

Logische und funktionale Programmierung

Vorlesung 12: Universalität der Logikprogrammierung

Babeş-Bolyai Universität, Department für Informatik, Cluj-Napoca
csacarea@cs.ubbcluj.ro



- Die Logikprogrammierung ist eine vollwertige Programmiersprache:
 - Man kann durch Logikprogramme **jede berechenbare** Funktion berechnen.
 - Solche Programmiersprachen heißen **Turing vollständig**.
- Wir beschränken uns bei der Betrachtung der berechenbaren Funktionen auf arithmetische Funktionen $f: \mathbb{N}^n \rightarrow \mathbb{N}$.
- **Alle anderen Datenstrukturen lassen sich durch eine entsprechende Abbildung in die natürlichen Zahlen codieren.**

BERECHENBARE FUNKTIONEN

- ...
- Turing Maschinen
- μ -rekursive Funktionen
- **Churchschen These:** Diese Mengen entsprechen der Mengen, der im intuitiven Sinne berechenbaren Funktionen.



μ -REKURSIVE FUNKTIONEN

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

- 1 Für jedes $n \in \mathbb{N}$ ist die Funktion $\text{null}_n: \mathbb{N}^n \rightarrow \mathbb{N}$ mit $\text{null}_n(k_1, \dots, k_n) = 0$ μ -rekursiv.
- 2 Die Nachfolgerfunktion $\text{succ}: \mathbb{N} \rightarrow \mathbb{N}$ mit $\text{succ}(k) = k + 1$ ist μ -rekursiv.
- 3 Für jedes $n \geq 1$ und jedes $1 \leq i \leq n$ ist die Projektionsfunktion $\text{proj}_{n,i}(k_1, \dots, k_n) = k_i$ μ -rekursiv.
- 4 Die μ -rekursiven Funktionen sind unter Komposition abgeschlossen.



μ -REKURSIVE FUNKTIONEN

- 5 Die μ -rekursiven Funktionen sind unter **primitiver Rekursion** abgeschlossen. Für alle $n \geq 0$ gilt: Falls $f: \mathbb{N}^n \rightarrow \mathbb{N}$ und $g: \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ μ -rekursiv sind, dann ist auch die folgende Funktion $h: \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ μ -rekursiv:

$$h(k_1, \dots, k_n, 0) = f(k_1, \dots, k_n)$$

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

Primitive Rekursion bedeutet also, dass die Definition von $h(\dots, k+1)$ auf die Definition von $h(\dots, k)$ zurückgeführt wird.



μ -REKURSIVE FUNKTIONEN

- ⑥ Die μ -rekursiven Funktionen sind unter (unbeschränkter) **Minimalisierung** abgeschlossen. Für alle $n \geq 0$ gilt: Falls $f: \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ μ -rekursiv ist, dann ist auch die folgende Funktion $g: \mathbb{N}^n \rightarrow \mathbb{N}$ μ -rekursiv:
- $$g(k_1, \dots, k_n) = k \text{ gdw. } f(k_1, \dots, k_n, k) = 0 \text{ und für alle } 0 \leq k' < k \text{ ist } f(k_1, \dots, k_n, k') \text{ definiert und } f(k_1, \dots, k_n, k') > 0.$$
- Falls es kein solches k gibt, dann ist $g(k_1, \dots, k_n)$ undefiniert, d.h., durch die Minimalisierung können auch partielle Funktionen entstehen.

Die Klasse der Funktionen, die nur mit Hilfe der Punkte 1 - 5 konstruiert werden, ist die Klasse der **primitiv rekursiven Funktionen**.



- Die Klasse der primitiv rekursiven Funktionen kann nicht alle berechenbaren Funktionen enthalten, da alle primitiv rekursiven Funktionen **total** sind und es aber (durch die Verwendung von nichtterminierenden Programmen) offensichtlich auch berechenbare partielle Funktionen gibt.
- Es existieren aber auch totale berechenbare und nicht primitiv rekursive Funktionen, wie z.B. die sogenannte **Ackermann - Funktion**.

BEISPIEL

PRIMITIV REKURSIVE FUNKTIONEN

- Die Additionsfunktion $\text{plus}: \mathbb{N}^2 \rightarrow \mathbb{N}$ ist primitiv rekursiv, denn sie kann wie folgt mit Hilfe von primitiver Rekursion dargestellt werden. Hierbei ist f die dreistellige Funktion mit $f(x, y, z) = z + 1$:

$$f(x, y, z) = \text{succ}(\text{proj}_{3,3}(x, y, z))$$

$$\text{plus}(x, 0) = \text{proj}_{1,1}(x)$$

$$\text{plus}(x, y + 1) = f(x, y, \text{plus}(x, y))$$

- Analog die Multiplikationsfunktion, die Vorgängerfunktion, oder die minus Funktion sind primitiv rekursiv.



BEISPIEL

μ -REKURSIVE FUNKTIONEN: DIV

$\text{div}(x, y) = \lceil \frac{x}{y} \rceil$ und $\text{div}(0, 0) = 0$. Hingegen ist $\text{div}(x + 1, 0)$ undefiniert.

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

Hierbei berechnet $i(x, y, z) = x - (y \cdot z)$, d.h.

$$\begin{aligned} i(x, y, z) &= \text{minus}(\text{proj}_{3,1}(x, y, z), j(x, y, z)) \\ j(x, y, z) &= \text{times}(\text{proj}_{3,2}(x, y, z), \text{proj}_{3,3}(x, y, z)) \end{aligned}$$



- Wir wollen zeigen, dass jede berechenbare (d.h. jede μ -rekursive Funktion) auch durch ein Logikprogramm berechnen kann.
- Kann man Funktionen mit Logikprogrammen berechnen?
← **Relationen!**
- Betrachte statt der n -stelligen Funktion $f: \mathbb{N}^n \rightarrow \mathbb{N}$ die $(n + 1)$ -stellige Relation, die dem Graphen von f entspricht.

DARSTELLUNG NATÜRLICHER ZAHLEN DURCH TERME

- $0 \in \Sigma_0, s \in \Sigma_1$
- $0, s(0), s(s(0)), \dots$



BERECHNUNG ARITH. FUNKTIONEN DURCH LOGIKPROGRAMME

- Jede Zahl $k \in \mathbb{N}$ wird durch den Term $\underline{k} \in \mathcal{T}(\Sigma, \mathcal{V})$ mit $\underline{k} = s^k(0)$ dargestellt.
- Ein Logikprogramm \mathcal{P} über (Σ, Δ) berechnet eine arithmetische Funktion $f: \mathbb{N}^n \rightarrow \mathbb{N}$ gdw. es ein Prädikatssymbol $f \in \Delta_{n+1}$ gibt, so dass

$$f(k_1, \dots, k_n) = k \text{ gdw. } \mathcal{P} \models f(\underline{k}_1, \dots, \underline{k}_n, \underline{k}).$$

Die Motivation dafür ist, dass man dann die Funktion f durch Anfragen an das Logikprogramm berechnen lassen kann. Um $f(k_1, \dots, k_n)$ zu berechnen, stellt man dem Logikprogramm die Anfrage $? - f(\underline{k}_1, \dots, \underline{k}_n, \underline{X})$.



BEISPIEL

plus(X,0,X).
plus(X,s(Y),s(Z)) :- plus(X,Y,Z).

times(X,0,0).
times(X,s(Y),Z) :- times(X,Y,U), plus(X,U,Z).

p(0,0).
p(s(X),X).

minus(X,0,X).
minus(X,s(Y),Z) :- minus(X,Y,U), p(U,Z).

div(0,Y,0).
div(s(X),s(Y),s(Z)) :- minus(X,Y,U), div(U,s(Y),Z).



UNIVERSALITÄT DER LOGIKPROGRAMMIERUNG

Theorem (Universalität der Logikprogrammierung)

Jede μ -rekursive Funktion ist durch ein Logikprogramm berechenbar.



BEWEIS

Der Beweis wird durch Induktion über den Aufbau der Klasse der μ -rekursiven Funktionen geführt.

- 1 Die Funktion $null_n$ wird durch das folgende Logikprogramm berechnet:

$$\underline{null}_n(X_1, \dots, X_n, 0).$$

- 2 Die Nachfolgerfunktion $succ$ wird durch das folgende Logikprogramm berechnet:

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

- 3 Die Projektionsfunktion $proj_{n,i}$ wird durch das folgende Logikprogramm berechnet:

$$\underline{proj}_{n,i}(X_1, \dots, X_n, X_i).$$



BEWEIS - FORTSETZUNG

- ④ Nun zeigen wir, wie die Komposition durch Logikprogramme realisiert werden kann. Seien $f: \mathbb{N}^m \rightarrow \mathbb{N}$ und $f_1, \dots, f_m: \mathbb{N}^n \rightarrow \mathbb{N}$ μ -rekursiv. Dann existiert nach der Induktionshypothese ein Logikprogramm mit Prädikaten $\bar{f} \in \Delta_{m+1}$ und $\bar{f}_1, \dots, \bar{f}_m \in \Delta_{n+1}$, das diese Funktionen berechnet. Sei \bar{g} durch Komposition von f mit f_1, \dots, f_m definiert, d.h.

$$g(k_1, \dots, k_n) = f(f_1(k_1, \dots, k_n), \dots, f_m(k_1, \dots, k_n)).$$

Zur Berechnung von g wird das Logikprogramm um die folgende Klausel ergänzt:

$$\bar{g}(X_1, \dots, X_n, Z) : -\bar{f}_1(X_1, \dots, X_n, Y_1), \dots, \\ \bar{f}_m(X_1, \dots, X_n, Y_m), \bar{f}(\bar{Y}_1, \dots, Y_m, Z).$$



BEWEIS - FORTSETZUNG

- 5 Als nächstes zeigen wir, wie die primitive Rekursion in der Logikprogrammierung realisiert werden kann. Seien $f: \mathbb{N}^n \rightarrow \mathbb{N}$ und $g: \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ μ -rekursiv. Dann existiert nach der Induktionshypothese ein Logikprogramm mit Prädikaten $\underline{f} \in \Delta_{n+1}$ und $\underline{g} \in \Delta_{n+3}$, das diese Funktionen berechnet. Sei h durch primitive Rekursion mit f und g definiert, d.h.

$$h(k_1, \dots, k_n, 0) = f(k_1, \dots, k_n)$$

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

Zur Berechnung von h wird das Logikprogramm um die folgenden Klauseln ergänzt:

$$\underline{h}(X_1, \dots, X_n, 0, Z) : \neg f(X_1, \dots, X_n, Z).$$

$$\underline{h}(X_1, \dots, X_n, s(X), Z) : \neg \underline{h}(X_1, \dots, X_n, X, Y),$$

$$\underline{g}(X_1, \dots, X_n, X, Y, Z).$$



BEWEIS - FORTSETZUNG

- 6 Schließlich zeigen wir, wie die Minimalisierung durch Logikprogramme realisiert werden kann. Sei $f: \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ μ -rekursiv. Dann existiert nach der Induktionshypothese ein Logikprogramm mit einem Prädikat $\bar{f} \in \Delta_{n+2}$, das diese Funktion berechnet. Sei g durch Minimalisierung mit \bar{f} definiert, d.h.

$g(k_1, \dots, k_n) = k$ gdw. $\bar{f}(k_1, \dots, k_n, k) = 0$ und für alle $0 \leq k' < k$ ist

$\bar{f}(k_1, \dots, k_n, k')$ definiert und $\bar{f}(k_1, \dots, k_n, k') > 0$.

Zur Berechnung von g wird das Logikprogramm um die folgenden Klauseln ergänzt.



BEWEIS - FORTSETZUNG

Hierbei gilt $f'(X_1, \dots, X_n, Y, Z)$ gdw. $f(X_1, \dots, X_n, Z) = 0$ und für alle X mit $\bar{Y} \leq X < Z$ ist $f(X_1, \dots, X_n, X) > 0$.

$$\underline{g}(X_1, \dots, X_n, Z) : -\underline{f}'(X_1, \dots, X_n, 0, Z).$$

$$\underline{f}'(X_1, \dots, X_n, Y, Y) : -\underline{f}(X_1, \dots, X_n, Y, 0).$$

$$\underline{f}'(X_1, \dots, X_n, Y, Z) : -\underline{f}(X_1, \dots, X_n, Y, s(U)),$$

$$\underline{f}'(X_1, \dots, X_n, s(Y), Z).$$



BEISPIEL

PLUS

proj_{3,3}(X, Y, Z).

succ(X, s(X)).

f(X, Y, Z, V) :- proj_{3,3}(X, Y, Z, U), succ(U, V).

proj_{1,1}(X, X).

plus(X, 0, U) :- proj_{1,1}(X, U).

plus(X, s(Y), U) :- plus(X, Y, Z), f(X, Y, Z, U).



BEISPIEL

DIV

$$\underline{\text{div}}(X1, X2, Z) :- \underline{i}'(X1, X2, 0, Z).$$

$$\underline{i}'(X1, X2, Y, Y) :- \underline{i}(X1, X2, Y, 0).$$

$$\underline{i}'(X1, X2, Y, Z) :- \underline{i}(X1, X2, Y, s(U)), \underline{i}'(X1, X2, s(Y), Z).$$

Hierbei gilt $\underline{i}(X, Y, Z, U)$ gdw. $X - (Y \cdot Z) = U$ ist.

INDETERMINISMUS UND AUSWERTUNGSSTRATEGIEN

- Um Logikprogramme auszuführen, geht man analog zur Definition der prozeduralen Semantik vor.
- Falls an das Logikprogramm \mathcal{P} die Anfrage G gestellt wird, versucht man also, eine Berechnung $(G, \emptyset) \vdash_{\mathcal{P}}^+ (\square, \sigma)$ zu finden und gibt dann als Ergebnis die Antwortsubstitution σ eingeschränkt auf die Variablen von G aus.
- Die Definition von $\vdash_{\mathcal{P}}^+$ weist bei jedem Schritt $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ zwei Indeterminismen auf.



- **Indeterminismus 1. Art:** Dieser Indeterminismus ist die Wahl der Programmklausel $K \in \mathcal{P}$, mit der G_1 resolviert werden soll.
- **Indeterminismus 2. Art:** Dieser Indeterminismus ist die Wahl des Literals A_i in G_1 , das zur Resolution verwendet werden soll.

Zu einem (G_1, σ_1) kann es mehrere (G_2, σ_2) mit $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2)$ geben.

BEISPIEL

```
mutterVon(renate, susanne).  
mutterVon(susanne, aline).  
vorfahre(V,X) :- mutterVon(V,X).  
vorfahre(V,X) :- mutterVon(V,Y),  
vorfahre(Y,X).  
?- vorfahre(Z,aline).
```


BEISPIEL

INDETERMINISMUS 1. ART

Im ersten Berechnungsschritt gibt es nun zwei Möglichkeiten, da man mit zwei verschiedenen Programmklauseln resolvieren kann. Dies entspricht also dem Indeterminismus 1. Art:

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

Falls man sich für den zweiten Berechnungsschritt entscheidet, so muss man nun die Anfrage

?- `mutterVon(Z, Y), vorfahre(Y, aline)`.
bearbeiten.



BEISPIEL

INDETERMINISMUS 2. ART

- Falls man sich zur Resolution mit Hilfe des ersten Literals $\neg \text{mutterVon}(Z, Y)$ entscheidet, so gibt es (wegen des Indeterminismus 1. Art und der zwei `mutterVon`-Fakten) zwei Möglichkeiten fortzufahren.
- Resolviert man mit der Klausel $\{\text{mutterVon}(\text{susanne}, \text{aline})\}$, so kann man keine erfolgreiche Berechnung mehr erhalten.
- Resolviert man hingegen mit der Klausel $\{\text{mutterVon}(\text{renate}, \text{susanne})\}$, so kann man eine erfolgreiche Berechnung erzeugen, die zur Antwortsubstitution $\{Z/\text{renate}\}$ führt.



BEISPIEL

INDETERMINISMUS 2. ART

- Falls man sich zur Resolution mit Hilfe des zweiten Literals $\neg \text{vorfahre}(Y, \text{aline})$ entscheidet, so gibt es ebenfalls (wegen des Indeterminismus 1. Art und der zwei vorfahre-Fakten) zwei Möglichkeiten fortzufahren etc.
- Insbesondere gibt es auch die Möglichkeit einer unendlichen Berechnung, indem man beim Indeterminismus 1. Art stets die zweite (rekursive) vorfahr-Klausel nimmt und beim Indeterminismus 2. Art stets das letzte vorfahre-Literal.

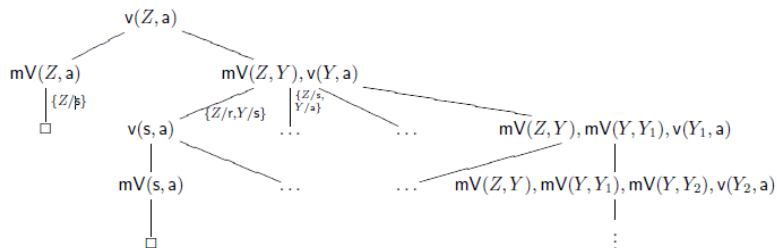


BEISPIEL

- Der folgende Baum zeigt die Möglichkeiten für Berechnungen auf.
- Hierbei wurde in den Knoten anstelle eines Paares $(\{\neg A_1, \dots, \neg A_k\}, \sigma)$ jeweils nur die Atome A_1, \dots, A_k in der Anfrage angegeben.
- Ein Berechnungsschritt der Form $(G_1, \sigma) \vdash_{\mathcal{P}} (G_2, \sigma' \circ \sigma)$ wird durch eine Kante dargestellt, die mit σ' (eingeschränkt auf die Variablen in G_1) markiert wird.
- Auf diese Weise kann man bei erfolgreichen Pfaden direkt die Antwortsubstitution ablesen.
- Zur Abkürzung steht **v** für **v**orfahre, **mV** für **m**utter**V**on, **s** für **s**usanne, **a** für **a**line und **r** für **r**enate.



BEISPIEL



BEISPIEL

- Der Indeterminismus 1. Art beeinflusst die Lösung: bei der linken erfolgreichen Berechnung erhält man die Antwortsubstitution $\{Z/susanne\}$ und bei der zweiten erfolgreichen Berechnung erhält man $\{Z/renate\}$.
- Der Indeterminismus 2. Art beeinflusst, ob man eine unendliche oder eine endliche Berechnung erhält, da der rechteste Pfad in diesem Baum unendlich ist.

- Um Logikprogramme auf einem (deterministischen) Rechner ausführen zu können, müssen wir diese beiden Indeterminismen auflösen.
- Wir müssen also eindeutig festlegen, in welches Paar (G_2, σ_2) ein Paar (G_1, σ_1) überführt werden soll.
- Der Indeterminismus 2. Art, d.h., die Auswahl des Literals in der Anfrage, das zur Resolution verwendet werden soll ist **unerheblich**, da er das Ergebnis (bei erfolgreichen Berechnungen) nicht beeinflusst.

VERTAUSCHUNGSLEMMA

Seien $A_1, \dots, A_k, B, C_1, \dots, C_n, D, E_1, \dots, E_m \in \mathcal{A} \sqcup (\Sigma, \Delta, \mathcal{V})$.
Seien $\{\neg A_1, \dots, \neg A_k\}, \{B, \neg C_1, \dots, \neg C_n\}$ und
 $\{D, \neg E_1, \dots, \neg E_m\}$ jeweils paarweise variablendisjunkt. Sei σ_1
der **mgu** von A_i und B und σ_2 der **mgu** von $\sigma_1(A_j)$ und D mit
 $j \neq i$. Dann sind daher offensichtlich die folgenden
SLD-Resolutionsschritte möglich:

$$\begin{array}{ccc}
 \{\neg A_1, \dots, \underline{\neg A_i}, \dots, \neg A_j, \dots, \neg A_k\} & & \{\underline{B}, \neg C_1, \dots, \neg C_n\} \\
 \downarrow & \swarrow & \\
 \sigma_1(\{\neg A_1, \dots, \neg C_1, \dots, \neg C_n, \dots, \underline{\neg A_j}, \dots, \neg A_k\}) & & \{\underline{D}, \neg E_1, \dots, \neg E_m\} \\
 \downarrow & \swarrow & \\
 \sigma_2(\sigma_1(\{\neg A_1, \dots, \neg C_1, \dots, \neg C_n, \dots, \neg E_1, \dots, \neg E_m, \dots, \neg A_k\})) & &
 \end{array}$$

Dann existieren auch ein mgu σ'_1 von A_j und D und ein mgu σ'_2 von $\sigma'_1(A_i)$ und B . Daher sind dann die folgenden SLD-Resolutionsschritte möglich:

$$\begin{array}{ccc}
 \{\neg A_1, \dots, \neg A_i, \dots, \underline{\neg A_j}, \dots, \neg A_k\} & & \{\underline{D}, \neg E_1, \dots, \neg E_m\} \\
 | & \swarrow & \\
 \sigma'_1(\{\neg A_1, \dots, \underline{\neg A_i}, \dots, E_1, \dots, \neg E_m, \dots, \neg A_k\}) & & \{\underline{B}, \neg C_1, \dots, \neg C_n\} \\
 | & \swarrow & \\
 \sigma'_2(\sigma'_1(\{\neg A_1, \dots, \neg C_1, \dots, \neg C_n, \dots, \neg E_1, \dots, \neg E_m, \dots, \neg A_k\})) & &
 \end{array}$$

Die Substitutionen $\sigma_2 \circ \sigma_1$ und $\sigma'_2 \circ \sigma'_1$ unterscheiden sich nur durch eine Variablenumbenennung.

BEISPIEL

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

und die Anfrage

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

Wenn man erst mit dem p -Literal und dann mit dem q -Literal resoltiert, erhält man z.B.

$$\begin{aligned}(\{\neg p(X, Y), \neg q(X)\}, \emptyset) &\vdash_p (\{\neg r(Z), \neg q(Z)\}, \{X/Z, Y/Z\}) \\ &\vdash_p (\{\neg r(Z)\}, \{W/Z\} \circ \{X/Z, Y/Z\})\end{aligned}$$

Wenn man hingegen erst mit dem q -Literal und dann mit dem p -Literal resoltiert, ergibt sich z.B.

$$\begin{aligned}(\{\neg p(X, Y), \neg q(X)\}, \emptyset) &\vdash_p (\{\neg p(W, Y)\}, \{X/W\}) \\ &\vdash_p (\{\neg r(Y)\}, \{W/Y, Z/Y\} \circ \{X/W\})\end{aligned}$$

Die beiden Resolventen und berechneten Substitutionen sind also bis auf die Variablenumbenennung $\nu = \{Y/Z, Z/Y\}$ gleich.



- Aus dem Vertauschungslemma folgt, dass man eine beliebige Ordnung der Literale in den Anfragen wählen kann und sich darauf einschränken kann, immer nur das **erste** Literal nach dieser Ordnung zur Resolution zu verwenden.
- Sofern es überhaupt eine erfolgreiche Berechnung gibt, gibt es dann auch eine Berechnung, die dieser Einschränkung genügt.
- Wie am Anfang erwähnt, betrachten wir daher Klauseln in der Logikprogrammierung als Folgen statt als Mengen von Literalen. Man kann nun also eine beliebige Selektionsfunktion wählen, die jeweils aus einer Anfrageklausel das zur Resolution verwendete Literal aussucht.
- Nun wird damit also auch die Bedeutung der **selection function** in der Abkürzung **SLD** deutlich.



- In der Programmiersprache Prolog wird die Selektionsfunktion verwendet, die stets das erste (bzw. das linkeste) Literal in der Anfrageklausel auswählt.
- Diese Einschränkung der Berechnungsfolgen (mit der Relation $\vdash_{\mathcal{P}}$) bezeichnen wir als kanonische Berechnungen.

KANONISCHE BERECHNUNG

Definition

Eine Berechnung $(G_1, \sigma_1) \vdash_{\mathcal{P}} (G_2, \sigma_2) \vdash_{\mathcal{P}} \dots$ eines Logikprogramms \mathcal{P} heißt *kanonisch* gdw. bei jedem Resolutionsschritt mit dem ersten Literal der jeweiligen Klausel G_i resoliert wird.

- Um zu zeigen, dass man sich in der Tat auf kanonische Berechnungen einschränken kann, benötigen wir noch das folgende Lemma.
- Es besagt, dass Variablenumbenennungen bei Berechnungen unerheblich sind.
- Wenn sich also zwei Anfragen nur durch eine Variablenumbenennung unterscheiden, dann kann man mit ihnen dieselben Berechnungen durchführen und erhält Antwortsubstitutionen, die ebenfalls bis auf Variablenumbenennungen gleich sind.

VARIABLENUMBENENNUNGEN BEI BERECHNUNGEN

Lemma

Sei $l \in \mathbb{N}$. Falls $(G, \sigma) \vdash_{\mathcal{P}}^l (G', \sigma')$ gilt und ν eine Variablenumbenennung ist, so gilt auch $(\nu(G), \nu \circ \sigma) \vdash_{\mathcal{P}}^l (\nu(G'), \nu \circ \sigma')$.

BEISPIEL

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

und die Anfrage

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

Es gilt

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

für alle Substitutionen σ . Sei nun $\nu = \{X/Y, Y/U, U/X\}$ eine Variablenumbenennung. Dann gilt

$$\begin{aligned}(\nu(\{\neg p(X, Y), \neg q(X)\}), \nu \circ \sigma) &= (\{\neg p(Y, U), \neg q(Y)\}, \nu \circ \sigma) \\ &\vdash_{\mathcal{P}} (\{\neg r(U), \neg q(U)\}, \{Y/U, Z/U\} \circ \nu \circ \sigma) \\ &= (\nu(\{\neg r(Y), \neg q(Y)\}), \{Y/U, Z/U\} \circ \nu \circ \sigma).\end{aligned}$$

Hierbei gilt

$$\begin{aligned}\{Y/U, Z/U\} \circ \nu &= \{Y/U, Z/U\} \circ \{X/Y, Y/U, U/X\} \\ &= \{X/U, Y/U, Z/U, U/X\}.\end{aligned}$$

Dies ist daher die gleiche Substitution wie

$$\begin{aligned}\nu \circ \{X/Y, Z/Y\} &= \{X/Y, Y/U, U/X\} \circ \{X/Y, Z/Y\} \\ &= \{X/U, Y/U, Z/U, U/X\}.\end{aligned}$$

- Nun zeigen wir den gewünschten Satz, der besagt, dass man sich auf kanonische Berechnungen einschränken kann.
- Jede erfolgreiche Berechnung ist auch dann noch möglich.
- Somit kann man den Indeterminismus 2. Art also eliminieren.
- Wir werden uns daher im Folgenden auf kanonische Berechnungen beschränken.

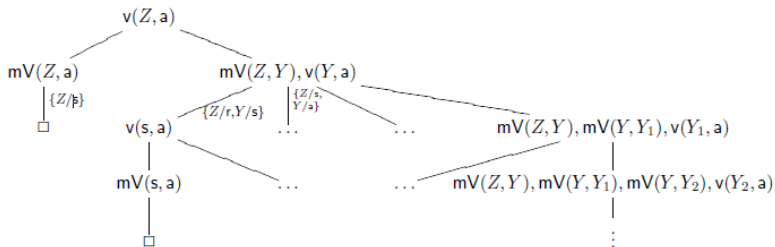
AUFLÖSUNG DES INDETERMINISMUS 2. ART

Theorem

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

BEISPIEL

Im Beispiel



können wir uns nun also auf kanonische Berechnungen einschränken und dabei sicher sein, dass wir immer noch alle Lösungen finden. Wenn man in dem Baum alle nicht-kanonischen Berechnungen löscht (d.h., wenn man immer nur mit dem ersten Literal der jeweiligen Anfrage resolviert), dann ergibt sich der folgende neue Baum.



◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ↺ 🔍 ↻ 46/62

- Der Indeterminismus 2. Art kann durchaus das Terminierungsverhalten von Logikprogrammen beeinflussen.

BEISPIEL

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

Die Anfrage

$?- q(b), p.$

terminiert in Prolog, da es von $(\{\neg q(b), \neg p\}, \emptyset)$ aus keine kanonischen Berechnungsschritte gibt. Der SLD-Baum besteht daher nur aus dem einzigen Knoten, der mit der Anfrage markiert ist. Hingegen existiert eine unendliche nicht-kanonische Berechnung

$$(\{\neg q(b), \neg p\}, \emptyset) \vdash_p (\{\neg q(b), \neg p\}, \emptyset) \vdash_p \dots$$

In einer Programmiersprache, die stets das rechteste Literal der Anfrage zur Resolution selektiert, würde diese Anfrage also nicht terminieren.



Definition

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

- *Falls $G = \{\neg A_1, \dots, \neg A_k\}$ ist, so ist die Wurzel des Baums mit A_1, \dots, A_k markiert.*

Definition

- Sei nun ein Knoten mit B_1, \dots, B_n markiert und sei B_1 mit den positiven Literalen von k Programmklauseln K_1, \dots, K_k unifizierbar, wobei die Klauseln in dieser Reihenfolge im Programm auftreten. Dann hat der Knoten k Nachfolger. Der i -te Nachfolger ist mit den Atomen markiert, die sich bei einem kanonischen Berechnungsschritt durch Resolution mit der Klausel K_i ergeben. Falls diese Berechnung also die Gestalt

$$(\{\neg B_1, \dots, \neg B_n\}, \emptyset) \vdash_{\mathcal{P}} (\{\neg C_1, \dots, \neg C_m\}, \sigma)$$

hat, so ist der i -te Nachfolgerknoten mit C_1, \dots, C_m markiert und die Kante zu diesem Knoten ist mit der Substitution σ eingeschränkt auf die Variablen in B_1, \dots, B_n markiert.

- Hierbei werden Berechnungsschritte, die durch Resolution mit Hilfe des gleichen Literals und der gleichen Programmklausel entstanden sind und sich nur durch Variablenumbenennungen unterscheiden, natürlich nicht unterschieden.
- Falls B_1 also mit den positiven Literalen von k Programmklauseln unifizierbar ist, so hat der Knoten mit der Markierung B_1, \dots, B_n genau k Nachfolger.

- Die Antwortsubstitutionen lassen sich nun aus den Pfaden ablesen, die mit einem Blatt \square enden.
- Wenn die Kanten von der Wurzel zu einem \square -Blatt mit $\delta_1, \dots, \delta_L$ beschriftet sind, so ergibt sich die Antwortsubstitution $\delta_l \circ \dots \circ \delta_1$, eingeschränkt auf die Variablen aus der ursprünglichen Anfrage G .

- 



- In der Definition des SLDBaums haben wir nicht nur den Indeterminismus 2. Art aufgelöst, sondern wir haben zur Behandlung des Indeterminismus 1. Art auch die Reihenfolge der Klauseln im Programm berücksichtigt.
- Die Reihenfolge der Kinder eines Knotens entspricht also jetzt der Reihenfolge der Klauseln im Programm.

AUSWERTUNGSSTRATEGIE

- Eine **Auswertungsstrategie** ist ein Verfahren, das angibt, wie ein SLDBaum zu durchlaufen (bzw. aufzubauen) ist.
- Solch eine Strategie löst dann den Indeterminismus 1. Art auf.
- Hierbei kann man noch danach unterscheiden, ob man nur an einer erfolgreichen Berechnung interessiert ist (dann stoppt man das Verfahren, sobald man einmal \square gefunden hat) oder ob man an allen erfolgreichen Berechnungen bzw. Antwortsubstitutionen interessiert ist.
- z.B. Prolog!

- In Prolog verwendet man als Auswertungsstrategie die **Tiefensuche**.

BEISPIEL

```
mutterVon(renate, susanne).
```

```
mutterVon(susanne, aline).
```

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

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

und die Anfrage

```
?- vorfahre(Z,aline).
```

BEISPIEL

Diesmal ersetzen wir jedoch in den Programmklauseln

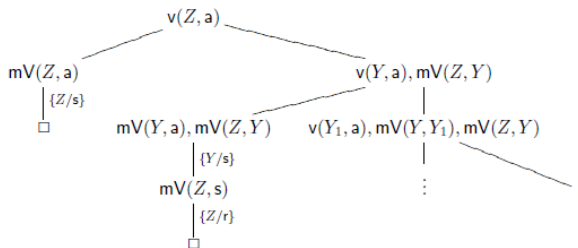
```
vorfahre(V,X) :- mutterVon(V,X).  
vorfahre(V,X) :- mutterVon(V,Y), vorfahre(Y,X).
```

die zweite Klausel durch folgende Modifikation, in der wir die beiden Literale im Rumpf vertauschen.

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

BEISPIEL

Es ergibt sich der folgende SLD-Baum:



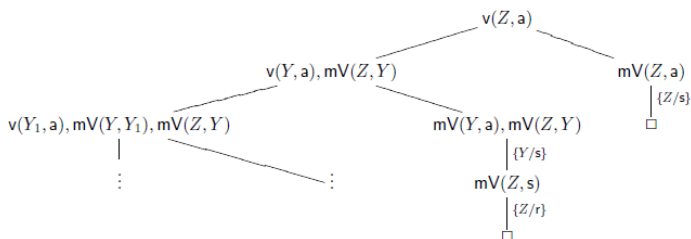
Der rechteste Pfad in diesem Baum ist nun unendlich. Falls man also eine Tiefensuche wählt, die immer von rechts nach links vorgeht, dann findet diese Strategie in diesem Beispiel keine Lösung. Die Tiefensuche, die in *Prolog* gewählt wird, geht allerdings von links nach rechts vor. Sie würde daher die ersten beiden Lösungen finden. Wenn man danach nach einer weiteren Lösung sucht, würde sie nicht mehr terminieren.

- Das obige Beispiel illustriert auch noch einmal den Effekt des Indeterminismus 2. Art auf die Terminierung: Die Vertauschung von Literalen im Rumpf einer Programmklausel kann einen Einfluss auf die Unendlichkeit des SLDBaums haben und damit auch auf die Terminierung der Auswertung.
- Das folgende Beispiel zeigt den Einfluss des Indeterminismus 1. Art: wir vertauschen die Reihenfolge der Klauseln.

BEISPIEL

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

Es ergibt sich somit der folgende SLD-Baum.



Nun würde die Tiefensuche in *Prolog* nicht terminieren und keine Lösung finden.

Das Beispiel illustriert also, dass man bei der Auswertungsstrategie von Prolog stets Fakten (bzw. nicht-rekursive) Klauseln vor den rekursiven Regeln desselben Prädikats anordnen sollte.

Zusammenfassend ergibt sich also:

- In Prolog werden Literale in Anfragen *von links nach rechts* bearbeitet. Diese Auflösung des Indeterminismus 2. Art beeinflusst zwar das Terminierungsverhalten, aber nicht die Vollständigkeit des SLD-Baums.
- In Prolog werden Programmklauseln *von oben nach unten* abgearbeitet. Dies entspricht einem Durchlauf durch den SLD-Baum in Tiefensuche, wobei linke Kinder stets zuerst betrachtet werden.