$
$\curvearrowright$ für alle $t \in T$ gilt: $(\mathfrak{A}, \beta[x_{n+1}/[\![t]\!]^\mathfrak{A}]) \models \varphi_n$
$\curvearrowright$ für alle $t \in T$ gilt: $\mathfrak{A} \models \varphi_n[x_{n+1}/t]$ (Lemma 2.2)
$\curvearrowright$ für alle $t \in T$ gilt: $\mathfrak{T}_\mathfrak{A} \models \varphi_n[x_{n+1}/t]$ (I.V.)
$\curvearrowright$ für alle $t \in T$ gilt: $(\mathfrak{T}_\mathfrak{A}, \beta[x_{n+1}/t] \models \varphi_n$ (Lemma 2.2)
$\curvearrowright$ für alle $t \in T$ gilt: $\mathfrak{T}_\mathfrak{A} \models \varphi$

q.e.d.

Korollar 2.1 *Für $\varphi \in SNF(\tau, X)$ gilt: φ erfüllbar $\curvearrowright$ φ besitzt eine Herbrand-Struktur.*

Bemerkung:

Es genügt für die Erfüllbarkeit einer Formel in SNF Herbrand-Strukturen zu betrachten.

Weitere Reduktion von FO-Erfüllbarkeit auf die aussagenlogische Erfüllbarkeit unendlich vieler Formeln.

Idee:

Expansion einer Formel $\varphi \in SNF(\tau, X)$ durch **Grundsubstitution**

Definition 2.8 *Für $\varphi = \forall x_1 \ldots \forall x_n \varphi' \in SNF(\tau, X)$ heißt*

$$E(\varphi) := \{\varphi'[x_1/t_1, \ldots, x_n/t_n] \mid t_i \in T\}$$

***Herbrand-Expansion** von φ.*

Beachte:

$\psi \in E(\varphi)$ enthält keine Variablen mehr und hat die Gestalt einer aussagenlogischen Formel: $Pt_1 \ldots t_n$ anstelle der A_i.

Satz 2.3 (Reduktion der Erfüllbarkeit von Formeln in SNF) *Für $\varphi \in SNF(\tau, X)$ gilt: φ ist erfüllbar $\curvearrowright$ $E(\varphi)$ ist erfüllbar.*

Beweis:

Sei $\varphi = \forall x_1 \ldots \forall x_n \varphi' \in SNF(\tau, X)$, φ erfüllbar
$\curvearrowright$ es gibt Herbrand-Struktur $\mathfrak{T}$ mit $\mathfrak{T} \models \varphi$
$\curvearrowright$ es gibt Herbrand-Struktur $\mathfrak{T}$, sodass für alle $t_1, \ldots, t_n \in T$ gilt: $\mathfrak{T} \models \varphi'[x_1/t_1, \ldots, x_n/t_n]$
$\curvearrowright$ es gibt Herbrand-Struktur $\mathfrak{T}$ mit $\mathfrak{T} \models E(\varphi)$

q.e.d.

Bemerkung:

Ersetzt man in $E(\varphi)$ die atomaren Formeln $Pt_1 \dots t_n$ durch aussagenlogische Variablen, so gilt für das Ergebnis $E(\varphi)_{AL}$:
$\psi \in E(\varphi)$ FO-erfüllbar $\curvearrowright$ $\psi \in E(\varphi)_{AL}$ AL-erfüllbar

Algorithmus von Gilmore

Problem:

Ist $\varphi \in FO(\tau, X)$ unerfüllbar?

Verfahren:

(1) Transformiere φ in erfüllbarkeitsäquivalentes $\psi \in SNF(\sigma, X)$
(2) Wähle Aufzählung $\{\psi_1, \psi_2, \dots\} = E(\psi)_{AL}$ der Herbrand-Expansion von ψ und ersetze dabei die atomaren Formeln durch aussagenlogische Variablen
(3) Prüfe, ob ψ, $\psi_1 \wedge \psi_2$, $\psi_1 \wedge \psi_2 \wedge \psi_3$, ... AL-erfüllbar
(4) Abbrechen, falls $\psi_1 \wedge \dots \wedge \psi_k$ unerfüllbar

Korrektheit:

φ unerfüllbar
$\curvearrowright$ $E(\psi)_{AL}$ AL-erfüllbar
$\curvearrowright$ es gibt eine endliche Teilmenge von $E(\psi)_{AL}$, welche unerfüllbar ist (Kompaktheitssatz)
$\curvearrowright$ es gibt $k \in \mathbb{N}$, sodass $\psi_1 \wedge \dots \wedge \psi_k$ unerfüllbar

q.e.d.

2.4 Resolution und Unifikation

Ziel: effizienter Erfüllbarkeitstest

Methode: Übergang zu Klauselmengen, FO-Resolution

Definition 2.9 *Sei* $\varphi = \forall x_1 \dots \forall x_n \varphi' \in SNF(\tau, X)$
O.B.d.A sei φ' *in KNF, also* $\varphi' = \big((L_{11} \vee \dots \vee L_{1n_1}) \wedge \dots \wedge (L_{m1} \vee \dots \vee L_{mn_m})\big)$ *und* $L_{ij} = Pt_1 \dots t_n \in FO(\tau, \{x_1, \dots, x_n\})$ *oder* $L_{ij} = \neg Pt_1 \dots t_n \in FO(\tau, \{x_1, \dots, x_n\})$
φ' *bestimmt somit die* ***Klauselmenge***

$$\mathcal{K}(\varphi') = \big\{ \underbrace{\{L_{11}, \dots, L_{1n_1}\}}_{\tau - Klausel\ über\ X}, \dots, \{L_{m1}, \dots, L_{mn_m}\} \big\}$$

Für $\psi \in E(\varphi)$ *existiert eine* ***Grundsubstitution*** *für* $x_1, \dots, x_n$*:* $s = [x_1/t_1, \dots, x_n/t_n]$,- $t_i \in T$*, sodass* $\psi = \varphi' s$*.* ψ *heißt* ***Grundinstanz*** *von* φ'*.*

Da φ' in KNF ist, gilt dies auch für jede Grundinstanz von $\varphi' s \in E(\varphi)$. $\varphi' s$ bestimmt die Klauselmenge $\mathcal{K}(\varphi' s)$.
$\rightsquigarrow$ Anwendbarkeit des Resolutionskalküls

Lemma 2.3 *Sei $\varphi = \forall x_1 \ldots \forall x_n \varphi' \in SNF(\sigma, X)$ und φ' in KNF. Dann gilt: φ unerfüllbar $\curvearrowright$ es gibt ein $r \in \mathbb{N}$, Klauseln $K_1, \ldots, K_r \in \mathcal{K}(\varphi')$ und Grundsubstitutionen $s_1, \ldots, s_r$ für $x_1, \ldots, x_n$, sodass $\{K_1 s_1, \ldots, K_r s_r\}$ unerfüllbar.*

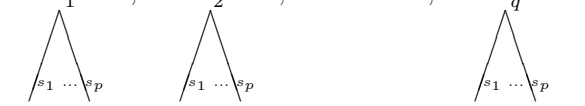
Beachte:

Weder K_i noch s_i müssen paarweise verschieden sein.

Bemerkung:

Geschickte Auswahl von Klauseln und Grundsubstitutionen genügt.

$\mathcal{K}(\varphi')$: K'_1 , K'_2 , $\ldots$, K'_q



$K'_1 s_1 \ldots K'_1 s_p$ $K'_2 s_1 \ldots K'_2 s_p$ $K'_q s_1 \ldots K'_q s_p$

Beweis:

φ unerfüllbar
$\curvearrowright$ $E(\varphi)$ unerfüllbar
$\curvearrowright$ es existiert $p \in \mathbb{N}, \psi_1, \ldots, \psi_p \in E(\varphi)$, sodass $\{\psi_1, \ldots, \psi_p\}$ unerfüllbar (Kompaktheitssatz)
$\curvearrowright$ es existiert $p \in \mathbb{N}$, Grundsubstitutionen $s_1, \ldots, s_p$, sodass $\{\varphi' s_1, \ldots, \varphi' s_p\}$ unerfüllbar
$\curvearrowright$ es existiert $p \in \mathbb{N}$, Grundsubstitutionen $s_1, \ldots, s_p$, sodass $\bigcup_{i=1}^{p} \mathcal{K}(\varphi') s_i$ unerfüllbar
$\curvearrowright$ es existiert $r \in \mathbb{N}$, Grundsubstitutionen $s_1, \ldots, s_r$ und Klauseln $K_1, \ldots, K_r \in \mathcal{K}(\varphi')$, sodass $\{K_1 s_1, \ldots, K_r s_r\}$ unerfüllbar

Grund: Ist $M \subseteq \mathcal{K}(\varphi')$ unerfüllbar so auch $\mathcal{K}(\varphi')$

q.e.d.

Korollar 2.2 (Grundresolutionssatz) *Sei $\varphi = \forall x_1 \ldots \forall x_n \varphi' \in SNF(\tau, X)$, φ' in KNF. Dann gilt: φ ist unerfüllbar $\curvearrowright$ es existieren Klauseln $K_1, \ldots, K_r$ mit $K_r = \square$ und für alle $i = 1, \ldots, r$ gilt: K_i ist Grundinstanz einer Klausel $K \in \mathcal{K}(\varphi')$ oder: K_i ist (AL-) Resolvente von K_j und K_l, $j, l < i$.*

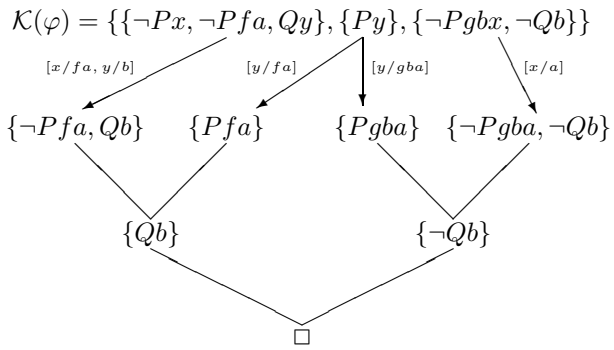
Beweis:

Aus Lemma 2.3 folgt: φ unerfüllbar $\curvearrowright$ $\{K_1 s_1, \ldots, K_r s_r\}$ unerfüllbar $\curvearrowright$ $\square \in Res^*(\{K_1 s_1, \ldots, K_r s_r\})$

q.e.d.

Beispiel:

$\varphi = \forall x \forall y((\neg Px \vee \neg Pfa \vee Qy) \wedge (Py) \wedge (\neg Pgbx \vee \neg Qb))$
$\tau = \{P^{(1)}, Q^{(1)}, g^{(2)}, f^{(1)}, a^{(0)}, b^{(0)}\}$



Problem:

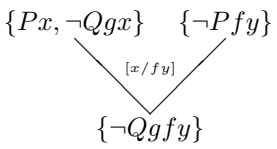
Wie findet man geeignete Grundsubstitutionen für die Herleitung der leeren Klausel ($\square$)?

Strategie:

- "zurückhaltend" substituieren, um Möglichkeiten offen zu halten
- Grundsubstitutionen aufbrechen durch Substitution beliebiger Terme (z. B. $[x/fga] \to [x/fx] \circ [x/gx] \circ [x/a]$)

$\rightsquigarrow$ FO-Resolution (J. A. Robinson),
Kombination von Resolution mit Substitution

Beispiel:



Sprechweise:

$[x/fy]$ *unifiziert* die Literale Px und Pfy

Definition 2.10 *Sei $K = \{L_1, \ldots, L_p\}$ eine τ-Klausel über X, d.h. L_i ist eine atomare Formel oder ihre Negation. Sei $s : X \longrightarrow T(\tau, X)$ eine Substitution. Dann heißt s ein **Unifikator** von $K :\curvearrowright\curvearrowright L_1 s = L_2 s = \ldots = L_p s$ (oder $|Ks| = 1$). Man sagt K ist mit s **unifizierbar**.*
*Ferner heißt s **allgemeinster Unifikator** von K, falls für jeden Unifikator $s' : X \longrightarrow T(\tau, X)$ eine Substitution $s'' : X \longrightarrow T(\tau, X)$ existiert mit $s' = s \circ s''$ (mit $[s \circ s''](x) = \widehat{s''}\big(s(x)\big)$)*

Folgerung

Sind s_1 und s_2 allgemeinste Unifikatoren von K, so gibt es eine Variablenumbenennung $s : X \longrightarrow X$ mit $s_2 = s_1 \circ s$.

Satz 2.4 (Unifikationssatz – Robinson) *Es ist entscheidbar, ob eine nicht-leere τ-Klausel K über X unifizierbar ist oder nicht. Eine unifizierbare Klausel besitzt einen allgemeinsten Unifikator, der sich berechnen lässt.*

Beweis:

durch Konstruktion: **Unifikationsalgorithmus**
Eingabe: $K = \{L_1, \ldots, L_p\}$ τ-Klausel über X, $p \geq 1$
Sei `sub` eine Variable vom Typ $X \longrightarrow T(\tau, X)$

```
        sub := [ ];   //identische Substitution
        while(|Ksub| > 1) do   //noch keine Unifikation
            lese alle L_i sub parallel von links nach rechts bis in
            zwei Literalen die gelesenen Zeichen verschieden sind;
            if(keines der beiden Zeichen ist Variable) then
(s1)            stop; Ausgabe = 'K nicht unifizierbar'
            else
                Sei x die Variable und t der Teilterm im anderen Literal
                mit Kopf an gelesener Stelle.
                if(x kommt in t vor) then   //"occur check"
(s2)                stop; Ausgabe = 'K nicht unifizierbar'
                else
                    sub := sub ∘ [x/t]
                end if;
            end if;
        end while;
(s3)    stop; Ausgabe = 'sub ist allgemeinster Unifikator von K'
```

Korrektheit:

(1) Der Algorithmus terminiert für jede Eingabe, weil die Zahl der in K`sub` vorhandenen Variablen nach jedem Schleifendurchlauf um 1 abnimmt.

(2) Der Algorithmus liefert eine korrekte Entscheidung, genauer:

(a) K unifizierbar $\curvearrowleft\curvearrowright$ `stop` bei (s3) mit `sub` als allgemeinstem Unifikator von K

(b) K nicht unifizierbar $\curvearrowleft\curvearrowright$ `stop` bei (s1) oder (s2)

Wegen (1) gilt (a) $\curvearrowright$ (b)

Zu (a) "$\curvearrowleft$": klar, weil beim Verlassen der `while`-Schleife $|K\texttt{sub}| = 1$

Noch zu zeigen (a) "$\curvearrowright$":

(L) Sei $s : X \longrightarrow T(\tau, X)$ mit $|Ks| = 1$. Sei sub_i der Wert von `sub` nach i-tem Durchlauf. Dann gilt für alle $i \in \mathbb{N}$: Wird die Schleife i-mal durchlaufen, so existiert $s_i : X \longrightarrow T(\tau, X)$ mit $s = sub_i \circ s_i$.

Aus (L) folgt:
K unifizierbar $\curvearrowright$ `stop` bei (s3) mit `sub` als allgemeinstem Unifikator von K, denn nach i-tem Durchlauf gilt: $Ks = Ksub_i \circ s_i$, also ist auch $Ksub_i$ unifizierbar. Bei erneutem Durchlauf wegen $|Ksub_i| > 1$ kann die Schleife weder bei (s1) noch bei (s2) verlassen werden. Also `stop` bei (s3) mit $s = suk_k$, falls dies nach k Durchläufen passiert. Der Wert von sub_k ist folglich allgemeinster Unifikator von K.

Beweis von (L):

durch Induktion über i

I.A.: $i = 0$: $sub_0 = [\ ]$, $s_0 := s$

$\curvearrowright s = sub_0 \circ s_0$

I.S.: Wird die Schleife $(i+1)$-mal durchlaufen, so auch i-mal und $sub_{i+1} = sub_i \circ [x/t]$.
$\curvearrowright$ es existiert s_i mit $s = sub_i \circ s_i$
Setze $s_{i+1}(y) := \begin{cases} s_i(y) & : \quad y \neq x \\ y & : \quad y = x \end{cases}$

$$\begin{aligned} \curvearrowright sub_{i+1} \circ s_{i+1} &= sub_i \circ [x/t] \circ s_{i+1} \\ &= sub_i \circ s_{i+1} \circ [x/ts_{i+1}] \\ &= sub_i \circ s_{i+1} \circ [x/ts_i] \\ &= sub_i \circ s_{i+1} \circ [x/xs_i] \\ &= sub_i \circ s_i \\ &= s \end{aligned}$$

q.e.d.

Wir gehen im Folgenden davon aus, das die Variablenmenge VAR mit X bezeichnet wird: $\boxed{X = VAR}$.

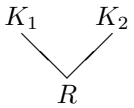
Definition 2.11 *Seien K_1, K_2 und R τ-Klauseln über X. Dann heißt R **Resolvent** (genauer FO-Resolvent) von K_1 und K_2, falls*

(1) es gibt Variablenumbenennungen s_1, s_2, sodass $K_1 s_1$ und $K_2 s_2$ keine gemeinsamen Variablen besitzen.

(2) es gibt $L_1, \ldots, L_n \in K_1 s_1$ $(n \geq 1)$ und $L'_1, \ldots, L'_m \in K_2 s_2$ $(m \geq 1)$, sodass $K := \{\overline{L_1}, \ldots, \overline{L_n}, L'_1, \ldots, L'_m\}$ unifizierbar mit allgemeinstem Unifikator s.

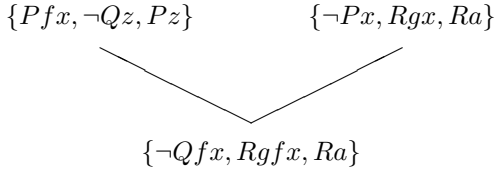
(3) $R = \big((K_1 s_1 \setminus \{L_1, \ldots, L_n\}) \cup (K_2 s_2 \setminus \{L'_1, \ldots, L'_m\})\big)s$

Schreibweise:



Oder ausführlicher durch unterstreichen der eliminierten Literale und Angabe der Umbenennungen sowie des Unifikators (mgu)

Beispiel:



$s_1 = [\,]$
$s_2 = [x/u]$
$s = [z/fx, u/fx]$

hierbei: $K = \{\neg Pfx, \neg Pz, \neg Pu\}$

Übertragung des Resolutionskalküls:

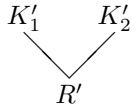
$$\underbrace{Res(\mathcal{K})}_{\text{Resolventenerweiterung}} \quad , \quad Res^n(\mathcal{K}) \quad , \quad \underbrace{Res^*(\mathcal{K})}_{\text{Resolutionshülle}}$$

Aussagenlogische Resolution von Grundinstanzen von FO-Klauseln können zur FO-Resolution "angehoben" werden. $\rightsquigarrow$ *FO-Resolutionskalkül*

Lemma 2.4 (Lifting-Lemma) *Seien K_1 und K_2 τ-Klauseln über X mit Grundinstanzen K_1' und K_2' und deren AL-Resolvente R'. Dann gibt es einen FO-Resolvent R von K_1 und K_2, sodass R' Grundinstanz von R ist:*

Beweis:

Wähle Variablenumbenennung s_1 und s_2, sodass $K_1 s_1$ und $K_2 s_2$ keine gemeinsame Variablen besitzen. K_i' Grundinstanz von $K_i \curvearrowright K_i'$ Grundinstanz von $K_i s_i$; wobei $K_i' = K_i s_i s_i'$ $(i = 1, 2)$. Wegen der Variablenumbenennung gibt es eine Substitution s mit $K_i' = K_i s_i s$.



$\curvearrowright$ $\exists L \in K_1',\ \exists \overline{L} \in K_2'$, sodass $R' = (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\})$

Wähle alle Urbilder von L bzw. $\overline{L}$ unter s, d. h. alle $L_1, \ldots, L_n \in K_1 s_1$ und $L_1', \ldots, L_m' \in K_2 s_2$ mit $L = L_1 s = \ldots = L_n s$ und $\overline{L} = L_1' s = \ldots = L_n' s$. s unifiziert also $\{\overline{L_1}, \ldots, \overline{L_n}, L_1', \ldots, L_m'\} = K$.

Sei s_0 allgemeinster Unifikator von K. Dann gilt: $R = \big((K_1 s_1 \setminus \{L_1, \ldots, L_n\}) \cup (K_2 s_2 \setminus \{L_1', \ldots, L_m'\})\big) s_0$ ist FO-Resolvent von K_1 und K_2. Da $s = s_0 s'$ mit geeigneter Substitution s', folgt:

$$
\begin{aligned}
R' &= (K_1' \setminus \{L\}) \cup (K_2' \setminus \{\overline{L}\}) \\
&= (K_1 s_1 s \setminus \{L\}) \cup (K_2 s_2 s \setminus \{\overline{L}\}) \\
&= \big((K_1 s_1 \setminus \{L_1, \ldots, L_n\}) \cup (K_2 s_2 \setminus \{L_1', \ldots, L_m'\})\big) s \\
&= \big((K_1 s_1 \setminus \{L_1, \ldots, L_n\}) \cup (K_2 s_2 \setminus \{L_1', \ldots, L_m'\})\big) s_0 s' \\
&= R s'
\end{aligned}
$$

q.e.d.

Satz 2.5 (Resolutionssatz der Prädikatenlogik) *Für $\varphi = \forall x_1 \ldots \forall x_n \varphi' \in SNF(\tau, X)$ mit φ' in KNF gilt: φ ist unerfüllbar $\curvearrowright\!\!\curvearrowright \square \in Res^*\big(\mathcal{K}(\varphi')\big)$.*

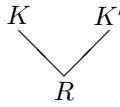
Beweis:

"$\curvearrowright$": (Vollständigkeit)
Sei φ unerfüllbar. Dann existiert ein Grundresolutionsbeweis von $\square$ in der Aussagenlogik. Nach Lifting-Lemma (Lemma 2.4) lässt sich dieser "anheben" zu einem FO-Resolutionsbeweis.

"$\curvearrowleft$": (Korrektheit)
Sei $\square \in Res^*\big(\mathcal{K}(\varphi')\big)$ und $\mathcal{K}(\varphi') = \{K_1, \ldots, K_r\}$. Für eine τ-Klausel $K = \{L_1, \ldots, L_s\}$ über $\{y_1, \ldots, y_k\} \subseteq X$ heißt

- $\forall K := \forall y_1, \ldots \forall y_k (L_1 \vee \ldots \vee L_s)$
- $\forall \square := 0$

All-Formel von K. K erfüllbar $\curvearrowleft\curvearrowright$ $\forall K$ erfüllbar.
Wir zeigen, dass für Klauseln K, K' und R gilt:

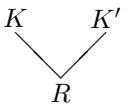
(!) $\quad$  $\quad \curvearrowright \ \forall K \wedge \forall K' \models \forall R$

Aus (!) folgt dann die Behauptung durch Widerspruchsbeweis wie folgt:

Sei φ erfüllbar $\curvearrowright$ es existiert eine τ-Struktur $\mathfrak{A}$, welche φ erfüllt ($\mathfrak{A} \models \varphi$). $\mathfrak{A} \models \forall K_1 \wedge \ldots \wedge \forall K_r$ und da $\square \in Res^*\big(\{K_1, \ldots, K_r\}\big)$ gilt auch $\mathfrak{A} \models \forall \square \curvearrowright$ Widerspruch $\curvearrowright$ Behauptung.

Beweis von (!):

durch Widerspruch:

Sei  und $\mathfrak{A} \models \forall K \wedge \forall K'$

Annahme: $\mathfrak{A} \not\models \forall R$

$R = \big((Ks \setminus \{L_1, \ldots, L_p\}) \cup (K's' \setminus \{L'_1, \ldots, L'_q\})\big)s_0$, wobei s_0 allgemeinster Unifikator von $\{\overline{L_1}, \ldots, \overline{L_p}, L'_1, \ldots, L'_q\}$. Da $\mathfrak{A} \not\models \forall R$, existiert ein $\beta : X \longrightarrow A$, mit $(\mathfrak{A}, \beta) \not\models R$.
$\curvearrowright (\mathfrak{A}, \beta) \not\models \big((Ks \setminus \{L_1, \ldots, L_p\}) \cup (K's' \setminus \{L'_1, \ldots, L'_q\})\big)s_0$
$\curvearrowright (\mathfrak{A}, \beta_{s_0}) \not\models (Ks \setminus \{L_1, \ldots, L_p\}) \cup (K's' \setminus \{L'_1, \ldots, L'_q\})$
$\curvearrowright (\mathfrak{A}, \beta_{s_0}) \not\models (Ks \setminus \{L_1, \ldots, L_p\})$ und $(\mathfrak{A}, \beta_{s_0}) \not\models (K's' \setminus \{L'_1, \ldots, L'_q\})$
Nach Voraussetzung gilt: $(\mathfrak{A}, \beta_{s_0}) \models Ks$ und $(\mathfrak{A}, \beta_{s_0}) \models K's'$.
$\curvearrowright (\mathfrak{A}, \beta_{s_0}) \models \{L_1, \ldots, L_p\}$ und $(\mathfrak{A}, \beta_{s_0}) \models \{L'_1, \ldots, L'_q\}$
$\curvearrowright (\mathfrak{A}, \beta) \models \{L_1, \ldots, L_p\}s := L$ und $(\mathfrak{A}, \beta) \models \{L'_1, \ldots, L'_q\}s' := L'$
$\curvearrowright \overline{L} = L' \curvearrowright$ Widerspruch $\curvearrowright \mathfrak{A} \models \forall R$

q.e.d.

2.5 Lineare Resolution, Horn-Klauseln, SLD-Resolution

Problem: viele Resolutionsmöglichkeiten $\rightsquigarrow$ "kombinatorische Explosion"

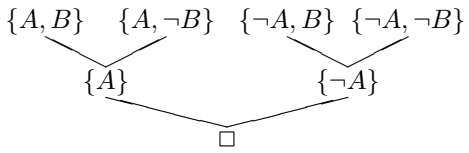
Ziel: Suchbaum verkleinern durch Beschränkung auf bestimmte Resolutionsstrategien (Vollständigkeit)

Ergebnis: Lineare Resolution ist vollständig für beliebige Klauselmengen. SLD-Resolution ist vollständig für Horn-Klausel-Mengen.

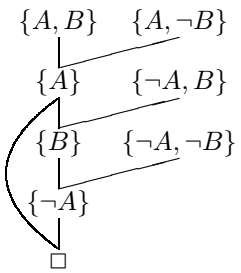
Definition 2.12 (Lineare Resolution) *Sei $\mathcal{K}$ eine Klauselmenge und $K \in \mathcal{K}$. Dann sagt man $\square$ ist aus K in $\mathcal{K}$ **linear resolvierbar**, falls eine Klauselfolge $K = K_0, K_1, \ldots, K_{n-1}, K_n$, mit $K_n = \square$ existiert, sodass für $i = 1, \ldots, n$ gilt: K_i ist Resolvent von K_{i-1} und $K' \in \{K_0, \ldots, K_{i-2}\} \cup \mathcal{K}$.*

Spezialfall: $\square \in \mathcal{K}$: $\square$ ist aus $\square$ linear resolvierbar.

Beispiel:



keine lineare Resolution



lineare Resolution

Satz 2.6 (Vollständigkeit der linearen Resolution) *Sei $\mathcal{K}$ eine unerfüllbare Klauselmenge. Dann existiert $K \in \mathcal{K}$, sodass $\square$ aus K in $\mathcal{K}$ linear resolvierbar ist.*

Beweis:

Sei $\mathcal{K}$ unerfüllbar $\curvearrowright \mathcal{K} \neq \emptyset$. Sei $\mathcal{K}_{min}$ eine minimal unerfüllbare Teilmenge von $\mathcal{K}$, d. h. für jede Klausel $K \in \mathcal{K}_{min}$ ist $\mathcal{K}_{min} \setminus \{K\}$ erfüllbar. Wir zeigen, dass für jedes $K \in \mathcal{K}_{min}$ $\square$ aus K in $\mathcal{K}$ linear resolvierbar ist.
Sei also $K \in \mathcal{K}_{min}$; Beweis durch Induktion über die Zahl n der aussagenlogischen Variablen in $\mathcal{K}_{min}$:

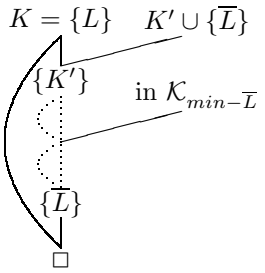
I.A.: $n = 0$: $\mathcal{K}_{min} = \{\square\} \curvearrowright K = \square$ ✓

I.S.: $\mathcal{K}_{min}$ enthalte $n+1$ aussagenlogische Variablen. Dann folgt:
$|K| \geq 1$, weil $\square \in \mathcal{K}_{min} \curvearrowright \mathcal{K}_{min} = \{\square\}$

(1) $|K| = 1 \curvearrowright K = \{L\}$ und $L \in VAR$ bzw. $\overline{L} \in VAR$. Entferne aus $\mathcal{K}_{min}$ alle Vorkommen von $\overline{L}$ und alle Klauseln, die L enthalten. Das Ergebnis $\mathcal{K}_{min-\overline{L}}$ ist ebenfalls unerfüllbar.

Sei $\mathcal{K}'_{min}$ eine minimal unerfüllbare Teilmenge von $\mathcal{K}_{min-\overline{L}}$. $\mathcal{K}'_{min}$ enthält ein K' mit $K' \cup \{\overline{L}\} \in \mathcal{K}_{min}$, denn anderenfalls wäre $\mathcal{K}'_{min} \subset \mathcal{K}_{min}$ erfüllbar.

Nach I.V. ist $\square$ aus K' in $\mathcal{K}_{min-\overline{L}}$ linear resolvierbar. Dieser lineare Resolutionsbeweis lässt sich passend erweitern:



Auffüllen der Klauseln mit $\overline{L}$
liefert Resolutionsbeweis in $\mathcal{K}$,
aber eventuell von $\{\overline{L}\}$.
Dann letzter Schritt mit $K = \{L\}$

(2) $|K| > 1$ ähnlich (siehe SCHÖNING)

q.e.d.

2.5.1 Horn-Logik

Für die Anwendung der FO-Resolution genügt es, sich auf die Horn-Logik zu beschränken.

Definition 2.13 *Eine **Horn-Formel** φ liegt dann vor, wenn φ in KNF ist und jede Disjunktion in φ höchstens ein positives Literal enthält. Entsprechend ist eine **Horn-Klausel** eine Klausel mit höchstens einem positiven Literal.*

Folgerung:

Eine Horn-Formel kann als Konjunktion von Implikationen geschrieben werden.

Beispiel:

$$\varphi := (A \vee \neg B) \wedge (\neg C \vee \neg A \vee D) \wedge (\neg A \vee \neg B) \wedge (D)$$
$$\equiv (B \rightarrow A) \wedge (A \wedge C \rightarrow D) \wedge (A \wedge B \rightarrow 0) \wedge (1 \rightarrow D)$$

Diese Sichtweise ermöglicht die prozedurale Deutung von Logikprogrammen:

Bezeichnung	Merkmal	PROLOG-Notation
Fakten	ein positives Literal, kein negatives Literal	`D.`
Regeln	ein positives Literal, negative Literale	`D :- A, C.`
Ziele	kein positives Literal, negative Literale	`?- A, B.`

Bemerkung:

Nicht jede aussagenlogische Formel besitzt eine äquivalente Horn-Formel. Deshalb stellt die Horn-Logik eine wirkliche Einschränkung dar. Die Erfahrung zeigt aber, dass dies kein Nachteil für die Praxis ist. Der Vorteil der Horn-Logik ist, dass sie einen effizienten Erfüllbarkeitstest in $O(n)$ erlaubt.

Für Horn-Klausel-Mengen kann ein Resolutionsbeweis weiter spezialisiert werden.

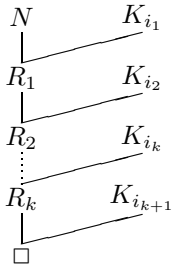
Sei $\mathcal{K}$ eine Horn-Klausel-Menge.
Dann heißt $K \in \mathcal{K}$ **negativ**, falls $K = \{\neg A_1, \ldots, \neg A_k\}$ mit $k \geq 1$, und **definit**, falls $K = \{B, \neg A_1, \ldots, \neg A_k\}$ mit $k \geq 0$.

Definition 2.14 (SLD-Resolution) *Sei $\mathcal{K}$ eine Horn-Klausel-Menge und $N \in \mathcal{K}$ negativ. Dann heißt $\square$ aus N **SLD-resolvierbar**, wenn $\square$ aus N linear resolvierbar ist.*

Beachte:

ein linearer Resolutionsbeweis hat in diesem Fall die folgende Form:

$$\mathcal{K} = \{\underbrace{K_1, \ldots, K_n}_{\text{definit}}, \underbrace{N_1, \ldots, N_m}_{\text{negativ}}\}$$



R_j negativ, $K_{i_j} \in \{K_1, \ldots, K_n\}$

Bemerkung:

SLD steht für "linear resolution with selection function for definite clauses".

Satz 2.7 (Vollständigkeit der SLD-Resolution für Horn-Klausel-Mengen) *Sei $\mathcal{K}$ eine unerfüllbare Horn-Klausel-Menge. Dann existiert eine negative Klausel $N \in \mathcal{K}$, sodass $\square$ aus N SLD-resolvierbar ist.*

Beweis:

Sei $\mathcal{K}_{min}$ eine minimal unerfüllbare Teilmenge von $\mathcal{K}$
$\curvearrowright$ es existiert eine negative Klausel $N \in \mathcal{K}_{min}$.
Nach dem Vollständigkeitsbeweis der linearen Resolution ist $\square$ aus N in $\mathcal{K}$ linear resolvierbar; da $\mathcal{K}$ eine Horn-Klausel-Menge ist, auch SLD-resolvierbar.

q.e.d.

Bemerkung:

SLD-Resolution für Horn-Klausel-Mengen bildet die operationelle Basis für die Logikprogrammierung.

Kapitel 3

Logikprogramme

Definition 3.1 *Eine nichtleere endliche Menge $\mathcal{P}$ von definiten τ-Horn-Klauseln über X $(= VAR)$ heißt* ***Logikprogramm*** *über (τ, X). $K \in \mathcal{P}$ heißt* ***Programmklausel*** *und zwar*

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

Aufruf eines Logikprogramms durch eine ***Zielklausel*** *(Anfrage(-Klausel), goal) der Form $G = \{\neg Q_1, \ldots, \neg Q_k\}$.*

Bedeutung der Unerfüllbarkeit von $\mathcal{P} \cup \{G\}$

Wir ordnen $\mathcal{P}$ und G Formeln zu: für $\mathcal{P} = \{K_1, \ldots, K_n\}$ mit $Var(\mathcal{P}) = \{x_1, \ldots, x_n\}$ sei

- $\varphi(\mathcal{P}) := \forall x_1 \ldots \forall x_n(\varphi(K_1) \wedge \ldots \wedge \varphi(K_n))$
- $\varphi(\{L_1 \ldots L_k\}) := L_1 \vee \ldots \vee L_k$

Dann folgt: $\mathcal{P} \cup \{G\}$ unerfüllbar
$\curvearrowright\!\curvearrowright \varphi(\mathcal{P} \cup \{G\})$ unerfüllbar
$\curvearrowright\!\curvearrowright \varphi(\mathcal{P}) \wedge \forall x_1 \ldots \forall x_n \varphi(G)$ unerfüllbar
$\curvearrowright\!\curvearrowright \varphi(\mathcal{P}) \models \neg\forall x_1 \ldots \forall x_n(\neg Q_1 \vee \ldots \vee \neg Q_k)$
$\curvearrowright\!\curvearrowright \varphi(\mathcal{P}) \models \exists x_1 \ldots \exists x_n(Q_1 \wedge \ldots \wedge Q_k)$

Die Unerfüllbarkeit von $\mathcal{P} \cup \{G\}$ entspricht also der Existenz von Grundtermen $t_1, \ldots, t_n \in T$, sodass $(Q_1 \wedge \ldots \wedge Q_k)[x_1/t_1, \ldots, x_n/t_n]$ aus $\varphi(\mathcal{P})$ folgt.

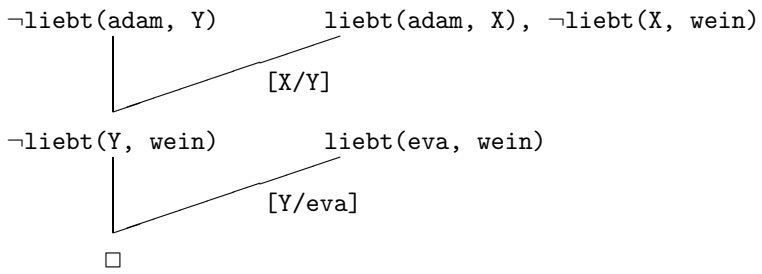
Darüber hinaus gilt:
Ein SLD-Resolutionsbeweis von $\square$ aus $\mathcal{P} \cup \{G\}$ liefert als Seiteneffekt der FO-Resolution eine Variablenbelegung durch Terme, eine s. g. **Antwortsubstitution**. Dabei können noch vorhandene Variablen durch beliebige Grundterme ersetzt werden.

Beispiel:

$\mathcal{P}$ gegeben durch: {`liebt(eva, essen)`}
{`liebt(eva, wein)`}
{`liebt(adam, X)`, ¬`liebt(X, wein)`}

Relationssymbol: `liebt`$^{(2)}$
Konstantensymbole: `eva, adam, essen, wein`
Variable: `X`

Ziel: {`liebt(adam, Y)`}



Anwortsubstitution: `[Y/eva]` $\leadsto$ `liebt(adam, eva)`

Konvention

Variablenumbenennung bei Resolutionsschritten nur in Programmklauseln (**standardisierte SLD-Resolution**)

3.1 Semantik eines Logikprogramms

Definition 3.2 *Sei $\mathcal{P}$ ein Logikprogramm über (τ, X) mit Zielklausel $G = \{\neg A_1, \ldots, \neg A_k\}$, $A_i \in Prim(\tau, X)$ – wobei $Prim(\tau, X)$ die Menge aller atomaren τ-Formeln über X ist. Dann ist die **deklarative Semantik** von $\mathcal{P}$ bezüglich G definiert als:*
$D[\![\mathcal{P}, G]\!] := \{H \mid \mathcal{P} \models H, H \text{ ist Grundinstanz von } A_1 \wedge \ldots \wedge A_k\}$

Bemerkung:

- Jede solche Grundinstanz H enthält als Lösung die entsprechende Grundsubstitution der Variablen aus $A_1 \wedge \ldots \wedge A_k$.
- Die deklarative Semantik ist eine statische, modelltheoretische Semantik.

Ziel: Operationalisierung der deklarativen Semantik

Idee: Berechnung der in $D[\![\mathcal{P}, G]\!]$ definierten Grundinstanzen in zwei Stufen:

(1) Ableitung von Rechenergebnissen mit SLD-Resolution

(2) Grundsubstitution freier Variablen in Rechenergebnissen

Berechnung = Folge von Konfigurationen
Konfiguration = (negative Klausel, $\underbrace{\text{Substitution}}_{\text{Protokoll der Unifikation}}$)

Anfangskonfiguration: (Zielklausel, [ ])
Endkonfiguration: (□, $\underbrace{\text{Endsubstitution}}_{\supseteq \text{Antwortsubstitution}}$)

Definition 3.3 *Sei $\mathcal{P}$ ein Logikprogramm über (τ, X).*

- *Eine **Konfiguration** ist ein Paar (G, sub) mit $G = \{\neg A_1, \ldots, \neg A_k\}$, $A_i \in Prim(\tau, X)$, und $sub : X \longrightarrow T(\tau, X)$.*
- *Ein **Rechenschritt** $(G_1, sub_1) \vdash_{\mathcal{P}} (G_2, sub_2)$ ist definiert falls gilt:*
  - *(1) $G_1 = \{\neg A_1, \ldots, \neg A_k\}$ mit $k \geq 1$*
  - *(2) es existiert $K = \{B, \neg C_1, \ldots, \neg C_n\} \in \mathcal{P}$ mit $n \geq 0$ und $i \in \{1, \ldots, k\}$, sodass*
  - *(3) G_1 und K keine gemeinsamen Variablen haben und A_i und B unifizierbar mit allgemeinstem Unifikator s sind. (Eventuell müssen in K die Variablen umbenannt werden)*
  - *(4) $G_2 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_n, \neg A_{i+1}, \ldots, \neg A_k\}s$*
  - *(5) $sub_2 = sub_1 \circ s$*
- *Eine **Berechnung** von $\mathcal{P}$ bei Eingabe von $G = \{\neg A_1, \ldots, \neg A_k\}$ ist eine (endliche oder unendliche) Folge von Konfigurationen $(G_0, sub_0), (G_1, sub_1), \ldots$ mit*
  - *(1) $G_0 = G$, $sub_0 = [\,]$*
  - *(2) $(G_i, sub_i) \vdash_{\mathcal{P}} (G_{i+1}, sub_{i+1})$*
- *Eine mit $(\square, sub)$ terminierende Berechnung heißt **erfolgreich** mit dem **Rechenergebnis** $(\neg A_1, \ldots, \neg A_k)sub$.*

*Dann ist die **prozedurale Semantik** von $\mathcal{P}$ bezüglich G definiert als:*

$$P[\![\mathcal{P}, G]\!] := \big\{ H \;\big|\; (G, [\,]) \vdash_{\mathcal{P}}^{*} (\square, sub) \text{ und } H \text{ ist Grundinstanz von } (A_1 \wedge \ldots \wedge A_k)sub \big\}$$

Bemerkung:

- Rechenschritte sind standardisierte SLD-Resolutionen mit Buchführung über Substitutionen
- Berechnung ist nicht-deterministisch
  - (1) Auswahl der Programmklausel K (Nicht-Determinismus 1. Art, 3.3.2)
  - (2) Auswahl des negativen Resolutionsliterals $\neg A_i$. (Nicht-Determinismus 2. Art 3.3.1)

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

Beweis:

(1) Korrektheit: $P[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$
zu zeigen: Jede Grundinstanz eines Rechenergebnisses $(A_1 \wedge \ldots \wedge A_k)sub$ von $\mathcal{P}$ mit Eingabe G ist Folgerung von $\mathcal{P}$.

Beweis durch Induktion über die Länge n der Berechnungen:

I.A.: $n = 0 \curvearrowright G = \square$, keine Grundinstanzen $\curvearrowright$ Behauptung

I.S.: Sei $(G, [\,]) \vdash_{\mathcal{P}} (G_1, sub_1) \vdash_{\mathcal{P}} \cdots \vdash_{\mathcal{P}} (\square, sub_1 \circ \cdots \circ sub_{n+1})$ eine Berechnung der Länge $n+1$.
Sei $G = \{\neg A_1, \ldots, \neg A_k\}$ und
$G_1 = \{\neg A_1, \ldots, \neg A_{i-1}, \neg C_1, \ldots, \neg C_l, \neg A_{i+1}, \ldots, \neg A_k\}$ mit
$\{B, \neg C_1, \ldots, \neg C_l\} \in \mathcal{P}$ $(l \geq 0)$ und sub_1 allgemeinster Unifikator von $\{B, A_i\}$
Für die Teilberechnung $(G_1, [\,]) \vdash_{\mathcal{P}} \cdots \vdash_{\mathcal{P}} (\square, sub_2 \circ \cdots \circ sub_{n+1})$ folgt nach I.V., dass eine Grundinstanz von $(A_1 \wedge \ldots \wedge A_{i-1} \wedge C_1 \wedge \ldots \wedge C_l \wedge A_{i+1} \wedge \ldots \wedge A_k) sub_1 sub_2 \ldots sub_{n+1}$ Folgerung von $\mathcal{P}$ ist, also auch jede Grundinstanz von $(C_1 \wedge \ldots \wedge C_l) sub_1 \ldots sub_{n+1}$. Wegen $C_1 \wedge \ldots \wedge C_l \rightarrow B$ (für alle Grundinstanzen) und $B sub_1 \ldots sub_{n+1} = A_i sub_1 \ldots sub_{n+1}$ ist auch jede Grundinstanz von $(A_1 \wedge \ldots \wedge A_k) sub_1 \ldots sub_{n+1}$ eine Folgerung von $\mathcal{P}$.

(2) Vollständigkeit: $D[\![\mathcal{P}, G]\!] \subseteq P[\![\mathcal{P}, G]\!]$
Sei $H \in D[\![\mathcal{P}, G]\!]$. Dann existiert eine Grundsubstitution sub_g mit $H = (A_1 \wedge \ldots \wedge A_k) sub_g$ und $\mathcal{P} \models H$. Also ist $\mathcal{P} \cup \{\{\neg A_1, \ldots, \neg A_k\} sub_g\}$ unerfüllbar. Wegen der Vollständigkeit der SLD-Reolution existiert ein linearer Resolutionsbeweis von $\square$ aus $\{\neg A_1, \ldots, \neg A_k\} sub_g$. O.B.d.A. kann dabei jeder Resolutionsschritt als binär angenommen werden, d. h. nur 2 Literale werden unifiziert.

Somit existiert eine erfolgreiche Berechnung $(\{\neg A_1, \ldots, \neg A_k\} sub_g, [\,]) \vdash^*_{\mathcal{P}} (\square, sub_1 \circ \cdots \circ sub_n)$.

(L1) Diese Berechnung lässt sich "liften" zu einer Berechnung $(\{\neg A_1, \ldots, \neg A_k\}, [\,]) \vdash^*_{\mathcal{P}} (\square, sub'_1 \circ \cdots \circ sub'_n)$ und es gibt eine Grundsubstitution sub'_g, sodass $sub_g = sub'_1 \circ \cdots \circ sub'_n \circ sub'_g$.

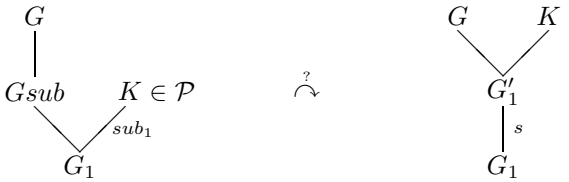
Aus (L1) folgt die Behauptung:
$H = (A_1 \wedge \ldots \wedge A_k) sub_g = \underbrace{(A_1 \wedge \ldots \wedge A_k) sub'_1 \ldots sub'_n}_{\text{Rechenergebnis}} sub'_g$ und somit $H \in P[\![\mathcal{P}, G]\!]$.

Zum Beweis von (L1) zeigen wir allgemeiner:

(L2) Wenn $(G sub, [\,]) \vdash_{\mathcal{P}} (\square, sub_1 \circ \cdots \circ sub_n)$, so existiert eine Berechnung $(G, [\,]) \vdash_{\mathcal{P}} (\square, sub'_1 \circ \cdots \circ sub'_n)$ und eine Substitution s mit $sub \circ sub_1 \circ \cdots \circ sub_n = sub'_1 \circ \cdots \circ sub'_n \circ s$.

Aus (L2) folgt (L1).

Beweis von (L2):



$G_1 = \big((G sub \setminus \{L\} \cup (K \setminus \{L'\}))\big) sub_1$ mit $\overline{L} sub_1 = L' sub_1$. Wähle $L_0 \in G$ mit $L_0 sub = L$. Dann müssen auch $\overline{L_0}$ und L' unifizierbar sein:
$\overline{L_0} sub \circ sub_1 = \overline{L} sub_1 = L' sub_1 = L' sub \circ sub_1$, da sub nur Variablen in G ersetzt. Sei sub'_1 allgemeinster Unifikator von $\overline{L_0}$ und L', so existiert eine

Substitution s mit $sub \circ sub_1 = sub'_1 s$ und es folgt:
$G'_1 := \big((G \setminus \{L_0\}) \cup (K \setminus \{L'\})\big) sub'_1$, sowie

$$\begin{aligned} G'_1 s &= \big((G \setminus \{L_0\}) \cup (K \setminus \{L'\})\big) sub \circ sub'_1 \\ &= \big((Gsub \setminus \{L\}) \cup (K \setminus \{L'\})\big) sub'_1 \\ &= G_1 \end{aligned}$$

q.e.d.

Im Unterschied zur *deklarativen Semantik* wird bei der *Fixpunktsemantik* die Horn-Eigenschaft vorausgesetzt. Dies wird im folgenden erläutert.

Sei $Prim := Prim(\tau, \emptyset)$ die Menge der atomaren τ-Formeln ohne Variablen. Ein Logikprogramm $\mathcal{P}$ über τ und X bestimmt eine Transformation
$trans_{\mathcal{P}} : \mathfrak{P}(Prim) \longrightarrow \mathfrak{P}(Prim)$ mit

$$trans_{\mathcal{P}}(M) := \big\{A' \;\big|\; \text{es existiert } \{A, \neg B_1, \ldots, \neg B_k\} \in \mathcal{P},\ k \geq 0 \text{ und davon Grundinstanz } \{A', \neg B'_1, \ldots, \neg B'_k\}, \text{ sodass } B'_i \in M\big\}$$

$trans_{\mathcal{P}}$ beschreibt die logische Implikation aus M mit $\mathcal{P}$. Dann gilt:

(1) $\langle \mathfrak{P}(Prim); \subseteq \rangle$ ist eine vollständige Halbordnung, d. h. insbesondere: jede Kette $M_0 \subseteq M_1 \subseteq \ldots$ hat eine kleinste obere Schranke: $\bigcup_{i=0}^{\infty} M_i$.

(2) $trans_{\mathcal{P}}$ ist monoton:
$M_1 \subseteq M_2 \curvearrowright trans_{\mathcal{P}}(M_1) \subseteq trans_{\mathcal{P}}(M_2)$

(3) $trans_{\mathcal{P}}$ ist stetig:
für jede Kette $(M_i \;|\; i \in \mathbb{N})$ gilt: $trans_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i) = \bigcup_{i\in\mathbb{N}} trans_{\mathcal{P}}(M_i)$

Beweis:

(3)
- "$\subseteq$" folgt aus Monotonie
- "$\supseteq$": $A' \in trans_{\mathcal{P}}(\bigcup_{i\in\mathbb{N}} M_i)$, dann existiert $\{A, \neg B_1, \ldots, \neg B_k\} \in \mathcal{P}$ mit Grundinstanz $\{A', \neg B'_1, \ldots, \neg B'_k\}$, sodass $B'_1, \ldots, B'_k \in \bigcup_{i\in\mathbb{N}} M_i$. Dann existiert ein $i \in \mathbb{N}$ mit $B'_1, \ldots, B'_k \in M_i$; daraus folgt $A' \in trans_{\mathcal{P}}(M_i) \curvearrowright A' \in \bigcup_{i\in\mathbb{N}} trans_{\mathcal{P}}(M_i)$

q.e.d.

Folgerung:

$trans_{\mathcal{P}}$ besitzt als kleinsten Fixpunkt M, d. h. $M = trans_{\mathcal{P}}(M)$, die Menge: $fix(trans_{\mathcal{P}}) := \bigcup_{n\in\mathbb{N}} trans^n_{\mathcal{P}}(\emptyset)$

Bemerkung:

$trans_{\mathcal{P}}$ enthält die Grundinstanzen der Fakten; daraus werden mit den Regeln sukzessiv weitere "gültige" Grundinstanzen erzeugt.

Definition 3.4 *Sei $\mathcal{P}$ ein Logikprogramm mit Zielklausel $G = \{\neg A_1, \ldots, \neg A_k\}$. Die **Fixpunktsemantik** von $\mathcal{P}$ bezüglich G ist definiert durch:*

$$F_P[\![\mathcal{P}, G]\!] := \big\{H \;\big|\; H \textit{ ist Grundinstanz } A'_1 \wedge \ldots \wedge A'_k \textit{ von } A_1 \wedge \ldots \wedge A_k \textit{ und } A'_i \in fix(trans_{\mathcal{P}})\big\}$$

Satz 3.2 $D[\![\mathcal{P}, G]\!] = P[\![\mathcal{P}, G]\!] = F_P[\![\mathcal{P}, G]\!]$

Beweis:

(1) $P[\![\mathcal{P}, G]\!] \subseteq F_P[\![\mathcal{P}, G]\!]$
analog zum Beweis von $P[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$ durch Induktion über die Länge der erfolgeichen Berechnungen. Induktionsschritt: an die Stelle der Folgerungen von $\mathcal{P}$ tritt die Menge $fix(trans_{\mathcal{P}})$. Beide Mengen sind unter logischer Implikation der Programmklauseln abgeschlossen.

(2) $F_P[\![\mathcal{P}, G]\!] \subseteq D[\![\mathcal{P}, G]\!]$
$G = \{\neg A_1, \ldots, \neg A_k\}$, $H = A'_1 \wedge \ldots \wedge A'_k \in F_P[\![\mathcal{P}, G]\!]$
H ist Grundinstanz von $A_1 \wedge \ldots \wedge A_k$ mit $A'_i \in fix(trans_{\mathcal{P}})$.
zu zeigen: $\mathcal{P} \models H$
Behauptung: $A' \in fix(trans_{\mathcal{P}}) \curvearrowright \mathcal{P} \models A'$

Beweis durch Induktion über $n \in \mathbb{N}$ mit $fix(trans_{\mathcal{P}}) = \bigcup_{n\in\mathbb{N}} trans^n_{\mathcal{P}}(\emptyset)$.

I.A.: $n = 0$: $trans^0_{\mathcal{P}}(\emptyset) = \emptyset$ enthält kein solches A' ✓

I.S.: $A' \in trans^{n+1}_{\mathcal{P}}(\emptyset) = trans_{\mathcal{P}}(trans^n_{\mathcal{P}}(\emptyset))$
Dann existiert $\{A, \neg B_1, \ldots, \neg B_k\} \in \mathcal{P}$ mit
Grundinstanz $\{A', \neg B'_1, \ldots, \neg B'_k\}$ und $B'_j \in trans^n_{\mathcal{P}}(\emptyset)$
$\curvearrowright \mathcal{P} \models B'_j$, also gilt auch $\mathcal{P} \models A'$

Daraus folgt die obige Behauptung, woraus $\mathcal{P} \models H$ folgt.

Aus (1) und (2) folgt mit dem Satz von Clark die Behauptung.

q.e.d.

3.2 Universalität der Logikprogrammierung

Mit Hilfe der Logikprogrammierung lassen sich auch *berechenbare Funktionen* programmieren: jede berechenbare arithmetische Funktion ist logikprogrammierbar. Man sagt: *Die Logikprogrammierung ist universell.*

Definition 3.5 (Berechnung arithmetischen Funktionen) *Festlegungen:*

- *Eine arithmetische Funktion $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ wird dargestellt als Relation durch ihren Graphen $\overline{f}(k_1, \ldots, k_n, k) :\curvearrowright f(k_1, \ldots, k_n) = k$.*
- *$k \in \mathbb{N}$ wird dargestellt durch den Term $\underline{k} := \underbrace{S \ldots S}_{k\text{-}mal} 0$, wobei $S^{(1)}$ ein Operationssymbol und $0^{(0)}$ ein Konstantensymbol ist.*
- *Ein Logikprogramm $\mathcal{P}$ berechnet $f : \mathbb{N}^n \longrightarrow \mathbb{N} :\curvearrowright$ es existiert ein Relationssymbol $\overline{f}^{(n+1)}$, sodass für alle $k, k_1, \ldots, k_n \in \mathbb{N}$ gilt: $f(k_1, \ldots, k_n) = k \curvearrowright \mathcal{P} \models \overline{f}(\underline{k_1}, \ldots, \underline{k_n}, \underline{k})$.*

Definition 3.6 (Kleene) *Die Klasse $RF = \bigcup_{n\in\mathbb{N}} RF^{(n)}$ der **μ-rekursiven Funktionen** ist die kleinste Klasse arithmetischer Funktionen mit:*

(1) Grundfuktionen: $null^{(n)}$, $suc^{(1)}$ und $proj^{(n)}_i$ sind μ-rekursiv, wobei

- $null^{(n)}(k_1, \ldots, k_n) = 0$
- $suc^{(1)}(k) = k + 1$

- $proj_i^{(n)}(k_1, \ldots, k_n) = k_i$

(2) RF ist abgeschlossen unter Komposition, primitiver Rekursion und unbeschränkter Minimalisierung, d. h.

- $f \in RF^{(m)},\ f_1, \ldots, f_m \in RF^{(n)} \curvearrowright comp(f; f_1, \ldots, f_m) \in RF^{(n)}$, *wobei* $comp(f; f_1, \ldots, f_m)(\widetilde{k}) := f\big(f_1(\widetilde{k}), \ldots, f_n(\widetilde{k})\big)$
- $f \in RF^{(n)}, g \in RF^{(n+2)} \curvearrowright prim(f, g) \in RF^{(n+1)}$, *wobei* $prim(f, g) =: h$ *definiert ist durch*
  - $h(\widetilde{k}, 0) = f(\widetilde{k})$
  - $h(\widetilde{k}, k+1) = g\big(\widetilde{k}, k, h(\widetilde{k}, k)\big)$
- $f \in RF^{(n+1)} \curvearrowright min(f) \in RF^{(n)}$, *wobei* $min(f) =: g$ *definiert ist durch*
  - $g(\widetilde{k}) = k :\curvearrowright\curvearrowright f(\widetilde{k}, k) = 0$ *und für alle* $0 \leq k' < k$ *gilt:* $f(\widetilde{k}, k') > 0$.

Satz 3.3 (Universalität der Logikprogrammierung) *Jede μ-rekursive Funktion ist durch ein Logikprogramm berechenbar.*

Beweis:

durch Induktion über den Aufbau von RF

(1) Grundfunktionen
- $null^{(n)}$ wird berechnet durch $\{Null_n(x_1, \ldots, x_n, 0)\}$
- $suc^{(1)}$ wird berechnet durch $\{Suc_1(x, Sx)\}$
- $proj_i^{(n)}$ wird berechnet durch $\{Proj_{n,i}(x_1, \ldots, x_n, x_i)\}$

(2) (a) Sei $h = comp(f; f_1, \ldots, f_m)$, dann wird h als $\overline{h}^{(n+1)}$ berechnet durch $\mathcal{P}_h$, wobei
$\mathcal{P}_h := \mathcal{P}_{f_1} \cup \ldots \cup \mathcal{P}_{f_m} \cup \mathcal{P}_f \cup \{\overline{h}(x_1, \ldots, x_n, z), \neg\overline{f_1}(x_1, \ldots, x_n, y_1),$-
$\ldots, \neg\overline{f_m}(x_1, \ldots, x_n, y_m), \neg\overline{f}(y_1, \ldots, y_m, z)\}$.

(b) Sei $h = prim(f, g)$, dann wird h als $\overline{h}^{(n+2)}$ berechnet durch $\mathcal{P}_h$, wobei
$\mathcal{P}_h := \mathcal{P}_f \cup \mathcal{P}_g \cup \big\{\{\overline{h}(x_1, \ldots, x_n, 0, z), \neg\overline{f}(x_1, \ldots, x_n, z)\}, \{\overline{h}(x_1, \ldots, x_n, Sx, z),$-
$\neg\overline{h}(x_1, \ldots, x_n, x, y), \neg\overline{g}(x_1, \ldots, x_n, x, y, z)\}\big\}$.

(c) Sei $g = min(f)$, dann wird g als $\overline{g}^{(n+1)}$ berechnet durch $\mathcal{P}_g$, wobei
$\mathcal{P}_g := \mathcal{P}_f \cup \big\{\{\overline{g}(x_1, \ldots, x_n, z), \neg\overline{f}(x_1, \ldots, x_n, z, 0), \neg\widehat{f}(x_1, \ldots, x_n, z)\},$-
$\{\widehat{f}(x_1, \ldots, x_n, 0)\}, \{\widehat{f}(x_1, \ldots, x_n, Sx), \neg\widehat{f}(x_1, \ldots, x_n, x), \neg\overline{f}(x_1, \ldots, x_n, x, Sy)\}\big\}$

q.e.d.

Beispiel:

$add(k, 0) = k$
$add(k, k'+1) = add(k, k') + 1$

```
Add(x, 0, x).
Add(x, Sy, Sz) :- Add(x, y, z).
```

oder:
```
Add(x, 0, x).
Add(x, Sy, z) :- Add(x, y, u), Suc(u, z).
Suc(x, Sx).
```

```
?- Add(x, y, 10).
⇝ X = 10, Y = 0;
⇝ X = 9, Y = 1;
...
```

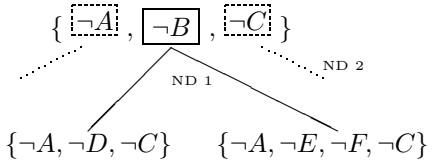
3.3 Nichtdeterminismus und Auswertungsstrategien

Die prozedurale Semantik von Logikprogrammen ist nicht-deterministisch:
Eine Konfiguration (G, sub) kann mehrere Folgekonfigurationen (G, sub_i) mit $(G, sub) \vdash_{\mathcal{P}} (G, sub_i)$ haben $(1 \leq i \leq n)$. Dies hat zwei Ursachen:

- Resolution mit verschiedenen Programmklauseln ist möglich
  (= Nichtdeterminismus 1. Art)
- Resolution mit verschiedenen Literalen von G ist möglich
  (= Nichtdeterminismus 2. Art)

Beispiel:

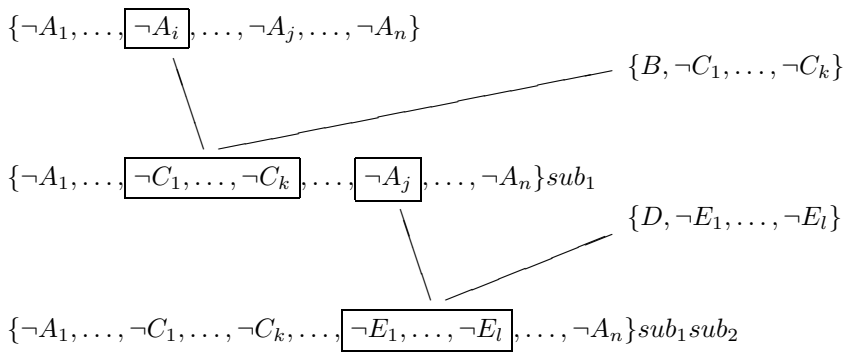
$\mathcal{P} := \{\ldots, \{B, \neg D\}, \{B, \neg E, \neg F\}, \ldots\}$
$G = \{\neg A, \neg B, \neg C\}$



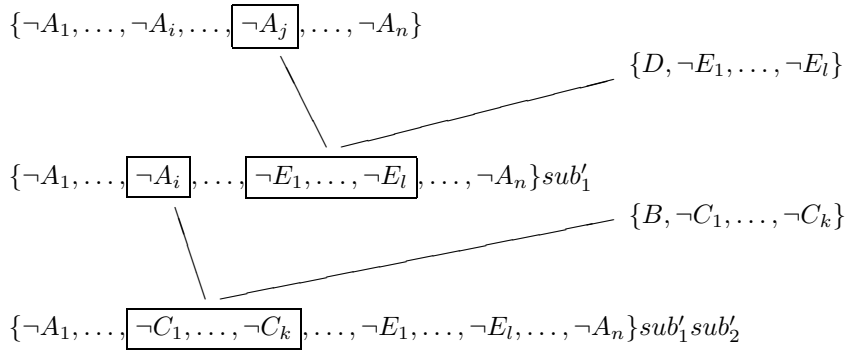
3.3.1 Nichtdeterminismus 2. Art

Es besteht die Möglichkeit verschiedene Literale aus der Zielklausel auszuwählen. Dies ist ein sogenannter *Don't Care*-Nichtdeterminismus, d. h. die Auswahl beeinflusst nicht das Rechenergebnis.

Lemma 3.1 (Vertauschungslemma) *Die Reihenfolge der Zielklauselliterale, nach denen resolviert wird, kann wie folgt vertauscht werden:*
Wenn



dann existieren allgemeinste Unifikatoren sub'_1 und sub'_2, so dass



und es gilt: $sub_1 sub_2 = sub'_1 sub'_2 u$ für eine Variablenumbenennung u.

Beweis:

O.B.d.A.: Resolution mit Programmklauseln jeweils mit "frischen" Variablen, d. h. die Programmklauseln und die Zielklauseln haben keine gemeinsamen Variablen. $(*)$

Für den allgemeinsten Unifikator sub_2 von $A_j sub_1$ und D gilt: $A_j sub_1 sub_2 = D sub_2 = D \underbrace{sub_1 sub_2}_{(*)}$.
Sei sub'_1 allgemeinster Unifikator von A_j und D, so folgt: $sub_1 sub_2 = sub'_1 s$
Weiter folgt: $A_i sub'_1$ und B sind mit s unifizierbar, weil $A_i sub'_1 s = A_i sub_1 sub_2 = B sub_1 sub_2 = B \underbrace{sub'_1}_{(*)} s = Bs$
Wähle sub'_2 als allgemeinsten Unifikator von $A_i sub'_1$ und B.
Bleibt zu zeigen: $sub_1 sub_2 = sub'_1 sub'_2 u$
Dazu wird gezeigt, dass Substitutionen s' und s'' existieren mit

(1) $sub_1 sub_2 = sub'_1 sub'_2 s'$
(2) $sub_1 sub_2 s'' = sub'_1 sub'_2 u$

$\vdots$
(Übung 8.4)

q.e.d.

Folgerung:

Wählt man eine Ordnung der Zielklauselliterale, so kann man stets mit dem ersten beginnen, ohne das Ergebnis zu beeinflussen.

Definition 3.7 *Die Berechnung eines Logikprogramms heißt **kanonisch**, wenn bei jedem Resolutionsschritt mit dem ersten Zielklauselliteral resolviert wird.*

Satz 3.4 (Elimination des Nichtdeterminismus 2. Art) *Sei $\mathcal{P}$ ein Logikprogramm mit der Anfrage G und alle Klauseln seinen geordnet. Dann existiert zu jeder Berechnung $(G, [\,]) \vdash^n_{\mathcal{P}} (\square, sub)$ eine kanonische Berechnung von $(\square, sub)$ mit gleicher Länge.*

Beweis:

Eine gegebene Berechnung $(G, [\,]) \vdash_{\mathcal{P}}^{n} (\square, sub)$ sei bis zum i-ten Schritt kanonisch $(0 \leq i \leq n)$. $(G, [\,]) \vdash_{\mathcal{P}}^{i} (H, sub_i)$ mit $H = \{A_1, \ldots, A_k\}$. Der nächste Schritt sei nicht kanonisch, d. h. Resolutionsliteral ist A_l mit $1 < l \leq k$. Mit A_{i+1} werde erst im j-ten Schritt resolviert, $j > i + 1$. Durch sukzessive Vertauschung der Rechenschritte j und $j - 1$, $j - 1$ und $j - 2$, $\ldots$, $i + 2$ und $i + 1$ kann wegen des Vertauschunglemmas die Resolution mit A_{i+1} schon im $(i + 1)$-ten Schritt erfolgen, ohne das Ergebnis zu verändern.

q.e.d.

Korollar 3.1 *Die Vollständigkeit der SLD-Resolution bleibt mit jeder Auswahlregel erhalten.*

Generalvoraussetzung

Im folgenden sei O.B.d.A jede Berechnung *kanonisch*.

3.3.2 Nichtdeterminismus 1. Art

Es besteht die Möglichkeit verschiedene Programmklauseln auszuwählen.
Die kanonische Berechnung wird dargestellt als *SLD-Baum* – so entspricht die Auswertungsstrategie dem Baumdurchlauf.

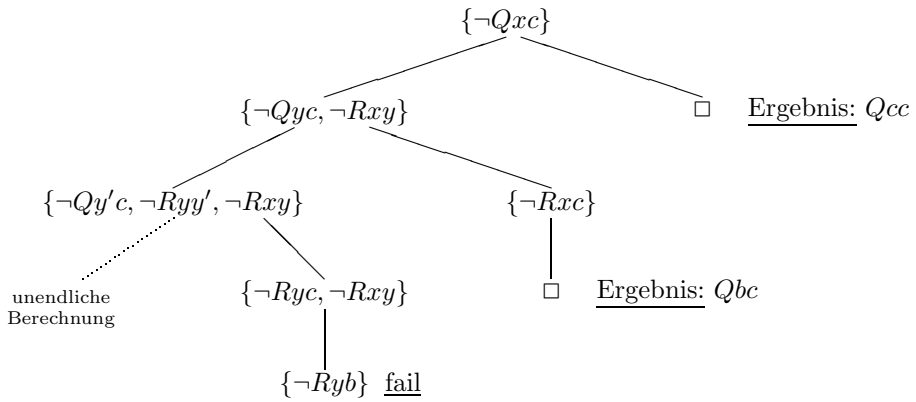
Definition 3.8 *Sei $\mathcal{P}$ ein Logikprogramm mit der Zielklausel G. Der **SLD-Baum** von $\mathcal{P}$ bei Eingabe von G ist ein endlicher oder unendlicher Baum, dessen Knoten mit Konfigurationen markiert sind, sodass gilt:*

(1) Die Wurzel ist mit $(G, [\,])$ markiert.

(2) Ist ein Knoten mit (G', s') markiert, so ist (G'', s'') die Markierung seines Nachfolgerknotens genau dann, wenn $(G', s') \vdash_{\mathcal{P}} (G'', s'')$ ein kanonischer Rechenschritt ist.

Beispiel:

$\mathcal{P} := \{\{Qxz, \neg Qyz, \neg Rxy\}, \{Qxx\}, \{Rbc\}\}$
$G := \{\neg Qxc\}$



Bemerkung:

Im SLD-Baum gibt es

- erfolgreiche Berechnungen mit verschiedenen Ergebnissen
- nicht erfolgreiche Berechnungen
  - ◇ abbrechende
  - ◇ unendliche

Auswertungsstrategie: Lauf durch den SLD-Baum

mögliche Ziele:
- eine erfolgreiche Berechnung
- alle erfolgreichen Berechnungen

Strategietypen:

- **Breitensuche** (*breadth-first-search, bfs*)
  ↪ vollständig, aber teuer
- **Tiefensuche** (*depth-first-search, dfs*)
  ↪ unvollständig, aber effizient

3.4 Zusammenfassung (von $\models$ zum SLD-Baum)

- FO: $\{\varphi_1, \ldots, \varphi_k\} \models \psi \curvearrowleft\!\!\curvearrowright \varphi_1 \wedge \ldots \wedge \varphi_k \wedge \neg\psi$ unerfüllbar
- Normalform: $\varphi = \forall x_1 \ldots \forall x_n \varphi' \in SNF$
- Reduktion: Herbrand-Modelle reichen aus (*eine* Trägermenge $T(\tau, \emptyset)$)
- Herbrand-Expansion: $E(\varphi) = \{\varphi'[x_i/t_i] \mid t_i \in T\}$ (aufzählbar)
- Satz: φ unerfüllbar $\curvearrowleft\!\!\curvearrowright$ $E(\varphi)$ unerfüllbar
  $\curvearrowleft\!\!\curvearrowright \exists k \in \mathbb{N} : \varphi_1 \wedge \ldots \wedge \varphi_k$ unerfüllbar (semientscheidbar)
- AL-Resolution: φ' in KNF $\rightsquigarrow$ Klauselmenge $\mathcal{K}(\varphi')$
- Satz: φ unerfüllbar $\curvearrowleft\!\!\curvearrowright \exists$ Klauseln $k_1, \ldots, k_r \in \mathcal{K}(\varphi')$ und Grundsubstitutionen $s_1, \ldots, s_r$: $\{k_1 s_1, \ldots, k_r s_r\}$ unerfüllbar
- AL-Resolution *liften* zur FO-Resolution mit Unifikation
- Resolutionssatz der FO: $\varphi = \forall x_1 \ldots \forall x_n \varphi' \in SNF$, φ' in KNF: φ unerfüllbar $\curvearrowleft\!\!\curvearrowright \square \in Res^*\big(\mathcal{K}(\varphi')\big)$
- Vereinfachung: lineare Resolution
- Horn-Logik: SLD-Resolution
- weitere Vereinfachungen:
  - ◇ binäre Resolution
  - ◇ standardisierte Resolution
  - ◇ kanonische Berechnung
  - ◇ SLD-Baum

Kapitel 4

PROLOG

PROLOG= *Programming in Logik*
Ausbau zu einer (vollständigen) Programmiersprache

Historische Entwicklung

1965	Resolutionsprinzip von ROBINSON, Unifikation
1972–75	Forschergruppe in Marseille (Colmeraner): effiziente Implementierung der Resolution $\rightsquigarrow$ PROLOG
1974	semantische Grundlagen (KOWALSKI)
1977	DEC-10-PROLOG (PREIRA/WARREN [WAM])
1981	Wahl von PROLOG als Sprache für Japans "5^{th} *Generation Computer Project*" $\rightsquigarrow$ weltweite Akzeptanz von PROLOG als KI-Sprache

Anwendungsbereiche

- KI, Expertensystem
- deduktive Datenbanken
- Verarbeitung natürlicher Sprachen
- symbolische Mathematik
- *Prototyping* (z. B. *Scanner, Parser*)

Vorteile

+ verbreiteter Sprachstandard mit vielen Implementierungen
+ klares Grundkonzept mit einfacher Syntax und Semantik
+ problemorientiert, einfache Programmierung

Nachteile

– Programmausführung manchmal ineffizient (SLD-Baum durchsuchen)
– noch kaum Unterstützung für den Entwurf großer Systeme

4.1 Syntax und Semantik

Ein PROLOG-**Programm** P ist eine Folge von geordneten definiten Horn-Klauseln: $P = (K_1, \ldots, K_r)$. Dabei heißt

- $K_i = \{B\}$ ein **Fakt** (Tatsachenklausel)
  Bezeichnung: $K_i =$ `B.`
- $K_i = \{B, \neg C_1, \ldots, \neg C_n\}$ eine **Regel** (Prozedurklausel)
  Bezeichnung: $K_i =$ `B :- C1,...,Cn.`
- $G = \{\neg A_1, \ldots, \neg A_k\}$ eine **Anfrage** (Zielklausel)
  Bezeichnung: $G =$ `?- A1,...,Ak.`

Hierbei sind `Ai`, `B`, `Ci` Primformeln. Die Signatur ergibt sich aus den auftretenden Funktions- und Prädikatsbezeichnern (Prädikat = Relation):

- Funktions- und Prädikatsbezeichner beginnen mit einem Kleinbuschstaben.
- Variablenbezeichner beginnen mit einem Großbuchstaben oder dem Unterstrich "_".

Beispiel:

```
p(X, Z) :- k(X, Y), p(Y, Z).
p(X, X).
k(a, b).
?- p(U, b).
```

Auswertung

- SLD-Resolution + kanonische Berechnung (SLD-Baum)
- "*depth-first/left-to-right*"-Strategie für den Lauf durch den SLD-Baum

Beachte:

Diese Strategie ist unvollständig, da die Klauselreihenfolge dabei eine Rolle spielt. Daher muss man die Reihenfolge der Klauseln so wählen, dass unendliche Berechnungen vermieden werden (Fakten zuerst oder *Cut*). Dies stellt einen Widerspruch zum Prinzip der deklarativen Programmierung dar, denn der Programmierer sollte das Problem, aber nicht dessen Lösungsstrategie beschreiben. Das bedeutet, dass die Trennung von Logik und Kontrolle bei PROLOG nicht vollkommen erreicht wurde.

4.2 Auswertungsstrategie

Eingabe: PROLOG-Programm $P = \{K_1, \ldots, K_r\}$ mit
$K_i =$ `Bi :- Ci1,...,Cini.` ($\mathtt{ni} \geq 0$) und
$G =$ `?- A1,...,Ak.`

```
main(){
  boolean success = false;
  eval(G, [ ]);
  if(success) println("ja");
  else println("nein");
}

void eval(Berechnungsklausel G, Substitution sub){
  if(G == □){        //Berechnungsende
    println("Result: " + (A_1 ∧...∧ A_k)sub);
(!)   success = true;
  } else {        //G =? − D_1, ..., D_l.
(!)   for(int i = 1; i <= r && !success; i++){
        if(D_1 und B_i unifizierbar mit mgu s)
          eval(?- (C_{i_1},...,C_{i_{n_i}},D_2,...,D_l)s, sub ∘ s);
      }
  }
}
```

Beachte:

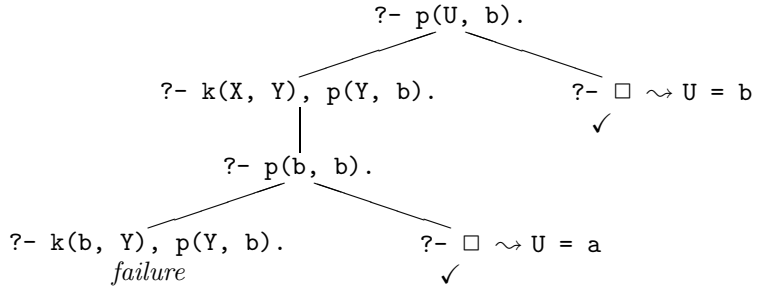
- Abbruch bei 1. Ergebnis wegen (!)
- Backtracking bei erfolgloser Unifikation am Ende der `for`-Schleife

Beispiel (Fortsetzung):

```
?- p(U, b).            [ ]
?- k(X, Y), p(Y, b).   [U/X, Z/b]
?- p(b, b).            [X/a, Y/b]
?- k(b, Y), p(Y, b).   [X/b, Z/b]
failure ↷ Backtracking
?- □                   [X/b]
```

$\rightsquigarrow$ Result: `p(U, b)[U/a] = p(a, b)`

SLD-Baum:



Bemerkung:

- *Backtracking:* Aufsuchen der letzten Stelle mit Alternativen für die Auswahl der Programmklauseln; auch erforderlich bei der Bestimmung weiterer Lösungen
- Bei unendlicher Berechnung können Lösungen verborgen bleiben
- keine Ausgabe von Grundinstanzen: Variablen werden am Schluss nicht instanziiert

4.3 Arithmetik

Ein primitiver Ansatz ist die natürlich Zahlen als Terme darzustellen:

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

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

Prozedurale Sicht eines Logikprogramms

$\underbrace{\texttt{B}}_{\text{Kopf}}$ `:-` $\underbrace{\texttt{A}_1\texttt{,...,A}_n}_{\text{Rumpf}}$ `.` $\qquad$ $\underbrace{\texttt{?- A.}}_{\text{Anfrage}}$

Parameterübergabe durch Unifikation.

In obigem Beispiel: `X = null`, `Y = null`, `U = succ(Z)`, `Z = null`
$\rightsquigarrow$ `U = succ(null)`

Beachte:

Funktionen als Relationen
$\curvearrowright\!\!\curvearrowright$ Funktionsargumente und -werte "gleichberechtigt"

Der Nachteil der Termdarstellung der natürlichen Zahlen ist die mangelnde Effizienz wegen fehlender Grundfunktionen. Daher ist es besser, die schnelle Rechnerarithmetik auszunutzen, indem man vordefinierte Zahlentypen (*int, float*) mit einigen Grundfunktionen und Prädikaten verwendet.

Beachte:

Zwei Auffassungen von Termen:

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

Arithmetische Ausdrücke

sind induktiv aufgebaut aus Zahlen, Variablen und binären Infixoperatoren:

`*`	Multiplikation	`**`	Potenz
`/`	Division	`//`	ganzzahlige Division
`+`	Addition	`mod`	ganzer Rest
`-`	Subtraktion	...	*einige andere*

Prädikate

(1) Vergleich: t_1 *op* t_2, wobei *op* $\in \{<, >, =<, >=, =:=, =/=, \ldots\}$
Erfolg, falls t_i auswertbar zu z_i und z_1 *op* z_2 gültig.
op erzwingt eine Auswertung!

(2) Wertzuweisung: t_1 `is` t_2
Erfolg, falls t_2 auswertbar zu z_2 und t_1 mit z_2 unifzierbar.

Unterscheide drei Arten von "Gleichheit":

Wertgleicheit	t_1 `=:=` t_2	Auswertung von t_1 und t_2 und Unifikation
Wertzuweisung	t_1 `is` t_2	Auswertung von t_2 und Unifikation
Termgleicheit	t_1 `=` t_2	keine Auswertung nur Unifikation

← vordefinierte Prädikate in SWI Prolog

Beispiel:

```
?- X = 3 + 4, Y is X.
⇝ X = 3 + 4, Y = 7.
?- 1 + 2 = 2 + 1.
⇝ No.
?- X = f(X).
⇝ X = f(f(f(f(f(f(f(f(f(...)))))))))).      % ohne occur check
?- unify_with_occurs_check(X, f(X)).
⇝ No.
?- 1 + X = Y + 1.
⇝ X = 1, Y = 1;
⇝ No.
?- 1 + X =:= Y + 1.
⇝ ERROR: Arguments are not sufficiently instantiated!
?- 1 + 6 =:= 6 + 1.
⇝ Yes.
?- 6 =:= 7 - 1.
⇝ Yes.
```

Verwendung

- ```
  adds(X, Y, Z) :- Z is X + Y.
  ?- adds(2, 2, Z). ⇝ Z = 4.
  ?- adds(X, Y, 4). ⇝ ERROR: Arguments ...!
  ```
- ```
  fact(0, 1).
  fact(N, F) :- N > 0, N1 is N-1, fact(N1, F1), F is N*F1.
  ```
- ```
  ggT(X, X, X).
  ggT(X, Y, Z) :- X < Y, Y1 is Y - X, ggT(X, Y1, Z).
  ggT(X, Y, Z) :- ggT(Y, X, Z).
  ```
  ```

4.4 Listen

Definition 4.1 *Sei A eine Menge. Eine **Liste** über A ist eine lineare Liste (= Wort) über A oder Listen über A: $L(A) = (A \cup L(A))^*$.*

Beispiel:

Notation: $[a, b, c]$, $[\,]$, $[a]$, $[[a, b], a, [\,]]$

Beachte:

$[a] \neq a$!

Termdarstellung von Listen

Seien $cons \in F^{(2)}$ und $nil \in F^{(0)}$ Funktionssymbole. Ferner sei $A \subseteq F^{(0)}$. Wir definieren $term : L(A) \longrightarrow T(\tau, \emptyset)$ durch:

- $term\big([\,]\big) := nil$
- $$term\big([E_0, E_1, \ldots, E_n]\big) := \begin{cases} cons\Big(E_0, term\big([E_1, \ldots, E_n]\big)\Big) & : \quad E_0 \in A \\ cons\Big(term(E_0), term\big([E_1, \ldots, E_n]\big)\Big) & : \quad \text{sonst} \end{cases}$$

Beispiel:

- $[\,] \mapsto nil$
- $[a] \mapsto cons(a, nil)$
- $[a, b, c] \mapsto cons(a, cons(b, cons(c, nil)))$
- $\big[[a, b], a, [\,]\big] \mapsto cons(cons(a, cons(b, nil)), cons(a, cons(nil, nil)))$

PROLOG**-Notation**

Sowohl $\big[a, b, c\big]$ als auch $[head \,\big|\, tail]$
z. B. `[a | [b, c]] = [a, b, c]`
oder: `[ a, b, c | [b, c]] = [a, b, c, b, c]`

(1) `member(X, Xs)` : "`X` ist Element der Liste `Xs`."

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

Anonyme Variablen: `_Xs`, `_Y` $\curvearrowright$ Wertbindung unwichtig

```
?- member(X, [[a, b], a, [ ]]).
⇝ X = [a, b];
⇝ X = a;
⇝ X = [ ];
⇝ No.
?- member(b, X).
⇝ X = [b | _G261];
⇝ X = [_G260, b | _G264];
   ⋮
```

(2) `append(Xs, Ys, Zs)` : "Zusammenfügen von `Xs` und `Ys` ergibt `Zs`."

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

?- append([a, b], [c, d], [a, b, c, d]).
⇝ Yes.
?- append(Xs, Ys, [a, b, c]).
⇝ Xs = [ ], Y = [a, b, c];
⇝ Xs = [a], Y = [b, c];
⇝ Xs = [a, b], Y = [c];
⇝ Xs = [a, b, c], Y = [ ];
⇝ No.
```

(3) `length(Xs, N)` : "Die Länge der Liste `Xs` ist `N`."

```
length([ ], 0).
length([X, Xs], N) :- length(Xs, N1), N is N1 + 1.

?- length([[a, b], a, [ ]], N).
⇝ N = 3.
```

4.5 Operatoren

Die PROLOG-Standardnotation für Terme lautet: `f(X, g(a))` mit Funktoren `f/2` und `g/1`. Funktoren sind Funktionssymbole in Präfixnotation.

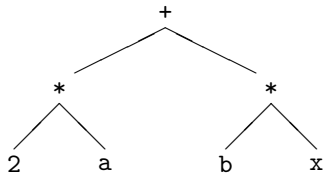
Operatoren: Funktionssymbole der Stelligkeit 1 oder 2 in Präfix-, Infix- oder Postfixnotation mit Präzedenzangaben zur Vermeidung von Klammern.

Ziel:

- benutzerfreundliche Syntax
- bessere Lesbarkeit
- "Programmieren in natürlicher Sprache"

Beispiel (arithmetische Ausdrücke):

PROLOG erlaubt anstelle von `+(*(2, a), *(b, x))` auch die übliche Notation `2*a + b*x`. Die interne Darstellung sieht in beiden Fällen wie folgt aus:



Eindeutige Termzerlegung aufgrund vordefinierter Direktiven:
Regel mit leerem Kopf, Systemprädikat `op/3`:
`:- op(`*Präzedenz, Typ, Name*`)`
Präzedenz : $0 \leq$ ganze Zahl ≤ 1200
Typ : `fx`, `fy`, `xf`, `yf`, `xfx`, `xfy`, `yfx`, `yfy`

Der Typ bestimmt die Reihenfolge von Operatoren und Argumenten:

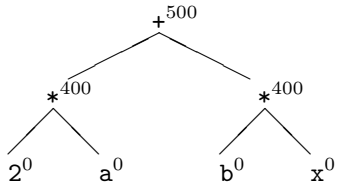
- Präzedenz des x-Arguments < Präzedenz von f
- Präzedenz des y-Arguments ≤ Präzedenz von f
- Konstanten und Variablen haben Präzedenz 0
- Präzedenz eines g-Terms hat Präzedenz von g

Vordefinierte Direktiven für das obige Beispiel:

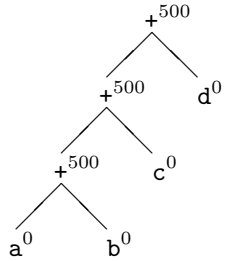
- `:- op(500, yfx, +).`
- `:- op(400, yfx, *).`

D. h. `*` bindet stärker als `+`, beide sind linksassoziativ.

(1) `2*a + b*x`:



(2) `a + b + c + d`:



Benutzerdefinierte Operatoren:

- `:- op(300, xfx, was).`
- `:- op(250, xfy, of).`
- `:- op(200, fy, the).`

Term: `laura was the secratary of the head of the department.`
→ `was(laura, of(the(secratary), of(the head), the(department))).`

```
?- Who was the secratary of the head of the department.
⇝ Who = laura.
?- laura was What.
⇝ What = the secratary of the head of the department.
```

4.6 Ein- und Ausgabe

bisher:

Eingabe nur in Form einer Anfrage

- ohne Variablen $\rightsquigarrow$ `Yes` oder `No`
- mit Variablen $\rightsquigarrow$ Variablenbindung

zusätzlich:

- extralogische Prädikate für *I/O* mit *files*
- Erfüllung mit Seiteneffekt
- Problem: *Backtracking*

Zu jedem Ausführungszeitpunkt existieren zwei aktive *files*:

(1) *current inputstream*

(2) *current outputstream*

Gewöhnlich aktuell sind die *ueserfiles*.

Wechsel des Eingabe*files*: `see(`*file*`)` $\rightarrow$ Erfüllung bei Existenz des *files*
Wechsel des Ausgabe*files*: `tell(`*file*`)`
zurück mit `see(user)` bzw. `tell(user)`

4.6.1 Ausgabe

`write(t)` ist erfolgreich und schreibt als Seiteneffekt den Term `t` (in Standard-Ausgabe-Syntax) auf den aktuellen Ausgabe*file*.

```
?- X is 2, write(X). ⇝ X = 2.
?- write(+(3, 2)). ⇝ 3 + 2.
?- write(f(Y)). ⇝ f(_G240).   % Y nicht instanziiert
?- write('Das ist ein Atom'). ⇝ Das ist ein Atom.
```

Anwendung

Funktionen mit Ergebnis als Ausgabe, z. B.:

```
mult(X, Y) :- Ergebnis is X*Y, write(X*Y), write(' = '),
              write(Ergebnis).

?- mult(3*4).
⇝ 3*4 = 12.
```

Vorsicht beim Backtracking

Ein Seiteneffekt kann nicht rückgängig gemacht werden, z. B.:

```
q(a).
q(b).
p :- q(X), write(X), X = b.

?- p.
⇝ ab.
```

4.6.2 Eingabe

`read(t)` liest einen Term `s` von der Eingabe und unifiziert `s` mit `t`. Ist keine Unifikation möglich, so ergibt sich ein Fehlschlag ohne Eingabealternative.
SWI PROLOG: Fehlschlag erzeugt *Exception*.

Beispiel (aus der Einleitung):

```
istVaterVon(V, K) :- verheiratet(M, V),
                     istMutterVon(M, K).
```

Aufgabe: Definition eines Prädikats `vater` ohne Parameter

(1) Ausgabe: "`Von welcher Person möchten Sie den Vater wissen?`"

(2) Eingabe: "`ana`"

(3) Ausgabe: "`Der Vater von ana ist hans.`"

Lösung:

```
vater :- write('Von welcher Person ...'), nl, read(Person),
         istVaterVon(V, Person), write('Der Vater von '),
         write(Person), write(' ist '), write(V).
```

4.7 Das *Cut*-Prädikat

Nachteile der *dfs*-Startegie von PROLOG:

- Unvollständigkeit bei unendlichen Zweigen im SLD-Baum
- Aufwendiges Backtracking bei einem Fehlschlag: Alternativen vor einem Fehlschlag müssen beim Beweis gespeichert werden.

Abhilfe durch das Kontroll-Prädikat *Cut* – Bezeichnung: `!`
Der *Cut* schneidet Teile des SLD-Baums ab.

4.7.1 Syntax

Der *Cut* kann auf der rechten Regelseite oder in Anfragen als Primformel (Literal) auftreten:

(1) $\texttt{A :- B}_1\texttt{, ..., B}_i\texttt{, !, B}_{i+1}\texttt{, ..., B}_k\texttt{,}$

(2) $\texttt{?- B}_1\texttt{, ..., B}_i\texttt{, !, B}_{i+1}\texttt{, ..., B}_k\texttt{,}$

4.7.2 Semantik

logisch: `!` ist wie `true` immer beweisbar.

prozedural:

(1) $\texttt{B}_1\texttt{, ..., B}_i$ nicht beweisbar:

  (a) nächste `A`-Regel wählen oder *Backtracking*

  (b) Anfrage nicht beweisbar

(2) `B1, ..., Bi` beweisbar:
Zum Beweis von `A` bzw. der Anfrage müssen dann auch `Bi+1, ..., Bk` bewiesen werden.

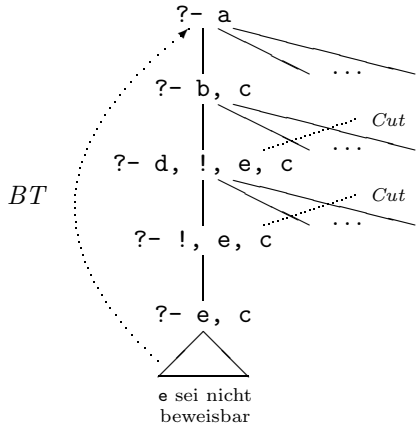
Folgerung:

Sind im Fall (2) `Bi+1, ..., Bk` nicht beweisbar, so werden beim *Backtracking* die Alternativen von `B1, ..., Bi` und `A` übersprungen. Im SLD-Baum wird somit ein Teil abgeschnitten.

Beispiel:

```
a :- b, c.
b :- d, !, e.
d.
⋮
?- a.
```

SLD-Baum:



Anwendungen

(1) Vermeidung unnötiger Berechnungen
Cut als Optimierung eines PROLOG-Programms

(2) Simulation von *if-then-else* (Determinismus)
Die Regel `A :- if B then C else D.` habe die Semantik:

- `A` erfüllbar, wenn `B` und `C` erfüllbar oder `B` nicht erfüllbar und `D` erfüllbar
- weitere Beweise von `A` sind danach ausgeschlossen

```
A :- B, !, C.
A :- !, D.
```

In SWI PROLOG: `B -> C; D.`

(3) Implementierung der Negation

bisher: nur positive Aussagen ableitbar.
Falls $Cons(\mathcal{P})$ vollständig ist, d. h. für eine Formel A gilt entweder $A \in Cons(\mathcal{P})$ oder $A \notin Cons(\mathcal{P})$, so folgt:
$\neg A \in Cons(\mathcal{P}) \quad \curvearrowright\!\!\curvearrowright \quad A \notin Cons(\mathcal{P})$
$\curvearrowright\!\!\curvearrowright \quad \square \notin Res^*(\mathcal{P} \cup \{\neg A\})$

Probleme:

- $Cons(\mathcal{P})$ muss nicht vollständig sein
- Es ist im Allgemeinen – wegen unendlicher Berechnungen – nicht entscheidbar, ob $\square \notin Res^*(\mathcal{P})$

Hilfslösung:

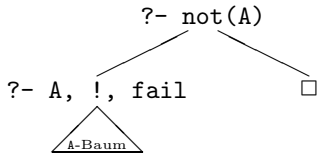
Negation als *finite failure*:
$\neg A$ bezüglich $\mathcal{P}$ erfolgreich :$\curvearrowright\!\!\curvearrowright$ SLD-Baum zu $\neg A$ bezüglich $\mathcal{P}$ ist endlich und hat keinen erfolgreichen Ast.

Simulation durch den *Cut*

```
not(A) :- A, !, fail.
not(A).
```

`fail` ist ein Standardprädikat, das stets fehlschlägt.

SLD-Baum:



(a) nur endlich viele Fehlschläge $\rightsquigarrow$ *Backtracking* $\rightsquigarrow \square$
(b) `A` beweisbar $\rightsquigarrow$ Fehlschlag
(c) unendliche Berechnung $\rightsquigarrow$ kein Ergebnis

In SWI Prolog: `\+`

Gefahren beim *Cut*

```
anzahlEltern(adam, 0) :- !.
anzahlEltern(eva, 0) :- !.
anzahlEltern(X, 2).

?- anzahlEltern(adam, N).
⇝ N = 0;
⇝ No.
?- anzahlEltern(john, N).
⇝ N = 2;
⇝ No.
?- anzahlEltern(adam, 2).
⇝ Yes.
```

```
?- anzahlEltern(X, 0).
⇝ X = adam;
⇝ No.
```

Kapitel 5

Programmiertechniken

5.1 Nichtdeterministische Programmierung

Der Nichtdeterminismus erlaubt oft einfache Algorithmen – Beispiele sind *Scanner* oder *Parser*. In PROLOG wird der Nichtdeterminismus durch sequezielle Auswertung und *Backtracking* simuliert.

Eine Programmiertechnik stellt "*generate and test*" dar, bei dem Lösungsvorchläge erzeugt und auf Brauchbarkeit überprüft werden:

```
find(X) :- genrate(X), test(X).
```

Hierbei führt ein erfolgloser Test zu *Backtracking*, also zu einem nächsten Lösungsvorschlag, der durch `generate(X)` generiert wird. Dieser Ansatz ist einfacher als eine direkte Lösung, hat aber den Nachteil, dass er im Allgemeinen weniger effizient ist.

Beispiel (Wortsuche):

```
verb(Sentence, Word) :- member(Word, Sentence), verb(Word).

?- verb([a, man, loves, a, women], V).
⇝ V = loves.
```

Beispiel (PermutationSort):

```
sort(Xs, Ys) :- permutation(Xs, Ys), ordered(Ys).

permutation([ ], [ ]).
permutation(Xs, [Z | Zs]) :- select(Z, Xs, Ys), permutation(Ys, Zs).

select(X, [X | Xs], Xs).
select(X, [Y | Ys], [Y | Zs]) :- select(X, Ys, Zs).

ordered([X]).
ordered([X, Y | Ys]) :- X =< Y, ordered([Y | Ys]).
```

`sort` hat exponenzielle Laufzeit, ist also ineffizient. Besser ist es, das Testen in das Erzeugen zu integrieren, d. h. `permutation` und `ordered` zu einem Prädikat kombinieren (→ *Insertion Sort*).

```
sort2([ ], [ ]).
sort2([X | Xs], Ys) :- sort2(Xs, Zs), insert(X, Zs, Ys).

insert(X, [ ], [X]).
insert(X, [Y | Ys], [Y | Zs]) :- X > Y, insert (X, Ys, Zs).
insert(X, [Y | Ys], [X, Y | Ys]) :- X =< Y.
```

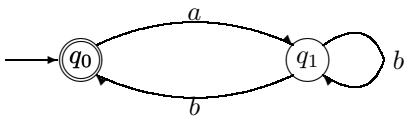
Die Laufzeit von `sort2` ist $O(n^2)$.

5.1.1 Nichtdeterministische Simulation

$\mathfrak{A} = \langle Q, \Sigma, \delta, q_0, F\rangle \in NFA(\Sigma)$ mit $\delta : Q \times \Sigma \longrightarrow \mathfrak{P}(Q)$

Beispiel:

$\mathfrak{A}_1$:



$L(\mathfrak{A}) = (ab^*b)^*$

Beschreibung von $\mathfrak{A}$ durch Fakten und Regeln:

```
initial(q0).
final(q0).
delta(q0, a, q1).
delta(q1, b, q1).
delta(q1, b, q0).
accept(W) :- initial(Q), accept(Q, W).
accept(Q, [ ]) :- final(Q).
accept(Q, [X | Xs]) :- delta(Q, X, Q1), accept(Q1, Xs).
```

Das Erkennen eines Wortes (Beispiel: `abb`) erfordert – aufgrund des Nichtdeterminismus – *Backtracking*.

5.2 Akkumulatortechnik

Beispiel (Spiegelung einer Liste):

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

Zeitaufwand: $O(n^2)$ – wobei n die Länge der Liste ist – da hier eine 2-geschachtelte Rekursion vorliegt; `append` hat Zeitaufwand $O(n)$.

Eine Verbesserung kann erreicht werden, indem man einen *Akkumulator* einfügt:

```
reverse(Xs, Ys) :- reverse2(Xs,    [ ]    , Ys).
                                  Akkumulator
reverse2([ ], Ys, Ys).
```

```
reverse2([X | Xs], Akku, Ys) :- reverse2(Xs, [X | Akku], Ys).
```

Zeitaufwand: $O(n)$.

Beispiel (Fakultät):

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

Platzbedarf: $O(\texttt{N})$ – Zwischenspeichern im Berechnungszustand.

iterativ:
```
fact(0, 1).
fact(N, F) :- fact(  0  , N,  1  , F).
                   Schleifenvariable  Teilergebnis
fact(N, N, F, F).
fact(I, N, T, F) :- I < N, I1 is I + 1, T1 is T*I1
                    fact(I1, N, T1, F).
```

5.3 Differenzlisten

Differenzlisten sind eine alternative Darstellung von Listen, haben aber eine unvollständige Datenstruktur. Sie stellen eine Art Verallgemeinerung der Akkumulatortechnik dar.

Idee:

Darstellung einer Liste als Differenz zweier Listen

Beispiel:

```
[1, 2, 3] = [1, 2, 3, 4, 5]-[4, 5]
[1, 2, 3] = [1, 2, 3 | Xs]-Xs
```

In Prolog: `\`, in SWI Prolog: `-` als binären Funktor in Infixnotation.

Mit Hilfe von Differenzlisten kann man das Konkatenieren zweier Listen in $O(1)$-Zeit statt in $O(n)$-Zeit schaffen:

```
append_dl(Xs-Ys, Ys, Xs).

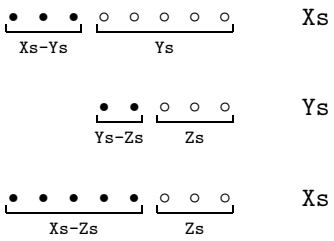
?- append_dl([1, 2, 3 | Ys]-Ys, [4, 5], X).
⇝ Ys = [4, 5], X = [1, 2, 3, 4, 5].
```

Beachte:

Hierbei wird lediglich ein Unifikationsschritt benötigt. Der Nachteil ist allerdings, das die Ergebnisliste `Xs` selbst keine Differenzliste ist, d. h. `append_dl` ist auf `Xs` nicht mehr anwendbar. Eine Verbesserung wäre also gegeben durch:

```
append_dl(Xs-Ys, Ys-Zs, Xs-Zs).
```

Vorstellung der Konkatenation von Differenzlisten




```
?- append_dl([1, 2, 3 | Ys]-Ys, [4, 5 | Zs]-Zs, X).
⇝ Ys = [4, 5 | _G406],
   Zs = _G406,
   X = [1, 2, 3, 4, 5 | _G406]-_G406.
```

Beispiel (Umwandeln geschachtelter in lineare Listen):

(1) übliche Definition mit geschachtelter Rekursion

```
flatten([ ], [ ]).
flatten(X, [X]) :- constant(X), X =/= [ ].
flatten([X | Xs], Ys) :- flatten(X, Ys1), flatten(Xs, Ys2),
                         append(Ys1, Ys2, Ys).
```

(2) mit Differenzlisten

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

Optimierung:

Teilbeweis von `append_dl` ausführen: `As = Ys`, `Bs = Cs`, `Ds = Zs`.
Aus der letzten Zeile wird dann:

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

5.4 Definite Klausel-Grammatiken (DCG's)

Eine wichtige Anwendung der Logikprogrammierung ist die Syntaxanalyse (*Parsing*). Dazu benötigt man DCG's (*definite clause grammars*), die eine Verallgemeinerung kontextfreier Grammatiken (CFG's) sind. Dabei werden sowohl Differenzlisten, als auch Nichtdeterministische Programmierung verwendet.

5.4.1 Übersetzung von CFG's in PROLOG-Programme

$G = \langle N, \Sigma, S, P\rangle \in CFG(\Sigma)$ mit

- N: Nichtterminalsymbole
- Σ: (Eingabe-)Alphabeth
- S: Startsymbol
- P: Produktionsregel, wobei $A \to \alpha \in P$, $A \in N, \alpha \in (\Sigma \cup N)^*$.

Die Ableitungsrelation $\Rightarrow$ist definiert durch:
$\beta \Rightarrow_G \gamma :\curvearrowright\curvearrowright \exists A \to \alpha \in P : \beta = \beta_1 A \beta_2,\ \gamma = \beta_1 \alpha \beta_2$

Die Sprache von G ist defniert durch $L(G) = \{w \in \Sigma \mid S \Rightarrow_G^* w\}$

PROLOG-**Notation** für Elemente aus

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

Übersetzung in PROLOG-Klauseln

(1) Ordne jedem Nichtterminalsymbol ein einstelliges Prädikat zu, welches testet, ob sein Argument aus diesem Symbol ableitbar ist.

(2) Übersetze jede Regel der Grammatik nach folgendem Schema in eine PROLOG-Klausel:

`a --> w.` $\mapsto$ `a(w).`
`a --> a1, a2.` $\mapsto$ `a(A) :- a1(A1), a2(A2), append(A1, A2, A).`

Beispiel:

```
satz(S, S0) :- nominalphrase(S, VP), verbalphrase(VP, S0).
nominalphrase(NP, D) :- artikel(NP, N), nomen(N, D).
verbalphrase(VP, D) :- verb(VP, D).
verbalphrase(VP, D) :- verb(VP, NP), nominalphrase(NP, D).
artikel([the | D], D).
nomen([man | D], D).
verb([sings | D], D).

?- satz([the, man, sings], [ ]).
     |
?- nominalphrase([the, man, sings], VP), verbalphrase(VP, [ ]).
     |
?- artikel([the, man, sings], N), nomen(N, VP), verbalphrase(VP, [ ]).
     | N = [man, sings]
?- nomen([man, sings], VP), verbalphrase(VP, [ ]).
     | VP = [sings]
```

```
?- verbalphrase([sings], [ ]).
     |
?- verb([sings], [ ]).
     | D = [ ]
     □
```

Beachte, dass man das Komma als Differenzlistenoperator benutzt. Daher erfolgt eine Anfrage bei SWI PROLOG-DCG's mit "*Differenzargument*", d. h. `satz` wird als binäres Prädikat verwendet:

```
?- satz([the, man, sings], [ ]) ⇝ Yes.
?- satz([the, man, sings, rubbish], [ ]) ⇝ No.
?- satz([the, man, sings, rubbish], [rubbish]) ⇝ Yes.
?- satz(X, [ ]) ⇝ L(G)
```

Beachte:

PROLOG-Auswerungsstrategie: "*top-down/left-to-right*"-Parser
Nichtdeterminismus erfordert Backtracking, was zu einem exponienziellen Aufwand führt.

5.4.2 Erweiterung von DCG's: Syntaxanalyse

Idee: Verwende ein zusätzliches Strukturargument.

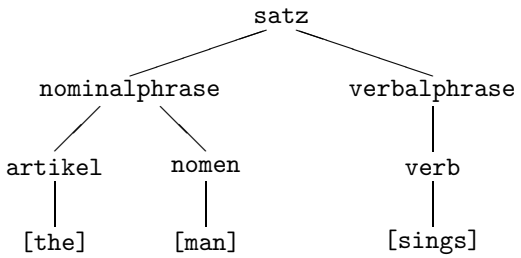
DCG-Grammatik:

```
satz(satz(NP, VP)) --> nominalphrase(NP), verbalphrase(VP).
nominalphrase(nominalphrase(A, N)) --> artikel(A), nomen(N).
verbalphrase(verbalphrase(V)) --> verb(V).
⋮
```

Übersetzung nach PROLOG:

```
satz(satz(NP, VP), S, D) :- nominalphrase(NP, S, V),
                            verbalphrase(VP, V, D).
⋮

?- satz(SB, [the, man, sings], [ ]).
⇝ SB = satz(nominalphrase(artikel(the), nomen(man)),
        verbalphrase(verb(sings))).
```



Eine weitere Verarbeitung von Syntaxbäumen ist möglich (z. B. Codeerzeugung in der KI). Es lassen sich sogar kontextsensitive Grammatiken in DCG's darstellen.

Kapitel 6

Datenbanken - DATALOG

Es besteht die Möglichkeit PROLOG als Anfragesprache für relationale Datenbanken zu benutzen und somit eine Kopplung von PROLOG an Datenbanksysteme zu schaffen (z. B. PRO-SQL).

Vorteile von PROLOG für Datenbanksysteme:

+ große Ausdruckskraft

+ rekursive Anfragen sind möglich (bei deduktiven Datenbanken)

Nachteile von PROLOG für Datenbanksysteme:

– PROLOG ist prozedural, d. h. abhängig von der Reihenfolge der Regeln – im Gegensatz zu nicht-prozeduralen Datenbanksprachen wie SQL oder der relationalen Algebra.

– es werden nur einzelne Antworten zurückgegeben – wegen der Unvollständigkeit der *depth-first-search*-Strategie

DATALOG bietet eine Alternative an, die die Nachteile von PROLOG behebt:

- nicht-prozedural, also unabhängig von der Reihenfolge der Regeln

- mengenorientiert, d. h. es wird die Menge aller Anworten zurückgegeben, indem die *breadth-first-search*-Strategie angewendet wird

- Einschränkung: keine Funktionssymbole
  $\rightsquigarrow$ keine komplexen Datenstrukturen
  Der Grund dafür ist, dass relationale Datenbanken – im Unterschied zu hierarchischen – "flach" sind

6.1 Syntax

Die Syntax von DATALOG unterscheidet, sich bis darauf, dass keine Funktoren verwendet werden (dürfen), nicht von der PROLOG-Syntax.

Folgerung:

- Term = Konstante oder Variable
- Grundterm = Konstante
- Atom (Primformel) = $p(t_1, \ldots, t_n)$, p: Prädikat, t_i: Term
- Grundatom = Atom ohne Variablen
- Literal = $p(t_1, \ldots, t_n)$ oder $\neg p(t_1, \ldots, t_n)$
- Grundliteral = Literal ohne Variablen

Wir definieren die **Herbrandbasis** als die Menge HB aller Grundatome.

Beachte:

Die Herbrandbasis ist unendlich, da es unendlich viele Konstanten und Prädikate gibt.

6.2 Extensionale Datenbanken und DATALOG-Programme

Unterscheide zwischen

- $EPred$: extensionale Prädikate (effektiv in der Datenbank gespeichert)
- $IPred$: intensionale Prädikate (definiert durch das DATALOG-Programm)

Entsprechend unterscheidet man zwischen den Mengen $E(HB)$ und $I(HB)$.

Definition 6.1 *Eine **extensionale Datenbank** ist eine endliche Teilmenge von $E(HB)$.*

Definition 6.2 *Ein DATALOG-**Programm** P ist eine endliche Menge von Regeln der Form:* `p(...) :- ...` *mit* `p` $\in IPred$*, und alle Variablen des Kopfes treten auch im Rumpf auf ("Variablenbedingung").*

Bemerkung:

- Die Fakten sind nicht in einem DATALOG-Programm enthalten, sondern in einer extensionalen Datenbank.
- Durch die Variablenbedingung ergibt sich eine endliche Anwortmenge.

Beispiel:

```
ahn(X, adam) :- person(X).
ahn(X, Y) :- eltern(X, Y).
ahn(X, Y) :- eltern(X, Z), ahn(Z, Y).
person(X) :- lebt(X, Y).
```

`eltern`, `lebt` $\in EPred$
`ahn`, `person` $\in IPred$

Definition 6.3 *Eine* DATALOG-***Anfrage*** *enthält genau ein Atom.*

Beispiel:

```
?- ahn(peter, Y).
```

Bemerkung:

Wird eine Anfrage weggelassen, so sucht man alle Antworten von P bezüglich einer extensionalen Datenbank $D \subseteq E(HB)$. Die Ausgabe ist dann eine endliche Teilmenge von $I(HB)$.

6.3 Deklarative Semantik

Sei P ein DATALOG-Programm und D eine extensionale Datenbank. P und D bestimmen die Mengen

- $Const$ der in P und D auftretenden Konstanten
- $EPred$ der in D auftretenden Prädikate
- $IPred$ der in P und nicht in D auftretenden Prädikate
- $Pred := EPred \cup IPred = \{p_1, \ldots, p_k\}$
- $HB := \{p(c_1, \ldots, c_n) \mid p \in Pred^{(n)}, c_i \in Const\} = I(HB) \cup E(HB)$ die Herbrandbasis von P und D

Die Folgerungsbeziehung $\models$

Es genügt die Betrachtung von Herbrand-Strukturen.
$\mathfrak{T} = \langle Const; p_1^{\mathfrak{T}}, \ldots, p_k^{\mathfrak{T}} \rangle$ mit $p_i \in Pred^{(n)} \curvearrowright p_i^{\mathfrak{T}} : Const^n \longrightarrow \{0,1\}$
$\mathfrak{T}$ ist darstellbar als Teilmenge von HB:
$\mathfrak{T}_M := \{p_i(c_1, \ldots, c_n) \mid p_i^{\mathfrak{T}}(c_1, \ldots, c_n) = 1\}$.
Dann gilt für $S = P \cup D$ und $F \in HB$: $S \models F \curvearrowright\!\!\curvearrowright$ jede Herbrand-Struktur, die S erfüllt, erfüllt auch F.

Definition 6.4 (Folgerungsmenge) $Cons(S) := \{F \mid F \in HB, S \models F\}$

Satz 6.1 *Es gilt:*

(1) $Cons(S) = \bigcap\{\mathfrak{T}_M \mid \mathfrak{T}$ *ist Herbrandmodell für* $S\}$

(2) $Cons(S)$ *ist das kleinste Herbrandmodell von* S.

Definition 6.5 *Die* ***deklarative Semantik*** *eines* DATALOG*-Programms bezüglich einer extensionalen Datenbank ist wie folgt definiert:*

(1) ohne Anfrage
$\mathcal{D}[\![P, D]\!] := \{F \mid F \in I(HB), (P \cup D) \models F\}$

(2) mit Anfrage $G =$ `?- p(t1,...,tn).` *(*`p` $\in IPred$, `ti` $\in Const \cup VAR$*)*
$\mathcal{D}[\![P, D, G]\!] := \{F \mid F \in \mathcal{D}[\![P, D]\!], F$ *ist Grundinstanz von* $G\}$

$\mathcal{D}[\![P, D, G]\!] \subseteq \mathcal{D}[\![P, D]\!] \subseteq I(HB)$

6.4 Operationelle Semantik

Definition 6.6 *Sei* $R = \mathtt{L_0}$ `:-` $\mathtt{L_1}, \ldots, \mathtt{L_n}$ *eine* DATALOG*-Regel und* $\overline{F} = (F_1, \ldots, F_n)$ *ein Tupel von Grundfakten. Wenn eine Grundsubstitution* σ *mit* $F_i = \mathtt{L_i}\sigma$ $(i = 1, \ldots, n)$ *existiert, so sagt man:*
$\mathtt{L_0}\sigma$ *ist aus* $\overline{F}$ *durch* ***Grundinferenz*** *mit der Regel* R ***direkt*** *(mit einem Schritt)* ***ableitbar****.*

Beachte:

$\mathtt{L_0}\sigma$ ist wegen der Variablenbedingung von R eindeutig.

Definition 6.7 *Sei* P *ein* DATALOG*-Programm und* D *eine extensionale Datenbank. Ein Grundfakt* F *heißt aus* $S = P \cup D$ ***durch Grundinferenz ableitbar*** *– Bezeichnung:* $S \vdash F$ *– wenn gilt:*

- $F \in D$
- *es gibt eine Regel* $R \in P$ *und Fakten* $F_1, \ldots, F_n$*, sodass* $S \vdash F_i$ *und* F *ist aus* $(F_1, \ldots, F_n)$ *durch Grundiferenz mit* R *direkt ableitbar.*

Definition 6.8 *Für ein* DATALOG*-Programm* P *mit extensionaler Datenbank* D *heißt* $FC[\![P, D]\!] := \{F \mid (P \cup D) \vdash F, F \in I(HB)\}$ *die* ***forward-chaining-Semantik*** *von* P *bezüglich* D*.*
$FC[\![P, D, G]\!] := \{F \mid F \in FC[\![P, D]\!],\ F$ *ist Grundinstanz von* $G\}$

Satz 6.2 $\mathcal{D}[\![P, D]\!] = FC[\![P, D]\!]$ *und* $\mathcal{D}[\![P, D, G]\!] = FC[\![P, D, G]\!]$

Bemerkung:

Grundinferenz bestimmt "*bottom-up*"-Berechnungen

- Start D
- neue Fakten sukkzessiv durch Grundinferenz bestimmen

Dieser Vorgang heißt *forward-chaining*. Eine Implementierung dieses Verfahrens ist die *Inferenz-Maschine*. Dazu gibt es einige Techniken zur Effizienzsteigerung. Die *forward-chaining*-Sematik entspricht der Fixpunktsematik, d. h. sie ist nicht zielgerichtet und eignet sich daher für das Finden aller Lösungen, nicht aber einer speziellen.

backward-chaining **und Resolution**

Gegeben sei ein DATALOG-Programm P mit extensionaler Datenbank D und Anfrage $G =$ `?-` $\mathtt{p(t_1, \ldots, t_n)}$. Wird $P \cup D$ als PROLOG-Programm aufgefasst, bestimmt G bezüglich $P \cup D$ einen SLD-Baum.

Beachte:

$P \cup D$ bestimmen endliche Mengen *Pred* und *Const*. Die Menge der Teilziele im SLD-Baum ist somit auch endlich. Daher sind unendliche Berechnungen erkennbar durch die Wiederholung eines Teilziels. Aus diesem Grund kann der SLD-Baum in der Tiefe beschränkt werden, sodass alle Lösungen mit einer beschränkten Tiefensuche bestimmbar sind (Vollständigkeit).

Bemerkung:

Für große Datenbanksysteme sind effiziente Methoden erforderlich.

Kapitel 7

Logikprogrammierung mit Constraints

7.1 Constraint-Programmierung

Gegeben sei eine Struktur $\mathfrak{A} = \langle A; f_1, \ldots, p_1, \ldots \rangle$ als semantischer Bereich. Dann heißt eine atomare Formel $p_i(t_1, \ldots, t_n)$ ein **Constraint** über $\mathfrak{A}$.

$P = \{constr_i \mid constr_i \text{ ist Constraint über } \mathfrak{A}, 1 \leq i \leq n\}$ heißt **Constraint-Problem** über $\mathfrak{A}$.

Sei $VAR(P)$ die Menge der in P auftretenden Variablen und $\beta : VAR(P) \longrightarrow A$ eine Variablenbelegung, dann heißt β eine **Lösung** von P, falls die Interpretation $\mathfrak{I} = (\mathfrak{A}, \beta)$ jedes Constraint von P erfüllt: $[\![constr_i]\!] = 1$, $i = 1, \ldots, n$.

Der **Lösungsraum** von P $[\![P]\!]$ ist die Menge aller Lösungen von P: $[\![P]\!] := \{\beta \mid \beta : VAR(P) \longrightarrow A, [\![constr_i]\!]^{\mathfrak{I}} = 1, i = 1, \ldots, n\}$.

Mit $P_i = \{constr_i\}$ folgt: $[\![P]\!] = \bigcap_{i=1}^{n} [\![P_i]\!]$.

Ein Constraint-Problem kann keine, genau eine oder mehrere Lösungen haben. Für ein Constraint-Problem über beliebiger Struktur $\mathfrak{A}$ gibt es keine allgemeine Lösungsstrategie.

Wichtige Spezialfälle

$\mathfrak{R} = \langle \mathbb{R}; +, -, \cdot, /, \ldots, =, \neq, <, >, \leq, \geq, \ldots \rangle$
ebenso: $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{B}$, endliche Bereiche, u. a.

typische Anwendungen

- *scheduling*
- Logistik
- *resource management:*
  - ◇ *production*
  - ◇ *transportation*
  - ◇ *placement*

Beispiel:

Vier Aufgaben a, b, c, d mit benötigten Arbeitszeiten: 2, 3, 4, 5 Stunden.
Planungsconstraints: a vor b, a vor c, b vor d.

Aufgabe: Bestimme Startzeiten $T_a, \ldots, T_d$ mit dem frühesten Endzeitpunkt.

$\mathfrak{R}^+ := \langle \mathbb{R}^+; +, \leq \rangle$
$VAR = \{T_a, T_b, T_c, T_d, T_{final}\}$

Constraint-Problem P:

$$\begin{array}{lcl} T_a + 2 & \leq & T_b \\ T_a + 2 & \leq & T_c \\ T_b + 3 & \leq & T_d \\ T_c + 5 & \leq & T_{final} \\ T_d + 4 & \leq & T_{final} \end{array}$$

$minimize(T_{final})$

$[\![P]\!] = \{T_a = 0, T_b = 2, 2 \leq T_c \leq 4, T_d = 5, T_{final} = 9\}$

Berechnung von Lösungen

Die Berechnung von Lösungen geschieht mit Hilfe der *operationellen Semantik*. Das Ziel hierbei ist die Erfüllung der Constraints.

Der Einfachheit halber gehen wir hier von binären Constraints aus, d. h.

- $c(X, Y) = p(t_1, \ldots, t_n)$ mit $VAR(t_i) = \{X, Y\}$
- $P = \{c_i(X_{i_1}, X_{i_2}) \mid i = 1, \ldots, n\}$
- $VAR(P) = \{X_1, \ldots, X_k\}$

P bestimmt ein Constraint-*Netzwerk*, d. h. einen gerichteten Graphen mit

- Knotenmenge: $VAR(P)$
- Kanten: $X \xrightarrow{c} Y$ und $Y \xrightarrow{c} X$ für jedes $c(X, Y)$

Idee: schrittweise Verkleinerung der Wertemengen der Variablen

Eine Kante $X \xrightarrow{c} Y$ heißt **konsistent** bezüglich der Wertemengen $D(X)$ und $D(Y)$, falls für jedes $a \in D(X)$ ein $b \in D(Y)$ existiert, welches das Constraint c erfüllt.

Ziel: alle Kanten konsitent machen

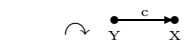
Beispiel:

$c(X, Y) = X + 4 \leq Y$
$D(X) = D(Y) = \{0, \ldots, 10\}$

(1) Entferne alle $a \in D(X)$, welche Konsistenz verletzen: $D(X) = \{0, \ldots, 6\}$

$\curvearrowright X \xrightarrow{c} Y$ konsistent

(2) Entferne alle $b \in D(Y)$, welche Konsistenz verletzen: $D(Y) = \{4, \ldots, 10\}$

$\curvearrowright$  konsistent

Wiederholung von (1) und (2) im c-Netzwerk, bis alle Kanten konsistent.

(1) $\exists X : D(X) = \emptyset \curvearrowright P$ unlösbar

(2) $\forall X \in VAR(P) : |D(X)| = 1 \curvearrowright P$ eindeutig lösbar

(3) Wertmengen mit mehreren Elementen $\curvearrowright$ keine Aussage möglich

Beachte:

Es werden nur solche Elemente entfernt, die mit Sicherheit nicht in der Lösung vorkommen können.

7.2 PROLOG mit Constraints

Constraint Logic Programming (CLP) ist eine Kombination von Logikprogrammierung und Constraint-Programmierung. Während die Logikprogrammierung auf Herbrand-Strukturen mit Termen und Unifikation arbeitet, ist die Grundlage der Constraintprogrammierung ein semantischer Bereich (z. B. $\mathbb{R}$, $\mathbb{N}$, ...). Da die CLP mit den reellen Zahlen rechnet verwendet man die Bezeichnung CLP($\mathbb{R}$).

Es ist in PROLOG nicht möglich eine Gleichung wie $1 + x = 5$ zu lösen:
`?- 1 + X = 5` $\rightsquigarrow$ `No.`
Dasselbe Ergebnis erhält man auch bei der Verwendung von `is` oder `=:=` statt `=`.
In SICSTUS PROLOG gibt es aber die Möglichkeit Constraints zu formulieren; dies geschieht mittels Einschließen in geschweifte Klammern:
`?- {1 + X = 5}` $\rightsquigarrow$ `X = 4.`
Ein eingebauter *Constraint-Solver* löst Gleichungen (GAUSS-Elimination) und Ungleichungen (Simplexverfahren).

Syntax für Constraints in SICSTUS PROLOG

Sei $P = \{constr_i \mid 1 \leq i \leq n\}$
P wird formuliert durch: {`constr`$_1$`,...,constr`$_1$}, wobei
`constr`$_i$ `= expr`$_1$ p `expr`$_2$,
`expr`$_i$: arithmetische Ausdrücke über $\mathbb{R}$,
$p \in$ {`=`, `=/=`, `<`, `=<`, `>`, `>=`}

Beispiel (Temperaturkonversion: C $\leftrightarrow$ F):

(1)
```
convert(C, F) :- C is (F - 32)*5/9
```
`?- convert(C, 50).` $\rightsquigarrow$ `C = 10.`
`?- convert(10, F).` $\rightsquigarrow$ `ERROR: Arguments ...!`

(2)
```
convert(C, F) :- {C = (F - 32)*5/9}
```
`?- convert(10, F).` $\rightsquigarrow$ `F = 50.`
`?- convert(C, F).` $\rightsquigarrow$ `{F = 32.0 + 1.8*C}.`

Beispiel (Lösen linearer Gleichungen):

```
?- {3*X - 2*Y = 6, 2*Y = X}.
⇝ X = 3.0, Y = 1.5.
```

Beispiel (Lösen linearer Ungleichungen):

```
?- {Z =< X - 2, Z =< 6 - X, Z + 1 = 2}.
⇝ Z = 1.0, {X >= 3.0}, {X =< 5.0}.
```

Optimierung mit `minimize(expr)` und `maximize(expr)`, wobei `expr` linear in Variablen der übrigen Constraints.

```
?- {X =< 5}, maximize(X). ⇝ X = 5.0.
?- {X =< 5}, minimize(X). ⇝ No.
?- {X =< 5, 2 =< X}, minimize(2*X + 3). ⇝ X = 2.0.

?- {Ta + 2 =< Tb, Ta + 2 =< Tc, Tb + 3 =< Td, Tc + 5 =< Tfin,
    Td + 4 =< Tfin, Ta >= 0}, minimize(Tfin).
⇝ Ta = 0.0, Tb = 2.0, Td = 5.0, Tfin = 9.0, {Tc =< 4.0},
  {Tc >= 2.0}.
```

7.2.1 Kombination von Logikprogrammierung und Constraint-Programmierung

Beispiel (Fibonacci-Funktion):

```
fib(N, F) :- {N = 0, F = 1};
             {N = 1, F = 1};
             {N > 1, F = F1 + F2, N1 = N - 1, N2 = N-2
              F1 >= N1, F2 >= N2, fib(N1, F1), fib(N2, F2)}.
?- fib(N, 13) ⇝ N = 6.
?- fib(N, 4) ⇝ No.
```

Beispiel (Hypothekenberechnung):

Parameter:		
	`P`	Kapital
	`T`	Laufzeit in Monaten
	`IR`	monatlicher Zins
	`B`	Restbetrag
	`MP`	monatliche Rückzahlung

mortgage.pl:

```
:- use_module(library(clpr)).

mortgage(P, T, IR, B, MP) :- {T =< 1, T > 0, B = P*(1 + T*IR) - T*MP};
                             {T > 1}, mortgage(P*(1 + IR) - MP, T - 1,
                                               IR, B, MP).
```

% Monatliche Rückzahlung einer Hyphothek:

```
?- mortgage(100000, 180, 0.01, 0, MP).
⇝ MP = 1200.17.
```

% Zeitdauer zur Finanzierung einer Hypothek:
`?- mortgage(100000, T, 0.01, 0, 1400).`
$\rightsquigarrow$ `T = 125.901.`

% Relation zwischen `P`, `B` und `MP`:
`?- mortgage(P, 180, 0.01, B, MP).`
$\rightsquigarrow$ {`P = 0.166783*B + 83.3217*MP`}.

Fazit

CLP($\mathbb{R}$) ist eine geschickte Kombination von Unifikation mit Termen und Contraint-Lösung in $\mathbb{R}$. Hierbei erhält man die Invertierbarkeit.

Kapitel 8

Meta-Programmierung

Ziel: Programme als Daten (z. B. Compiler, Interpreter, ...)

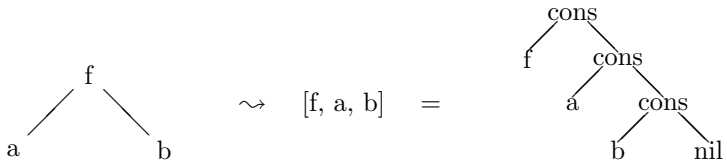
Da PROLOG auf Termen, d. h. symbolisch arbeitet, eignet es sich gut für die *Metaprogrammierung*. Eine besondere Eigenschaft von PROLOG ist die implizite Deklaration: die Bedeutung eines Bezeichners wird durch seine Position im Literal klar.

8.1 Verarbeitung von Termen

bisher: Unifikation durch Substitution von Variablen,
keine Veränderung von Funktoren

Idee:

Terme umstrukturieren:



Folge: Funktoren werden zu Konstanten.

Meta-Prädikat in SWI PROLOG: `T =.. L` ist erfüllt, wenn `L` die Listendarstellung des Terms `T` ist.

Beispiel:

```
?- f(a, g(b)) =.. L.
⇝ L = [f, a, g(b)].

?- T =.. [f, g(a), c].
⇝ T = f(g(b), c).
```

Beispiel (Vergrößern geometrischer Figuren):

Figuren als Terme:
`square(Side), triangle(Side1, Side2, Side3), circle(Radius), ...`

`enlarge(Fig, Faktor, Fig1)` ist erfüllt, wenn `Fig1` aus `Fig` durch Vergrößerung um `Faktor` hervorgeht.

1. Lösung:

```
enlarge(square(A), F, square(A1)) :- A1 is A*F.
enlarge(triangle(A, B, C), F, triangle(A1, B1, C1) :- ...
⋮
```

2. Lösung:

```
enlarge(Fig, F, Fig1) :- Fig =.. [Type | Pars],
                         multlist(Pars, F, Pars1),
                         Fig1 = [Type | Pars1].
multlist([ ], _, [ ]).
multlist([X | L], F, [X1 | L1]) :- X1 is X*F, multlist(L, F, L1).
```

8.1.1 Funktor und Argument eines Terms

Meta-Prädikate in SWI Prolog:

- `functor(T, F, N)` ist erfüllt, wenn `F` der Funktor des Terms `T` mit der Stelligkeit `N` ist.
  `?- functor(t(f(X), X, s), F, N).`
  $\rightsquigarrow$ `F = t, N = 3.`
- `arg(N, T, A)` ist erfüllt, wenn `A` das `N`-te Argument des Terms `T` ist.
  `?- arg(2, f(X, t(a), t(b)), Y).`
  $\rightsquigarrow$ `Y = t(a).`

Beispiel:

```
?- functor(D, date, 3), arg(1, D, 25), arg(2, D, april),
   arg(3, D, 1979).
⇝ D = date(25, april, 1979).
```

Anwendung:

- Termersetzung
- symbolisches Differenzieren

8.2 Verarbeitung von Programmen

Programme können als Terme aufgefasst werden. Prolog eignet sich daher zur Implementierung von Programmiersprachen (vgl. Lisp), insbesondere zum *rapid prototyping*. Hier konzentrieren wir uns auf die Anwendung auf Prolog selbst (Interpreter von Prolog in Prolog).

8.2.1 Meta-Interpreter

Er definiert für ein beliebiges PROLOG-Programm und eine Anfrage eine Auswertungsstrategie. Ferner ist eine andere Funktionalität möglich (z. B. *Tracer*, Beweisbäume, Beweisstrategien). Dargestellt wird der Meta-Interpreter als das PROLOG-Prädikat `prove` mit einer Anfrage als Argumentterm und Programmklauseln als Fakten.

Darstellung einer Programmklausel

Sei $K =$ `L0 :- L1,...,Ln.` (`n` ≤ 0) eine Programmklausel. K wird dargestellt als Fakt mit binärem System-Prädikat `clause/2`:

`clause(L0, t(L1,...,Ln)).` mit

- `t()` $\rightsquigarrow$ `true`
- `t(L1)` $\rightsquigarrow$ `L1`
- `t(L1,...,Ln)` $\rightsquigarrow$ `(L1, t(L2,...,Ln))`

Dabei wirkt der binäre Infixoperator `(_ , _)` als Paar-Konstruktor.

Beispiel:

$K =$ `L0 :- L1, L2, L3.`

`clause(` $\underbrace{\texttt{L0}}_{1.}$ `,` $\underbrace{\texttt{(L1, (L2, L3))}}_{2.}$ `)`

Beachte:

`clause` hat zwei Argumentterme, 2. Term im Beispiel: `(L1, (L2, L3))`

Beispiel:

```
?- clause(member(X, L), Body).
~> X = _G287, L = [_G287 | _G383], Body = true.
~> X = _G287, L = [_G385 | _G386], Body = member(_G287, _G386).
~> No.

?- clause(X, Y).
~> ERROR: Arguments ...!
```

Meta-Interpreter für PROLOG

(1) Der triviale Meta-Interpreter

```
prove(G) :- G.

?- prove(member(X, [a, b])).
     |
?- member(X, [a, b]).
     ⋮
```

Da dies eigentlich nicht der korrekten PROLOG-Syntax entspricht, funktioniert dieser triviale Meta-Interpreter auch nur in SWI PROLOG. Korrekter wäre:

```
prove(G) :- call(G).,
```

was das Systemprädikat `call/1` für die Auswertung von `G` benutzt.

(2) Der Basis-Meta-Interpreter

```
prove(true).
prove((A, B)) :- prove(A), prove(B).
prove(A) :- clause(A, G), prove(G).
```

Man kann den Basis-Meta-Interpreter um "sinnvolle" Funktionalitäten erweitern. Ein Beispiel dafür ist die Anzeige der Beweislänge:

```
lprove(true, 0).
lprove((A, B), N) :- lprove(A, NA), lprove(B, NB), N is NA + NB.
lprove(A, N) :- clause(A, G), lprove(G, NG), N is NG + 1.


?- lprove(append([a, b], [b, c], E), N).
⇝ E = [a, b, b, c], N = 3.
```

Hierbei muss man beachten, dass die Beweislänge `N` kleiner als die Berechnungslänge des Meta-Interpreters ist. Weitere Möglichkeiten der Erweiterung sind *Tracer*, *Debugger*, Expertensysteme, u. a.

Index