Logikprogrammierung

Prof. Dr. M. Hanus
Lehr- und Forschungsgebiet Informatik II
Rheinisch-Westfälische Technische Hochschule Aachen
Ahornstraße 55, 52072 Aachen
WWW:
`http://www-users.informatik.rwth-aachen.de/~leucker/lp/index.html`

SS 1995

`h.g.esser@gmx.de`
WWW: `http://home.pages.de/~hge/`

Inhaltsverzeichnis

1 Elementare Einführung in Prolog **3**

2 Objekte in Prolog **7**
2.1 Zahlen, Atome und Strukturen . . . 7
2.2 Listen . . . 7
2.3 Operatoren . . . 8
2.4 Variablen . . . 9
2.5 Gleichheit von Termen . . . 9

3 Beweisen mit dem Resolutionsprinzip **10**
3.1 Einfaches Resolutionsprinzip . . . 10
3.2 Unifikation . . . 11
3.3 Allgemeines Resolutionsprinzip . . . 14
3.4 Gleichheit in Prolog . . . 15

4 Elementare Programmiertechniken **15**
4.1 Aufzählung des Suchbaums . . . 15
4.2 Musterorientierte Wissensrepräsentation . . . 17
4.3 Verwendung von Relationen . . . 19
4.4 Datenstrukturen als Fakten . . . 20

5 Theoretische Grundlagen der Logikprogrammierung **22**
5.1 Aufbau logischer Programme . . . 22
5.2 Interpretationen und Modelle für Formeln . . . 25
5.3 Fixpunkte . . . 31
5.4 Deklarative Semantik logischer Programme . . . 33
5.5 Beweisen mit logischen Programmen . . . 35

5.6 Beweisen in Prolog 43

6 Nichtdeklarative Bestandteile von Prolog 44
6.1 Beweisstrategie 44
6.2 Der „Cut“-Operator 46
6.3 Negation 47
6.4 Zyklische Strukturen 51
6.5 Arithmetik 52
6.6 Ein-/Ausgabe von Daten 53
6.7 Zerlegung und Konstruktion von Termen 55
6.8 Daten als Programme 55
6.9 Programme als Daten 56

7 Fortgeschrittene Programmiertechniken 57
7.1 Differenzlisten 57
7.2 Meta-Interpretierer 59
7.3 Debugging von Prolog-Programmen 60

8 Erweiterungen der reinen Logikprogrammierung 61
8.1 Flexible Berechnungsregeln 61
8.2 Constraints 62

9 Implementierungen von Prolog 65
9.1 Grundideen der Warren Abstract Machine 66
9.2 Die Warren Abstract Machine (WAM) 70
9.2.1 Speicherbereiche und Register der WAM 70
9.2.2 Instruktionssatz 72
9.3 Globale Analyse 75

Zu diesem Skript

Das vorliegende Skript wurde von mir auf der Grundlage einer Vorlesungsmitschrift aus dem Sommersemester 1995 erstellt. Dies ist kein offizielles Skript des Lehrstuhls. Zwar habe ich mich bei der Erstellung bemüht, Fehler meiner handschriftlichen Mitschrift zu korrigieren, kann aber natürlich keine Garantie für die Korrektheit übernehmen.

Dieses Skript darf nicht kommerziell, wohl aber privat und unentgeltlich weiterverbreitet werden (sofern nicht das Urheberrecht von Prof. Hanus dies verbietet), für Anregungen und Fehlerhinweise an meine Email-Adresse `h.g.esser@gmx.de` wäre ich dankbar.

Außer dem Skript zur Vorlesung Logikprogrammierung habe ich auch Skripte zur Funktionalen Programmierung aus dem Wintersemester 1995/96 von Prof. Indermark und zur Vorlesung Approximationstheorie I aus dem Wintersemester 1995/96 von Prof. Stens erstellt.

Zu dieser Vorlesung

Die in der Vorlesung Logikprogrammierung verwendete Sprache ist *Prolog*. Gründe dafür sind, daß Prolog weit verbreitet und ein de facto-Standard für Logiksprachen ist und es viele Implementierungen gibt (siehe dazu das FAQ in der Newsgroup `comp.lang.prolog`).
Nachteil: es gibt wenig Konzepte für große Systeme.

Entstehung

1965:	Robinsons *Resolutionsprinzip* (Grundlage zum Arbeiten)
1972-73:	Colmeraner, Roussel (Marseille): Effiziente Implementierung der Resolution $\rightsquigarrow$ Prolog
1974:	Kowalski: Predicate Logic as Programming Language $\rightsquigarrow$ Rechnen = Beweisen
1977:	Pereira/Warren: DEC-10-Prolog (Übersetzer in MC-Code)

Anwendungen: Expertensysteme, Verarbeitung natürlicher Sprache, Datenbank, Übersetzerbau, Symbolische Mathematik

1981: Prolog als Implementierungssprache von Japans fünfter Computergeneration (5th generation project)

Eigenschaften:

+ Einfache Syntax
+ Klares Grundkonzept
+ Einfache Semantik (mathematische Strukturen)
+ Prototyping, Symbolverarbeitung einfach
– Programmausführung häufig nicht so effizient
– Speicherintensiv
– Noch kein Standard für Module und Typen

1 Elementare Einführung in Prolog

Prinzip der Logikprogrammierung

- Formulierung der logischen Zusammenhänge des Problems,
- Lösungsfindung ist Aufgabe des Rechners.

Programmierer der Logikprogrammierung beschreiben nicht, *wie* das Problem gelöst wird, sondern *was* das Problem ist.

Eingabe:
– Wissen über das Problem.
– Anfrage: „Ist die Aussage richtig?“ bzw. „Für welche Werte ist eine Aussage richtig?“
Ausgabe: yes/no bzw. Werte für Variablen
Wissen:

1. Eigenschaften und Beziehungen:
   „Rot ist eine Farbe“, „16 ist größer als 5“

2. Regeln zur Ableitung neuer Tatsachen:
   „Wenn 16 größer als 5 ist und 5 größer als 2, dann ist auch 16 größer als 2.“

Beispiel: Verwandtschaftsbeziehungen
Ziel: „Wer ist die Mutter von Monika?“, „Welche Großväter hat Andreas?“

Lösung 1: Speicherung sämtlicher Beziehungen.
Nachteil: Aufwendig, unflexibel bei Änderungen, fehleranfällig.

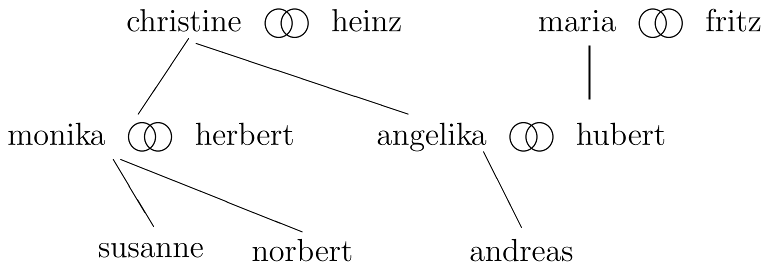
Lösung 2: Speichere nur elementare Beziehungen, die übrigen werden durch Regeln definiert.

Elementare Beziehungen:

1. männlich / weiblich

2. Wer ist mit wem verheiratet?

3. Wer ist die Mutter?



Ableitbar: angelika ist Tante von susanne.

Berechnungsprinzip von Prolog:
Eingabe: Tatsachen und Regeln. Prolog-System sucht Antworten durch Anwendung von Tatsachen und Regeln.

Prolog-Syntax
Aussagen:

- „3 ist ungerade"
  (Objekt: 3, Art der Aussage/Name: „ist ungerade")
- „monika ist mit herbert verheiratet"
  (Objekte: monika, herbert, Art der Aussage/Name: verheiratet)

In Prolog: $name(objekt_1, ..., objekt_n)$

```
ungerade (3)
verheiratet (monika, herbert)
```

Anmerkungen:

1. Alle Namen werden klein geschrieben.
2. Die Reihenfolge der Objekte ist relevant.
3. Andere Schreibweisen: → Operatoren (später)

Solche Aussagen heißen *Fakten.*

```
weiblich (christine).
weiblich (maria).
weiblich (monika).
weiblich (angelika).
weiblich (susanne).
maennlich (heinz).
maennlich (fritz).
maennlich (herbert).
maennlich (hubert).
maennlich (norbert).
maennlich (andreas).
verheiratet (christine,heinz).
...
istMutterVon (christine,herbert).
...
```

Regeln in Prolog

```
Aussage3 :- Aussage1, Aussage2.
```

Beispiel:

```
istVaterVon (herbert,susanne) :-
      verheiratet (monika,herbert),
      istMutterVon (monika,susanne).
```

Weitere Regel: dieselbe Regel mit `maria` statt `monika`.

Ausweg: statt unendlich vieler Regeln: *Variablen* (unbekannte Objekte) → beginnen mit Großbuchstaben

⇒ `istVaterVon(V,K) :- verheiratet(M,V), istMutterVon(M,K).`
entspricht unendlich vielen konkreten Regeln.

```
istGrossvater(G,E) :- istVaterVon(G,V), istVaterVon(V,E).
```

Anfragen:

- `?- verheiratet (maria,fritz).`
  ⇝ `yes`

- `?- verheiratet (monika,M).`
  ⇝ `M = herbert`

- `?- istGrossvater(heinz,E).`
  ⇝ `E = susanne` ;
  `E = norbert` ;
  `no`

Klauseln: Fakten/Regeln für Eigenschaften/Beziehungen (*Prädikate* und *Relationen*) → relationale Programmierung

Anfrage: Frage, ob Aussagen wahr/falsch sind bezüglich des vorhandenen Wissens (!)

Soweit: Prolog $\subseteq \underbrace{\text{Prädikatenlogik erster Stufe}}_{\text{unentscheidbar}}$
⇒ Auch Prolog-Programme terminieren nicht in jedem Fall.
Beispiel: Interpreter für Pascal: möglich in Prolog (Endlosschleife ausführen)

Vergleich Prolog / imperative Sprachen:

Prolog	imperative Sprachen
Problemwissen angeben	Operationen (Algorithmus) angeben
Rechner sucht Lösung	Rechner arbeitet Operationen ab
Logisches Schließen sollte bekannt sein	Aufbau des von-Neumann-Rechners sollte bekannt sein, ebenso Semantik der Sprache
Variablen in Prolog = unbekannte Objekte	Variablen = Speicherzellen
Rechnen = Beweisen	Rechnen = Manipulation von Speicherzellen

Aber: beide Sprachtypen haben gleiche Mächtigkeit.

2 Objekte in Prolog

„Wissen“ entspricht Aussagen über *abstrakte* Objekte: Aus der realen Person „Susanne“ wird durch Abstraktion das Objekt `susanne` oder `(susanne,schmidt,geb.3.4.60)`.

2.1 Zahlen, Atome und Strukturen

Wir unterscheiden vier Kategorien von Zeichen:

- $GBs := \{\ A, B, ..., Z\ \}$,
- $KBs := \{\ a, b, ..., z\ \}$,
- $Zif := \{\ 0, 1, ..., 9\ \}$,
- $SZ := \{\ +, -, *, /, <, =, >, `, \backslash, :, ., ?, @, \#, \$, \&, \hat{}, \tilde{}\ \}$.

Zahlen: Folgen von Ziffern (auch Gleitkommazahlen)
Atome:

- Folge von KBs, GBs, Zif und '`_`', die mit KBs beginnt (z.B. `susanne_schmidt`)
- Folge von SZ (z.B. `:-`, `==>`)
- beliebige Zeichenfolge in einfachen Anführungszeichen (z.B. `'haLL?o'`)
- Atome mit besonderer Bedeutung: `,` `;` `!` `[]`

Konstanten: Zahlen oder Atome

Beispiel: Relation `geboren`: Fakten der Form `geboren (fritz,1,6,27)`.
Beobachtung: 1,6,27 sind drei einzelne Objekte, sie sollten aber besser ein Objekt bilden.

Strukturen: Zusammenfassung von mehreren Objekten zu einem.
Darstellung: `datum(1,6,27)` mit *Funktor* `datum` und *Komponenten* 1,6,27.
→ `geboren (fritz, datum(1,6,27))` (zweistellige Relation)

Der Funktor ist dabei relevant: `datum(9,6,35)` ≠ `zeit(9,6,35)`. Auch geschachtelte Strukturen sind möglich, z.B. `person(susanne,schmidt,datum(3,4,60))`.

2.2 Listen

Definition (Listen)
Listen werden induktiv über die folgenden beiden Fälle definiert:

- Die *leere Liste* `[]` ist eine Liste.

- Wenn L eine Liste ist und E beliebig, dann ist `.(E,L)` eine Liste.

Beispiel: Liste mit Elementen a,b,c: `.(a,.(b,.(c,[])))`

Notationen:

- `[E1,E2,...,En]` = Liste mit den Elementen $E_1, ..., E_n$
- `[E|L]` = Liste mit Anfang E und Rest L (=`.(E,L)`)

Beispiel: `[a,b,c] = [a|[b,c]] = [a|[b|[c|[]]]]`

Texte: Liste von ASCII-Werten. Notation: "Prolog" → `[80,114,111,108,111,103]`

2.3 Operatoren

„Summe der Zahl 1 und Produkt aus 3 und 4": `+(1,*(3,4))`. Natürliche Schreibweise: $1 + 3 * 4$ (mit Infixoperatoren $+, *$)

Operatorschreibweise:

1. Strukturen mit einer Komponente:
   (a) *Präfixoperator*: `-2` steht für $-(2)$,
   (b) *Postfixoperator*: `2 fak fak` steht für $fak(fak(2))$
2. Strukturen mit zwei Komponenten:
   Infixoperator: `2+3` ist $+(2, 3)$

Problem: Eindeutigkeit
(a) `1-2-3` steht für $-(-(1, 2), 3)$ (*linksassoziativ*) oder $-(1, -(2, 3))$ (*rechtsassoziativ*),
(b) `12/6+1` steht für $+(/(12, 6), 1)$ ($/$ hat kleinere *Präzedenz* als $+$) oder $/(12, +(6, 1))$ (andersrum).

Behebung:

1. Der Prolog-Benutzer kann Operatoren mit *Assoziativität* und *Präzedenz* definieren,
2. Übliche mathematische Operatoren $(+, -, *, /)$ sind vordefiniert,
3. Das Setzen von *Klammern* ist immer möglich, z.B. `12/(6+1)`.

2.4 Variablen

Variablen sind unbekannte Objekte, die durch *Variablennamen* repräsentiert werden: Folgen von GBs, KBs, '_', beginnend mit GBs.

Beispiel:

- `datum(1,4,Jahr)`: alle ersten Apriltage
- `[A,B|L]`: alle Listen mit mindestens zwei Elementen

Gleiche Variablennamen sind gleiche Objekte:

- `[E,E|L]`: Liste mit zwei identischen Anfangselementen und beliebigem Rest

Anonyme Variable: `_` für Objekte, deren Werte nicht interessieren, z.B.
`?- istGrossvaterVon(G,_)` $\leadsto G = ...$,
`istEhemann(P) :- verheiratet(_,P).`

Term: Terme sind Konstanten, Variablen und Strukturen (= Prolog-Objekte), Terme ohne Variablen heißen *Grundterm*.

Beispiel: Listenstrukturen: Prädikat `member(E,L)` wahr, falls E in L vorkommt:
Intuitive Lösung:

- E ist erstes Element von $L \rightarrow$ wahr
- E ist zweites Element von $L \rightarrow$ wahr
- ...

Bessere Lösung:

- E ist erstes Element von $L \rightarrow$ wahr
- E im Rest von $L \rightarrow$ wahr

Implementierung:

```
member(E,[E|_]).
member(E,[_|R]) :- member(E,R).
```

2.5 Gleichheit von Termen

Relevant: Suche von passenden Fakten und Regeln. Der Begriff der Gleichheit von Termen ist nicht trivial – gilt z.B. $5 = 2 + 3$?

Definition (Termgleichheit)

1. Zahlen und Atome sind nur zu sich selbst gleich.

2. Variablen sind gleich, wenn sie den gleichen Namen haben.

3. Strukturen sind gleich, wenn

   - der Funktor gleich ist,
   - sie die gleiche Anzahl von Komponenten haben,
   - die Komponenten paarweise gleich sind

$\Rightarrow$ 5 und $2+3$ sind nicht gleich. Andererseits sind $2*2$ und $*(2,2)$ gleich. (Vergleiche Indermark: „abstrakte Syntax“). $verheiratet(M, fritz)$ und $verheiratet(maria, fritz)$ sind gleich, wenn M durch $maria$ ersetzt wird $\Rightarrow$ *Unifikation*.

Zwei Gleichheitsbegriffe:

1. Termgleichheit (s.o.), in Prolog: `s==t`

2. Unifizierbarkeit, in Prolog: `s=t`

3 Beweisen mit dem Resolutionsprinzip

3.1 Einfaches Resolutionsprinzip

Literal: Aussage über Objekte: $p(O_1, ..., O_n)$ (p: Art der Aussage, O_i: Terme)

Fakten sind Literale, die immer richtig sind, z.B. `maennlich(herbert).` Vorsicht: `maennlich(maria).` (??)

$\rightarrow$ **Jedes Faktum ist eine beweisbare Aussage.**

Regeln: $L : -L_1, ..., L_n$. (L, L_i Literale, keine Negation).
Logisch: $(L_1 \wedge ... \wedge L_n \Rightarrow L) \equiv \neg L_1 \vee \neg L_2 \vee ... \neg L_n \vee L$.

Semantik: **Wenn $L : -L_1, ..., L_n$ eine Regel ist und $L_1, ..., L_n$ beweisbar sind, dann ist auch L beweisbar.** (*Modus ponens*, *Abtrennungsregel*)

Anfrage: Aussagen, die überprüft werden sollen: $? - L_1, ..., L_n$

Semantik: Sind $L_1, ...L_n$ beweisbar bezüglich der Fakten und Regeln?

Beweisen durch Umkehrung des modus ponens:

Definition (Resolutionsprinzip, einfach)
Eine Aussage L ist beweisbar, falls

- $L : -L_1, ..., L_n$ eine Regel ist und

- jedes L_i beweisbar ist.

Spezialfall: Fakten ($n = 0$).

Anwendung:
(1) verheiratet (monika,herbert).
(2) istMutterVon (monika,susanne).
(3) istVaterVon (herbert,susanne) :– verheiratet (monika,herbert), istMutterVon (monika,susanne).
?- istVaterVon(herbert,susanne).
$\vdash_{(3)}$?- verheiratet(monika,herbert), istMutterVon(monika,susanne).
$\vdash_{(1)}$?- istMutterVon(monika,susanne).
$\vdash_{(2)}$?- .

Kann eine Anfrage mit dem Resolutionsprinzip auf eine leere Anfrage reduziert werden, dann ist sie beweisbar.

Aspekte:

1. *Nichtdeterminismus* (mehrere Literale, mehrere Klauseln)

2. *Nichtterminierung*:
   Beispiel: p(a) :- p(a).
   ?- p(a). $\vdash$?- p(a). $\vdash$?- p(a). $\vdash$ ...
   Negation: $p(a) : -\neg p(a). \iff (\neg p(a) \Rightarrow p(a)) \iff (\neg\neg p(a) \vee p(a)) \iff p(a)$
   ?- p(a). $\vdash$?- ¬p(a). $\vdash$?- p(a). $\vdash$ ...

3.2 Unifikation

Problem:
istGrossvaterVon(X,Z) :– istVaterVon(X,Y), istVaterVon(Y,Z).
?– istGrossvaterVon(heinz,E).

Semantik von Variablen: Ersetze X durch heinz, Z durch E.

Definition (Substitution)
Eine Abbildung σ, die auf den Termen definiert ist, Variablen durch Terme ersetzt und die folgenden Eigenschaften erfüllt, heißt *Substitution*:

1. Für alle $f(t_1, ..., t_n)$ gilt $\sigma(f(t_1, ..., t_n)) := f(\sigma(t_1), ..., \sigma(t_n))$ (*strukturerhaltend*, *Homomorphismus*) (analog für Literale),

2. Die Menge $\{X | X \text{ Variable mit } \sigma(X) \neq X\}$ ist endlich.

Daraus ergibt sich eine eindeutige Darstellung von Substitutionen durch

$$\{X/\sigma(X) \mid X \text{ Variable und } \sigma(X) \neq X\}$$

Beispiel: $\sigma = \{X/heinz, Z/E\}$
$\sigma(istGrossvaterVon(X, Z) = istGrossvaterVon(heinz, E)$.

Problem:
verheiratet (maria,fritz).
?– verheiratet (maria,M).
(M $\rightsquigarrow$ fritz) Das führt zur
Unifikation: Anpassung von Termen durch Variablenersetzung.

Beispiel: Unifikation ist nicht eindeutig:
$datum(Tag, Monat, 83)$ und $datum(3, M, J)$

- $\sigma_1 = \{Tag/3, Monat/4, M/4, J/83\}$
- $\sigma_2 = \{Tag/3, Monat/M, J/83\}$

σ_1 und σ_2 machen die Terme gleich, aber σ_1 ist zu speziell.

Definition (Unifikator)

- Eine Substitution σ heißt *Unifikator* für t_1 und t_2, falls $\sigma(t_1) = \sigma(t_2)$. Dann heißen t_1 und t_2 *unifizierbar*.
- Ein Unifikator σ heißt *allgemeinster Unifikator* (*mgu*, *most general unifier*), falls für jeden Unifikator σ' eine Substitution φ mit $\sigma' = \varphi \circ \sigma$ existiert.

Mgu's sind effektiv berechenbar (Robinson 1965).

Definition (Unstimmigkeitsmenge)
Seien t, t' Terme. Die *Unstimmigkeitsmenge* (*disagreement set*) $ds(t, t')$ wird definiert durch:

1. $t = t' \Rightarrow ds(t, t') = \emptyset$
2. t Variable, $t \neq t' \Rightarrow ds(t, t') = \{t, t'\}$
3. t' Variable, $t \neq t' \Rightarrow ds(t, t') = \{t, t'\}$
4. $t = f(t_1, ..., t_n), t' = g(s_1, ..., s_m)$, $n, m \geq 0$
   - Falls $f \neq g$ oder $m \neq n \Rightarrow ds(t, t') = \{t, t'\}$
   - Falls $f = g, m = n$ und $t_i = s_i$ für $i = 1, ..., k-1$ und $t_k \neq s_k \Rightarrow ds(t, t') = ds(t_k, s_k)$

Unifikationsalgorithmus
Eingabe: Terme t_0, t_1
Ausgabe: mgu σ für t_0, t_1, falls diese unifizierbar sind, ansonsten „fail“.

1. $k := 0, \sigma_0 := \{\}$

2. Falls $\sigma_k(t_0) = \sigma_k(t_1)$, dann „$\sigma_k$ ist mgu“
3. Falls $ds(\sigma_k(t_0), \sigma_k(t_1)) = \{x, t\}$ (mit x Variable, die nicht in t vorkommt), dann $\sigma_{k+1} := \{x/t\} \circ \sigma_k; k := k+1$; goto 2
   sonst „fail“

Beispiel

1. $t_0 = verheiratet(monika, M), t_1 = verheiratet(F, herbert)$
   $ds(t_0, t_1) = \{F, monika\} \Rightarrow \sigma_1 = \{F/monika\}$
   $ds(\sigma_1 t_0, \sigma_1 t_1) = \{M, herbert\} \Rightarrow \sigma_2 = \{M/herbert\} \circ \sigma_1 = \{M/herbert, F/monika\}$
   $\sigma_2 t_0 = \sigma_2 t_1 = verheiratet(monika, herbert)$
   $\Rightarrow \sigma_2$ ist mgu.
2. $t_0 = equ(f(1), g(X)), t_1 = equ(Y, Y)$
   $ds(t_0, t_1) = \{Y, f(1)\} \Rightarrow \sigma_1 = \{Y/f(1)\}$
   $ds(\sigma_1 t_0, \sigma_1 t_1) = \{g(X), f(1)\} \Rightarrow$ „fail“
3. $t_0 = X, t_1 = f(X)$
   $ds(t_0, t_1) = \{X, f(X)\} \Rightarrow$ „fail“ (da X in $f(X)$ vorkommt)

Die Abfrage „x in t?“ im Algorithmus heißt *occur check* (*Vorkommenstest*); sie kostet Zeit, ist aber in der Praxis selten erfolgreich. Daher gibt es in vielen Prolog-Systemen keinen occur check. Dies führt eventuell zu fehlerhafter Unifikation: Erzeugung *zyklischer Terme*: $X = f(X) \Rightarrow \{X/f(f(f(f(\ldots\ \}$ (Korrektheit)

Satz (Unifikationssatz von Robinson)
Seien t_0, t_1 Terme. Falls t_0 und t_1 unifizierbar sind, dann gibt der obige Algorithmus einen mgu aus, ansonsten „fail“.

Konsequenz: Unifizierbare Terme haben immer einen mgu.

Beweis:

- *Terminierung*:
  1. Schleifendurchlauf nur, falls $ds(\sigma_k t_0, \sigma_k t_1)$ mindestens eine Variable enthält.
  2. In jedem Durchlauf wird eine Variable eliminiert (d.h. in $\sigma_{k+1} t_i$ $(i = 0, 1)$ kommt x nicht vor)
  3. t_0, t_1 enthalten nur endlich viele Variablen

  Aus 1-3 folgt, daß es nur endlich viele Schleifendurchläufe geben kann, also ist der Algorithmus terminierend.

- *Korrektheit*:
  1. t_0, t_1 nicht unifizierbar: Falls der Algorithmus in Schritt 2 hielte, wären t_0, t_1 doch unifizierbar. Also muß der Algorithmus wegen der Terminierungseigenschaft in Schritt 3 halten. $\Rightarrow$ er gibt „fail“ aus.

2. t_0, t_1 unifizierbar: Sie θ ein beliebiger Unifikator für t_0, t_1. Zu zeigen ist: Für alle $k \geq 0$ gibt es eine Substitution γ_k mit $\theta = \gamma_k \circ \sigma_k$, und im k-ten Durchlauf wird nicht „fail“ ausgegeben (dann folgt wegen der Terminierung, daß die Ausgabe tatsächlich ein mgu ist).
   Beweis durch Induktion über k:
   $k = 0$: Sei $\gamma_k := \theta$. Dann: $\gamma_0 \circ \sigma_0 = \theta \circ \{\} = \theta$.
   $k \to k+1$: Induktionsvoraussetzung: $\theta = \gamma_k \circ \sigma_k$, d.h. γ_k ist ein Unifikator für $\sigma_k t_0$ und $\sigma_k t_1$.
   Entweder: $\sigma_k t_0 = \sigma_k t_1$: Dann gibt es keinen $(k+1)$-ten Schleifendurchlauf.
   Oder: $\sigma_k t_0 \neq \sigma_k t_1$. Also: $\emptyset \neq ds(\sigma_k t_0, \sigma_k t_1) = \{x, t\}$, und x nicht in t (anderenfalls $\sigma_k t_0, \sigma_k t_1$ nicht unifizierbar) $\Rightarrow$ im $(k+1)$-ten Schleifendurchlauf keine Ausgabe von „fail“ und $\sigma_{k+1} = \{x/t\} \circ \sigma_k$.
   Sei $\gamma_{k+1} := \gamma_k \setminus \{x/\gamma_k(x)\}$ (d.h. nehme Ersetzung für x aus γ_k). Dann:
   $\gamma_{k+1} \circ \sigma_{k+1} = \gamma_{k+1} \circ \{x/t\} \circ \sigma_k = \{x/\gamma_{k+1}(t)\} \circ \gamma_{k+1} \circ \sigma_k$ (da x von γ_{k+1} nicht ersetzt wird)
   $= \{x/\gamma_k(t)\} \circ \gamma_{k+1} \circ \sigma_k$ (da x nicht in t)
   $= \gamma_k \circ \sigma_k$ (Definition von γ_{k+1})
   $= \theta$ (nach Induktionsvoraussetzung)
   Also ist die Induktionsbehauptung wahr.

Komplexität: Im schlechtesten Fall exponentielle Laufzeit bezüglich der Größe der Eingabeterme (das liegt an exponentiell wachsenden Termen).

Anmerkungen:

1. Auch ohne Vorkommenstest ist die Laufzeit exponentiell, da die exponentiell wachsenden Terme aufgebaut werden müssen.

2. Ausweg: keine explizite Darstellung der Terme, sondern Graphen zur Darstellung
   $\Rightarrow$ Laufzeitverbesserung bis zu linearen (!) Algorithmen [Martelli, Montenari], [Paterson, Wegman]

3. In der Praxis sind exponentiell wachsende Terme äußerst selten $\Rightarrow$ klassische Algorithmen mit „sharing“ von Variablen sind ausreichend ($\to$ Implementierung)

3.3 Allgemeines Resolutionsprinzip

Definition (allgemeines Resolutionsprinzip)
Der Beweis der Anfrage $? - A_1, ..., A_m$ kann reduziert werden auf den Beweis der Anfrage

$$? - \sigma(A_1, ..., A_{i-1}, \quad L_1, ..., L_n, \quad A_{i+1}, ..., A_m)$$

wenn $L : -L_1, ..., L_n$ eine Regel (mit neuen Variablen) und σ ein mgu für A_i und L sind.

Freiheiten:

1. Welches Literal A_i wird im nächsten Schritt ausgewählt?
2. Welche Regel wird ausprobiert?

Genaueres dazu später (Resolution mit Auswahlregeln; Kapitel 5)

Beweisen mit Resolutionsprinzip

Programm: `istVaterVon(hans,peter).`

```
istVaterVon(peter,frank).
istOpaVon(X,Z) :- istVaterVon(X,Y), istVaterVon(Y,Z).
```

Beweis:
`?- istOpaVon (hans,E).`
$\vdash$ (Resolutionsschritt mit mgu $\sigma_1 = \{X/hans, Z/E\}$)
`?- istVaterVon (hans,Y), istVaterVon (Y,E).`
$\vdash$ ($\sigma_2 = \{Y/peter\} \circ \sigma_1$)
`?- istVaterVon (peter,E).`
$\vdash$ ($\sigma_3 = \{E/frank\} \circ \sigma_2$)
`?- .`
(Ausgabe: E=frank)

3.4 Gleichheit in Prolog

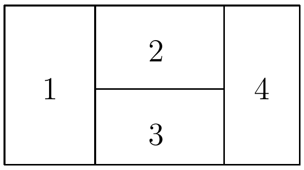
In Prolog ist für das Prädikat „=“ die Klausel `=(X,X).` vordefiniert; es gibt keine weiteren Klauseln für =. Außerdem kann = als Infixoperator verwendet werden (`X=X`). Als Konsequenz ist die Anfrage $? - t_1 = t_2$ genau dann beweisbar, wenn t_1 und t_2 unifizierbar sind.
Beispiel:

- `?- 3+2=5.` $\rightarrow$ no
- `[X|Y]=[1].` $\rightarrow X = 1, Y = []$

4 Elementare Programmiertechniken

4.1 Aufzählung des Suchbaums

Beispiel: Färben einer Landkarte



Zur Verfügung stehen vier Farben: rot, gelb, grün, blau. Problem: Die Landkarte ist so einzufärben, daß aneinandergrenzende Länder unterschiedlich gefärbt sind.

Wissen:

1. Vier Farben:
```
farbe(rot).
farbe(gelb).
farbe(gruen).
farbe(blau).
```

2. Jedes Land hat eine Farbe:
```
faerbung(L1,L2,L3,L4) :- farbe(L1),farbe(L2),farbe(L3),farbe(L4).
```

3. Ungleiche Farben:
```
verschieden(rot,gelb).
verschieden(rot,gruen).
...
```

4. Korrekte Lösungen:
```
korrekteFaerbung(L1,L2,L3,L4) :-
verschieden(L1,L2), verschieden(L1,L3), verschieden(L2,L3),
verschieden(L2,L4), verschieden(L3,L4).
```

5. Gesamtlösung:
```
?- faerbung(L1,L2,L3,L4), korrekteFaerbung(L1,L2,L3,L4).
```
liefert:
```
L1=rot, L2=gelb, L3=gruen, L4=rot
```

Analyse:

- Typisches Beispiel, ohne Algorithmus zur Lösung.
- Wissen entspricht Angabe über Lösungen.
- Aber: es gibt unendlich viele Prolog-Objekte.
- Daher: Einschränkung der Menge der potentiellen Lösungen (*Suchraum*); hier: faerbung(L1,..,L4)
- Schema:
  loesung(L) :- moeglicheLoesung(L), korrekteLoesung(L)
  Der Suchraum sollte möglichst klein sein.

- Komplexität: hier: $4^4 = 256$ mögliche Lösungen

Beispiel: Sortieren von Zahlen: `sortiere(UL,SL)` mit einer beliebigen Liste UL von Zahlen (SL sortierte Liste)

1. Wann ist eine Liste sortiert?
```
sortiert([]).
sortiert([_]).
sortiert([E1,E2|L]) :- E1=<E2, sortiert([E2|L]).
```

2. Mögliche Lösungen: SL ist eine Permutation von UL
```
perm([],[]).
perm(L1,[E2|R2]) :- streiche(E2,L1,R1), perm(R1,R2).
```

3. `streiche(E,L,R)`: $E \in L, R = L \setminus \{E\}$
```
streiche(E,[E|R],R).
streiche(E,[A|R],[A|RohneE]) :- streiche(E,R,RohneE).
```
(perm kann nicht symmetrisch definiert werden; unzureichende Auswertungsstrategie von Prolog)

4. Gesamtlösung:
```
sortiere(UL,SL) :- perm(UL,SL), sortiert(SL).
```

5. Anfrage:
```
?- sortiere([3,1,4,2],SL).
```
$SL = [1, 2, 3, 4]$

6. Komplexität: Suchraum = Menge aller Permutationen
$|UL| = n \Rightarrow |\text{Suchraum}| = n!$
Praktisch unbrauchbar $\Rightarrow$ Nutze Wissen über bessere Sortieralgorithmen.

4.2 Musterorientierte Wissensrepräsentation

Typisch für Listenverarbeitung: kleiner Suchraum, weil nur eine Klausel „paßt“. Jeder Klauselkopf hat eine spezifische Struktur.

Beispiel: `addlast(L,E,LE)` ($L = [E_1, ..., E_n], LE = [E_1, ..., E_n, E]$)

1. L ist leer: `addlast(` <u>`[]`</u>`,E,[E] )`

2. L nichtleer: `addlast(` <u>`[F|R]`</u>`,E,[F|RE]) :- addlast(R,E,RE).`

Die unterstrichenen Teilausdrücke sind *Muster* (*Pattern*), es paßt immer höchstens eines.

Beweisbeispiel:
```
?- addlast([a,b],c,[a,b,c]).
```
$\vdash$ `?- addlast([b],c,[b,c]).`

```
⊢ ?- addlast([],c,[c]).
⊢ ?- .
```

Anmerkungen:

1. Der Suchraum ist ein-elementig.
2. Beweise sind deterministisch.
3. Der Verarbeitungsdauer ist linear abhängig von der Größe der Eingabeliste.

Beispiel: Symbolisches Differenzieren $f(x) = 2x^2 + \ln x \Rightarrow \frac{df}{dx} = 4x + 1/x$

Gesucht ist ein Prädikat `dx(F,DF)`, welches wahr ist, falls $DF = \frac{d}{dx}F$, wobei F aus Zahlen, x, arithmetischen Operatoren, F^c, $\ln F$ besteht.

1. Repräsentation von Funktionen in Prolog:

Mathematische Funktion	Prolog-Term
c (Konstante)	`k(c)`
x	`x`
$f + g$ $(-, *, /)$	`F+G` $(-, *, /)$
f^c	`exp(F,k(c))`
$\ln f$	`ln(F)`
$2x^2 + \ln x$	`k(2)*exp(x,k(2))+ln(x)`

2. Wissen über `dx`: Formelsammlung

Mathematische Formel	Prolog-Klausel
$(d/dx)c = 0$	`dx(k(C),k(0)).`
$(d/dx)x = 1$	`dx(x,k(1)).`
$(d/dx)(f+g) = f' + g'$	`dx(F+G,DF+DG) :- dx(F,DF), dx(G,DG).`
$(d/dx)(cf) = c(d/dx)f$	`dx(k(C)*F,k(C)*DF) :- dx(F,DF).`
$(d/dx)(fg) = f'g + fg'$	`dx(F*G,DF*G+F*DG) :- dx(F,DF),dx(G,DG).`
$(d/dx)(f^c) = cf^{c-1}f'$	`dx(exp(F,k(C)),k(C)*exp(F,k(C-1))*DF)` `:- dx(F,DF).`
$(d/dx)(\ln f) = (1/f)f'$	`dx(ln(F),(1/F)*DF) :- dx(F,DF).`

- In linken Seiten: Muster möglichst genau spezifizieren $\Rightarrow$ kleiner Suchraum
  Alternativ für `dx(F+G,...)`:
  `dx(U,DU) :- U=F+G, DU=DF+DG, dx(F,DF), dx(G,DG).`
  (prinzipiell immer anwendbar)

- Größe der Terme unbekannt
  $\Rightarrow$ endlich viele Fakten nicht ausreichend
  $\Rightarrow$ Regeln der Form $p(...) : -...q_i(...)$
  (als Argument von p: teilspezifizierter Term, z.B. $(F + G)$; als Argumente der

q_i Unterterme, z.B. F)
⇒ Beweis durch Zerlegen des Problems und Konstruktion einer Gesamtlösung.

Beispiel: Es gibt keine Klausel für `dx(x*ln(x),...)`. Allerdings gibt es eine Klausel für $F * G$; Anwendung mit Beweis für `dx(x,...)` und `dx(ln(x),...)`.

- Symbolisches Differenzieren entspricht der Übersetzung von Funktionstermen in neue Funktionsterme und ist recht einfach spezifizierbar.
  ⇒ Prolog ist sehr gut geeignet für Übersetzer-Implementierungen.
  Spezielle Methode fest eingebaut: Definite Clause Grammars for Language Analysis (Pereira/Warren) in: Artificial Intelligence, Vol. 13 (1980) 231-278; oder: Hanus, Kap. 9
- Beachte: Prolog definiert Prädikate und keine Funktionen:
  `dx(x*ln(x),F)` $\leadsto$ erste Ableitung
  `dx(F,k(1)...)` $\leadsto$ Integral

4.3 Verwendung von Relationen

Häufig: Probleme sind funktionaler Natur: n Eingaben, berechne

$$f : M_1 \times ... \times M_n \to M : (x_1, ..., x_n) \mapsto y$$

Implementierung als Relation: `f(X1,...,Xn,Y)` genau dann erfüllt, wenn Y das Ergebnis bezüglich Eingabe $X_1, ..., X_n$ ist.

Verwendung:

1. als Funktion: $f(t_1, ..., t_n, Y) \leadsto Y = ...$
2. als Relation: $f(X_1, ..., X_n, t) \leadsto X_1 = ..., X_n = ...$
   Umkehrfunktion/ Umkehrrelation

Beispiel: Konkatenation von Listen
append$(L_1, L_2, L_3) \iff L_1 = [a_1, ..., a_n], L_2 = [b_1, ..., b_m], L_3 = [a_1, ..., a_n, b_1, ..., b_m]$
Definition über Struktur des ersten Arguments:
```
append([],L,L).
append([E|R],L,[E|RL]) :- append(R,L,RL).
```
Funktionaler Gebrauch:
```
?- append([a,f],[f,e],X).
X = [a,f,f,e]
```
Relationaler Gebrauch:
```
?- append([1],[2,3],[1,2,3]).
yes.
```
Umkehrrelation:
```
?- append(X,[3,4],[1,2,3,4]).
X = [1,2].
```

```
?- append([1,2],Y,[1,2,3,4]).
Y = [3,4].
?- append(X,Y,[a,b,c]).
X = [], Y = [a,b,c]  ;
X = [a], Y = [b,c]  ;
X = [a,b], Y = [c]  ;
X = [a,b,c], Y = []  ;
no.
```

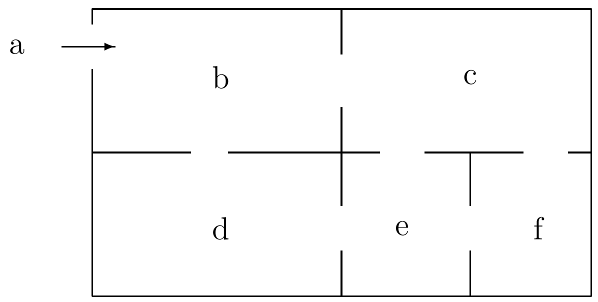
Verwendung zur Definition neuer Prädikate:

- Anhängen eines Elements an eine Liste:
  `anhang(L,E,LE) :- append(L,[E],LE).`
- Letztes Element einer Liste:
  `last(L,E) :- append(_,[E],L).`
- Element enthalten?
  `member(E,L) :- append(_,[E|_],L).`
- Streichen eines Elementes:
  ```
  streiche(L,E,LohneE)
    :- append(L1,[E|L2],L), append(L1,L2,LohneE).
  ```
- Teilliste:
  `teilliste(T,L) :- append(L1,R,L), append(T,L2,R).`
  $(L = L_1; \underbrace{T; L_2}_{R})$

1. Denke in Relationen (Beziehungen) und nicht in Zuordnungen.
2. Alle Parameter sind gleichberechtigt (keine Ein- oder Ausgabeparameter).
3. Nutze vorhandene Prädikate. Achte auf Allgemeinheit bei der Definition neuer Prädikate.

4.4 Datenstrukturen als Fakten

Beispiel: Wegesuche im Labyrinth



Frage: Gibt es einen Weg von a nach f? Allgemein: Weg von X nach Y.
Lösung:

1. $X = Y$?
2. X hat Tür zu Z, und es gibt einen Weg von Z nach Y

⇒ Labyrinth muß als Objekt bekannt sein. Darstellung?

1. Erste Möglichkeit: Labyrinth = Liste aller Türen:
   ```
   [tuer(a,b), tuer(b,c), ..., tuer(e,f)]
   ```
   Lösungsfindung mittels Durchsuchen der Liste
2. Einfache Lösung: Labyrinth = Wissen über Türen:
   ```
   tuer(a,b).
   tuer(b,c).
   ...
   tuer(e,f).
   ```
   „Einseitige" Formulierung. Eine Klausel der Form `tuer(X,Y) :- tuer(Y,X).` würde zur Gefahr von Endlosbeweisen führen. Vermeidung:
   - zusätzliche Fakten oder
   - hier: Berücksichtigung der Asymmetrie in Klauseln

   `gehe(X,Y)`; Weg von X nach Y
   Klauseln:
   ```
   gehe(X,X).
   gehe(X,Y) :- tuer(X,Z), gehe(Z,Y).
   gehe(X,Y) :- tuer(Z,X), gehe(Z,Y). (Asymmetrie)
   ```
   Dies ist logisch korrekt, allerdings besteht die Gefahr von Endlosbeweisen durch endlose Wege, z.B. a-b-c-e-d-b-c-e-d-b-c-e-...
   Zusätzlich: Information über besuchte Räume:
   ```
   gehe(X,X,Besucht).
   gehe(X,Y,Besucht) :- tuer(X,Z), nichtEnthalten(Z,Besucht),
                        gehe(Z,Y,[Z|Besucht]).
   gehe(X,Y,Besucht) :- tuer(Z,X), nichtEnthalten(Z,Besucht),
                        gehe(Z,Y,[Z|Besucht]). (Asymmetrie)
   nichtEnthalten(E,[]).
   nichtEnthalten(E,[K|R]) :- ungleich(E,K), nichtEnhtalten(E,R).
   ungleich(a,b).
   ungleich(b,a).
   ...
   ```

Fragen:

- Weg von a nach f?
  ```
  ?- gehe(a,f,[a]).
  ```
  $\rightsquigarrow$ `yes.`

- Weg von a nach f, der nicht durch c oder e führt?
  `?- gehe(a,f,[a,c,e]).`
  $\leadsto$ `no.`

Merke:

1. Es kann günstiger sein, Daten als Wissen darzustellen.
2. Problematisch ist die Änderung der Daten zur Laufzeit.
3. Typisch: Datenbankimplementierung (vergleiche Verwandtschaft)

5 Theoretische Grundlagen der Logikprogrammierung

Semantik von Prolog ≈ Beweisen von Aussagen.

1. Was heißt „konkreter“ Beweis?
2. Endlosbeweise: Kann Prolog alles beweisen?
3. Negation: Warum ist dies ein Problem?

Weitere Struktur:

- Syntax und Semantik logischer Programme (≠ Prolog-Programme)
- Vergleich mit Prolog

5.1 Aufbau logischer Programme

Hier: Formeln der *Prädikatenlogik erster Stufe.*

Definition (Prädikat, Term, Stelligkeit)
Ein *Prädikat* hat die Form

$$p(t_1, ..., t_n), \quad n \geq 0$$

(p Prädikatname, $t_1, ..., t_n$ Terme)
Ein *Term* ist

- eine Variable (Großbuchstaben)
- eine Konstante (z.B. $a, b, c, ...$)
- eine Struktur $f(t_1, ..., t_n),\ n \geq 1,\ t_i$ Terme

Die *Stelligkeit* von Prädikaten/Funktoren ist die Anzahl ihrer Argumente.

Beispiel: $p(f(a,b),X,Y)$ ist ein dreistelliges Prädikat, dessen erstes Argument eine zweistellige Struktur und dessen zweites und drittes Argument Variablen sind.

Definition (Formel)

1. Ein Prädikat ist eine Formel.
2. Wenn F und G Formeln sind, dann auch
   - $(\neg F)$: *Negation* „nicht“
   - $(F \wedge G)$: *Konjunktion* „und“
   - $(F \vee G)$: *Disjunktion* „oder“
   - $(F \Rightarrow G)$: *Implikation* „folgt“
3. Wenn F eine Formel und X eine Variable sind, dann sind auch
   - $(\exists X : F)$ $\quad$ (*Existenzquantor*)
   - $(\forall X : F)$ $\quad$ (*Allquantor*)

   Formeln.

(Dies beschreibt die Sprache der *Prädikatenlogik erster Stufe*:
„Erste Stufe“: Quantifizierung über Objekte,
„Zweite Stufe“: Quantifizierung über Prädikate.

Beispiel:

- $(\forall X : (\exists Y : (p(f(X,Y)) \Rightarrow q(X,Y))))$
- $(\forall X : ((p(X,a) \wedge (\neg p(X,a))) \Rightarrow q(X,a)))$

Vereinbarung: Klammern können weggelassen werden, solange die Eindeutigkeit klar ist:

- $\forall X : \exists Y : p(f(X,Y)) \Rightarrow q(X,Y)$
- $\forall X : p(X,a) \wedge \neg p(X,a) \Rightarrow q(X,a)$

Definition (gebunden, frei, geschlossen)
Eine Variable X heißt *quantifiziert* oder *gebunden* in F genau dann, wenn F eine Teilformel $\exists X : G$ oder $\forall X : G$ enthält und X außerhalb dieser Teilformel nicht vorkommt. Anderenfalls heißt X *frei* in F.
Eine *geschlossene Formel* ist eine Formel ohne freie Variablen.

Definition (Literal, Klausel)
Ein *Literal* ist ein Prädikat (positiv) oder ein negiertes Prädikat (negativ). Eine *Klausel* ist eine geschlossene Formel der Form

$$\forall X_1 : ...\forall X_k : (L_1 \vee L_2 \vee ... \vee L_n)$$

(mit Literalen L_i).

Beispiel: $\forall X : \forall Y : \neg p(f(X, Y)) \vee q(X, Y)$

Notation: Der Ausdruck

$$A_1, A_2, ..., A_m \Rightarrow B_1, ..., B_n \qquad (n, m \geq 0)$$

mit Prädikaten A_i, B_j steht für

$$\forall X_1 : ...\forall X_k : \neg A_1 \vee ... \vee \neg A_m \vee B_1 \vee ... \vee B_n$$

(wobei $X_1, ..., X_k$ alle Variablen in A_i, B_j sind).

Mathematische Gesetze:

- $\neg A \vee \neg B \equiv \neg(A \wedge B)$ (de Morgan)
- $\neg A \vee B \equiv A \Rightarrow B$

Beachte: In $A_1, ..., A_m \Rightarrow B_1, ..., B_n$ stehen die vorderen Kommata für „und“, die hinteren Kommata für „oder“.

Definition (Programmklausel, logisches Programm)
Eine *Programmklausel* hat die Form

$$\underbrace{A_1, ..., A_m}_{\text{Rumpf}} \Rightarrow B \qquad (m \geq 0),$$

mit *Kopf* B, für $m = 0$ heißt $\Rightarrow B$ ein *Faktum*.
Ein *logisches Programm* ist eine Menge von Programmklauseln, eine *Anfrage* hat die Form

$$A_1, ..., A_m \Rightarrow$$

Die *leere Klausel* schreiben wir $\Rightarrow$, und unter einer *Hornklausel* verstehen wir eine Programmklausel oder eine Anfrage. → Logikprogrammierung basiert auf Hornklausellogik.

Beispiel:
$\Rightarrow append([], L, L)$
$append(R, L, RL) \Rightarrow append(\cdot(E, R), L, \cdot(E, RL))$
„bedeutungslose Zeichenkette“.

5.2 Interpretationen und Modelle für Formeln

$\forall X : p(f(X, a), X)$
Fragen: 1) Über welche Menge wird quantifiziert? 2) Welche Bedeutung haben p, f, a?

Definition (Interpretation)
Eine *Interpretation* $I = (U, \delta)$ von Formeln besteht aus

1. Dem *Universum*, einer nichtleeren Menge U von Objekten,
2. Einer Zuordnung von Konstanten c zu Objekten $\delta_c \in U$,
3. Einer Zuordnung von n-stelligen Funktionen f zu Abbildungen $\delta_f : U^n \to U$,
4. Einer Zuordnung von n-stelligen Prädikaten p zu Abbildungen $\delta_p : U^n \to \{w, f\}$.

Beispiel: Verschiedene Interpretation der obigen Formel:

1. $U = \mathbb{N}, \delta_a = 1, \delta_f = *, \delta_p = \ '='$
   $p(f(X, a), X) \leadsto_\delta X * 1 = X$
2. $U = \mathbb{Z}, \delta_a = 2, \delta_f = +, \delta_p = \ '>'$
   $p(f(X, a), X) \leadsto_\delta X + 2 > X$

Definition (Variablenbelegung)
Sei $I = (U, \delta)$ eine Interpretation. Eine *Variablenbelegung* φ bezüglich I ist eine Abbildung von Variablen nach U.
Notation: $\varphi = \{X_1/e_1, ..., X_n/e_n\},\ e_i \in U$.

Ausrechnen von Termen:

Definition
Sei $I = (U, \delta)$ eine Interpretation, φ eine Variablenbelegung bezüglich I. Dann wird die Fortsetzung $\bar{\varphi}$ von φ auf Terme induktiv definiert über

- $\bar{\varphi}(X) := \varphi(X)$ für alle Variablen X,
- $\bar{\varphi}(c) := \delta_c$ für alle Konstanten c,
- $\bar{\varphi}(f(t_1, ..., t_n)) := \delta_f(\bar{\varphi}(t_1), ..., \bar{\varphi}(t_n))$ für alle Strukturen.

Beispiel: Sei $\delta_f = +$.
$\varphi_1 = \{X/0, Y/1\} \Rightarrow \bar{\varphi}_1(+(X, Y)) = 1$,
$\varphi_2 = \{X/3, Y/5\} \Rightarrow \bar{\varphi}_2(+(X, Y)) = 8$.

Definition (Wahrheit von Formeln)
Seien $I = (U, \delta)$ eine Interpretation und φ eine Variablenbelegung.

1. $p(t_1, ..., t_n)$ ist *wahr* $:\iff \delta_p(\bar{\varphi}(t_1), ..., \bar{\varphi}(t_n)) = w$.

2. Für zusammengesetzte Formeln ergibt sich die Wahrheit aus der Wahrheitstafel:

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \Rightarrow G$
w	w	f	w	w	w
w	f	f	f	w	f
f	w	w	f	w	w
f	f	w	f	f	w

3. $\exists X : F$ ist wahr, falls ein $e \in U$ existiert, so daß F bezüglich I und $\varphi[X/e]$ wahr ist, wobei
$$\varphi[X/e](Y) := \begin{cases} \varphi(Y), \text{ falls } X \neq Y \\ e, \text{ falls } X = Y \end{cases}$$

4. $\forall X : F$ ist wahr, falls für jedes $e \in U$ F bezüglich I und $\varphi[X/e]$ wahr ist.

Im folgenden betrachten wir geschlossene Formeln.

Definition (Modell)
Ein *Modell* für eine Formel ist eine Interpretation, unter der die Formel wahr wird. Eine Formel heißt *erfüllbar*, falls sie ein Modell besitzt, ansonsten heißt sie *unerfüllbar*. Eine Formel heißt *allgemeingültig*, falls jede Interpretation ein Modell ist.

Beispiel:

- $\forall X : p(f(X, a), X)$ ist erfüllbar,
- $p(a) \wedge \neg p(a)$ ist unerfüllbar,
- $p(a) \vee \neg p(a)$ ist allgemeingültig.

Definition Ein *Modell* für eine endliche Menge von Formeln $\{F_1, ..., F_n\}$ ist eine Interpretation, unter der die Formel $(F_1 \wedge ... \wedge F_n)$ wahr ist (d.h. jede der Formeln F_i ist wahr).

(logische Programme: Mengen von Formeln)

Vereinbarung: $F, F_1, ..., F_n$ immer Formeln, S endliche Formelmenge.

Definition (logische Konsequenz)
F heißt *logische Konsequenz* aus S, wenn jedes Modell für S auch Modell für F ist.

Folgerung 5.1
F ist logische Konsequenz aus $\{F_1, ..., F_n\}$ genau dann, wenn $(F_1 \wedge ... \wedge F_n \Rightarrow F)$ allgemeingültig ist.

Beweis: Übung.

Satz 5.2
F ist logische Konsequenz aus S genau dann, wenn $S \cup \{\neg F\}$ unerfüllbar ist.

Beweis: „$\Rightarrow$“: Sei I eine Interpretation für $S \cup \{\neg F\}$. Annahme: I ist Modell für S (sonst trivial). Dann folgt (da F logische Konsequenz aus S): I ist Modell für F. $\Rightarrow$ $\neg F$ falsch bezüglich I. $\Rightarrow$ I ist kein Modell für $S \cup \{\neg F\}$. Da I beliebige Interpretation war, gibt es kein Modell.
„$\Rightarrow$“: Sei I ein Modell für S. Dann folgt (da $S \cup \{\neg F\}$ unerfüllbar): I ist kein Modell für $\neg F$. Also ist F wahr bezüglich I und somit I ein Modell für F.

Falls S unerfüllbar ist, ist jedes F logische Konsequenz aus S.

Konsequenz für Logikprogrammierung:
P Programm, A Anfrage. Sei $P \cup \{A\}$ unerfüllbar. $\Rightarrow$ (Satz 5.2) $\neg A$ ist logische Konsequenz aus P.
A hat die Form „$A_1, ..., A_n \Rightarrow$“,
d.h. Formel $\forall X_1 ... \forall X_k : \neg A_1 \vee ... \vee \neg A_n$
d.h. $\neg A$: $\underbrace{\exists X_1 ... \exists X_k : A_1 \wedge ... \wedge A_n}_{\text{wahr}}$
d.h. es existieren Werte für $X_1, ..., X_k$, so daß jedes A_i wahr ist.

Daher: zeige Unerfüllbarkeit von Formelmengen.

Lösung (unpraktikabel): Zeige, daß alle Interpretationen keine Modelle sind.

Im Fall von *Klausel*mengen ist es ausreichend, nur (syntaktische) Herbrand-Interpretationen zu untersuchen:

Definition (Herbrand-Basis)
Sei S eine Formelmenge. Dann bezeichnen

- $U_S :=$ Menge aller Grundterme mit Konstanten und Faktoren aus S das *Herbrand-Universum* und
- $B_S := \{p(t_1, ..., t_n) \mid t_1, ..., t_n \in U_S, p$ kommt in S vor $\}$ die *Herbrand-Basis* von S.

Beispiel: $S = \{p(X, Y) \Rightarrow p(f(X), g(Y)), \quad \Rightarrow p(f(a), X)\}$.
Dann ist $U_S = \{a, f(a), g(a), f(f(a)), f(g(a)), g(f(a)), ...\}$.

Definition (Herbrand-Interpretation)
Eine Interpretation $I = (U, \delta)$ eine Formelmenge S heißt *Herbrand-Interpretation*, falls

1. $U = U_S$ (Herbrand-Universum),
2. $\delta_c = c$ für alle Konstanten c,

3. $\delta_f(t_1, ..., t_n) = f(t_1, ..., t_n) \in U_S$ für alle n-stelligen Funktoren f und $t_1, ..., t_n \in U_S$ (*Term-Interpretation*)

Intuitiv: Herbrand-Interpretationen sind syntaktische Terminterpretationen.
Prädikate sind in Herbrand-Interpretationen nicht festgelegt.

Definition (Herbrand-Modell)
Herbrand-Interpretationen, die Modelle sind, heißen *Herbrand-Modell*.

Herbrand-Interpretationen lassen sich durch Angabe einer Teilmenge $M \subset B_S$ der Herbrand-Basis spezifizieren. Intuitiv: $L \in M$ bedeutet L ist wahr bezüglich M.

Beispiel: $M = \{p(a), p(f(f(a))\}$.
Das Universum ist $U = \{a, f(a), f(f(a)), f(f(f(a))), ...\}$.
Wahr sind: $p(a)$ und $p(f(f(a)))$.
Falsch sind: $p(f(a)), p(f^3(a)), p(f^4(a)), ...$.

Satz 5.3
Sei S eine Klauselmenge, die ein Modell hat. Dann hat S auch ein Herbrand-Modell.

Beweis: Sei I ein Modell für S. Dann definieren wir die Herbrand-Interpretation I' durch

$$I' := \{p(t_1, ..., t_n) \in B_S \mid p(t_1, ..., t_n) \text{ wahr bezüglich } I\}$$

Man kann nun recht einfach zeigen (Übung): I' ist Modell für S.

Satz 5.4
Sei S eine Klauselmenge. Dann gilt:
S unerfüllbar $\iff$ S hat kein Herbrandmodell.

Beweis: „$\Rightarrow$“: trivial. „$\Leftarrow$ “: Wäre S erfüllbar, dann gäbe es nach Satz 5.3 ein Herbrandmodell, Widerspruch.

Also: In der Logikprogrammierung: Untersuche alle Herbrand-Interpretationen (aufzählbar). $\Rightarrow$ Rein syntaktische Methoden sind ausreichend.

Anmerkung: Bei Nicht-Klauselmengen gilt Satz 5.4 nicht! (vergleiche Übung)

Konflikt:

- Zum automatischen Beweisen: Klauseln,
- Manchmal natürlicher: Problemspezifikation in Prädikatenlogik (also mit $\exists$-Quantoren)

Exkurs: Transformation von Formeln in Klauseln

Definition (logisch äquivalent)
Zwei Formeln F, G heißen *logisch äquivalent*, wenn für jede Interpretation I und jede Variablenbelegung φ gilt:

$$F \text{ wahr bezüglich } I, \varphi \iff G \text{ wahr bezüglich } I, \varphi$$

Notation: $F \equiv G$.

Beispiel: $(F \Rightarrow G) \equiv (\neg F \vee G)$

Im folgenden versuchen wir Formeln mit $\forall, \exists, \neg, \wedge$ und $\vee$ auf Klauselform

$$\forall X_1 ... \forall X_k : \neg A_1 \vee ... \neg A_m \vee B_1 \vee ... \vee B_n$$

zu transformieren.

Erster Transformationsschritt: Quantoren nach außen schieben

Definition (Pränex-Form)
Eine Formel ist in *Pränex-Form*, falls sie die Form $Q_1 X_1 ... Q_n X_n F$ hat, wobei die $Q_i \in \{\forall, \exists\}$ sind, $n \geq 0$, X_i Variablen und F eine Formel ohne Quantoren sind.

Zur Transformation einer Formel in Pränex-Form:

Algorithmus (Transformation in Pränex-Form)

1. $\neg(\forall X F) \equiv \exists X(\neg F)$
   $\neg(\exists X F) \equiv \forall X(\neg F)$

2. Falls X in G nicht frei vorkommt (sonst benenne X um durch Äquivalenz $\forall X F = \forall Y F[X/Y]$; Ersetzung aller freien Variablen X durch Y in F)
   $(\forall X F) \wedge G \equiv \forall X(F \wedge G)$
   $(\forall X F) \vee G \equiv \forall X(F \vee G)$
   $(\exists X F) \wedge G \equiv \exists X(F \wedge G)$
   $(\exists X F) \vee G \equiv \exists X(F \vee G)$

Im folgenden: Jeder Quantor bindet eine andere Variable.

Satz 5.5
Zu jeder Formel existiert eine logische äquivalente Formel in Pränexform.

Beweis: Sukzessives Anwenden der obigen Äquivalenzen (genauer: Literatur: Schöning)

Zweiter Schritt: Elimination von Existenzquantoren (Skolemisierung)

Algorithmus (Skolemisierung)
Eingabe: Formel F in Pränexform

Ausgabe: *Skolemform* von F
while (F enthält Existenzquantoren)
do
Sei $F = \forall X_1 ... \forall X_n \exists Y G$
($\exists Y$ „linkester" Existenzquantor)
Sei f ein neues n-stelliges Funktionssymbol.
$F := \forall X_1 \forall X_n G[Y/f(X_1, ..., X_n)]$
od
Ausgabe F

Im allgemeinen ist dies keine Äquivalenzumformung. Es gilt aber:

Satz 5.6
Sei F in Pränexform. Dann gilt: F ist genau dann erfüllbar, wenn die Skolemform von F erfüllbar ist.

Beweis: Schöning

Erfüllbarkeitsäquivalenz ist für viele Anwendungen ausreichend, insbesondere für *Unerfüllbarkeitstests.*

Dritter Schritt: Konjunktive Normalform

Konjunktive Normalform (*KNF*): $\bigwedge_i \bigvee_j L_{ij}$

Algorithmus (KNF)
Eingabe: quantorenfreie Formel F
Ausgabe: KNF von F

1. Führe folgende Ersetzungen durch:
   $\neg\neg G \rightarrow G$
   $\neg(G \wedge H) \rightarrow \neg G \vee \neg H$ (de Morgan)
   $\neg(G \vee H) \rightarrow \neg G \wedge \neg H$

2. Führe folgende Ersetzungen durch:
   $G \vee (H \wedge K) \rightarrow (G \vee H) \wedge (G \vee K)$
   $(F \wedge H) \vee K \rightarrow (G \vee K) \wedge (H \vee K)$

Schritte 1, 2 und 3 führen zu Formel

$$\forall X_1 ... \forall X_n : \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} L_{ij} \approx \{\forall X_1 ... \forall X_n : \bigvee_{j=1}^{m} L_{ij} \mid i = 1, ..., n\}$$

(L_{ij} Literale)

Beispiel:

$$F = (\neg \exists X (p(X, Z) \vee \forall Y q(X, f(Y))) \vee \forall Y p(q(Y, Y), Z))$$

1. Eliminiere mehrfache Vorkommen quantifizierter Variablen:
   $F_1 = (\neg\exists X(p(X,Z) \vee \forall Y\, q(X, f(Y))) \vee \forall W\, p(q(W,W),Z))$

2. Freie Variablen mit $\exists$ binden:
   $F_2 = \exists Z F_1$

3. Pränexform:
   $\exists Z \forall X \exists Y \forall W (\neg(p(X,Z) \vee q(X, f(Y))) \vee p(q(W,W),Z))$

4. Skolemform: $Z/a, Y/h(X)$:
   $\forall X \forall W (\neg p(X,a) \vee q(X, f(h(X))))) \vee p(q(W,W),a))$

5. Rumpf in KNF:
   $\forall X \forall W (\neg p(X,a) \vee p(q(W,W),a)) \wedge (\neg q(X, f(h(X))) \vee p(q(W,W),a))$

6. Klauselnotation:
   $\{p(X,a) \Rightarrow p(q(W,W),a), \quad q(X, f(h(X))) \Rightarrow p(q(W,W),a)\}$
   (sogar zufällig Hornklauseln $\Rightarrow$ Ergebnis ist logisches Programm)

Theorembeweiser basieren meist auf Klauseln (allgemeinen – nicht nur Hornklauseln).

Allerdings: Effiziente Beweisverfahren sind nur für Hornklauseln bekannt.

5.3 Fixpunkte

Motivation: Wir wollen eine Beziehung zwischen Modellen und Resolutionsverfahren hestellen.

Definition (partielle Ordnung)
Eine *partielle Ordnung* ist eine binäre Relation „$\leq$“ auf einer Menge M mit

1. $x \leq x \forall x \in M$ (*Transitivität*)

2. $x \leq y$ und $y \leq x \Rightarrow x = y \quad \forall x, y \in M$ (*Antisymmetrie*)

3. $x \leq y$ und $y \leq z \Rightarrow x \leq z \quad \forall x, y, z \in M$ (*Reflexivität*)

Beispiel: A Menge, $2^A = Pot(A)$ Potenzmenge von A. Dann ist $\subseteq$ eine partielle Ordnung auf 2^A.

Definition (Schranken)
Sei M eine Menge mit partieller Ordnung $\leq$.

- $a \in M$ heißt *obere Schranke* für $A \subseteq M$, falls $x \leq a$ für alle $x \in A$.

- $a \in M$ heißt *untere Schranke* für $A \subseteq M$, falls $a \leq x$ für alle $x \in A$.

- $a \in M$ heißt *kleinste obere Schranke* (*least upper bound*, *lub(A)*), falls a obere Schranke ist und für jede obere Schranke b gilt: $a \leq b$.

- $a \in M$ heißt *größte untere Schranke* (*greatest lower bound*, *glb(A)*), falls a untere Schranke ist und für jede untere Schranke b gilt: $b \leq a$.

Anmerkung: lub und glb sind eindeutig, falls sie existieren.

Definition (Verband)
Eine partiell geordnete Menge M heißt *vollständiger Verband*, wenn lub(A) und glb(A) für alle $A \subseteq M$ existieren. In diesem Fall heißen

- $\bot$ =glb(M) *kleinstes Element* (*bottom*),
- $\top$ =lub(M) *größtes Element* (*top*).

Beispiel: 2^A ist vollständiger Verband, wobei

$$\text{lub}(B) = \bigcup_{C \in B} C,\ \text{glb}(B) = \bigcap_{C \in B} C,\ \bot = \emptyset,\ \top = A$$

(für $B \subseteq 2^A$)

Definition
Sei M ein vollständiger Verband, $f : M \to M$ eine Abbildung.

- f heißt *monoton*, falls $f(x) \leq f(y)$ für alle $x, y \in M$.
- Eine Teilmenge $X \subseteq M$ heißt *gerichtet*, falls jede endliche Teilmenge von X eine obere Schranke in X hat.
- f heißt *stetig*, falls $f(\text{lub}(X)) = \text{lub}(f(X))$ für alle gerichteten Teilmengen $X \subseteq M$.

Interessant für die Logikprogrammierung ist die Herbrandbasis B_P. Die Menge 2^{B_P} aller Herbrand-Interpretationen ist ein vollständiger Verband.

Definition (kleinster Fixpunkt)
Sei M ein vollständiger Verband, $f : M \to M$.
$a \in M$ heißt *kleinster Fixpunkt* von f, falls

- a Fixpunkt von f, d.h. $f(a) = a$ und
- $a \leq b$ für alle übrigen Fixpunkte b von f.

Analog: *größter Fixpunkt*.

Satz 5.7 (Tarski)
Sei M ein vollständiger Verband, $f : M \to M$ monoton. Dann existieren ein kleinster Fixpunkt lfp(f) und ein größter Fixpunkt gfp(f). Außerdem:

$$\text{lfp}(f) = \text{glb}\{x \mid f(x) = x\} = \text{glb}\{x \mid f(x) \leq x\}$$

$$\mathrm{gfp}(f) = \mathrm{lub}\{x \mid f(x) = x\} = \mathrm{lub}\{x \mid x \leq f(x)\}$$

Beweis: Lloyd-Buch

Definition
Sei M ein vollständiger Verband, $f : M \to M$ monoton. Dann definiere

- $f \uparrow 0 := \bot$
- $f \uparrow k := f(f \uparrow (k-1)) \forall k > 0$
- $f \uparrow \omega := \mathrm{lub}\{f \uparrow k \mid k \geq 0\}$
  ($\omega \approx$ unendlich, $\to$ *Ordinalzahlen*)

Konstruktive Berechnung des kleinsten Fixpunktes lfp:

Satz 5.8 (Kleene)
Sei M ein vollständiger Verband, $f : M \to M$ stetig. Dann gilt:

$$\mathrm{lfp}(f) = f \uparrow \omega$$

Konsequenz: Berechne lfp durch Iteration, ausgehend von $\bot$.

5.4 Deklarative Semantik logischer Programme

Im folgenden sei P immer ein logisches Programm.

Satz 5.9

$$\bigcap\{M | M \text{ Herbrand-Modell für } P\}$$

ist auch ein Herbrand-Modell, genannt *kleinstes Herbrand-Modell M_P*.

Beweis: Die Herbrand-Basis B_P ist ein Modell für P. Es ist einfach zu zeigen ($\to$ Übung), daß der Durchschnitt einer nichtleeren Menge von Herbrand-Modellen ein Herbrand-Modell ist.

Häufig sagt man: die deklarative Semantik von P entspricht M_P. Dies wird durch den folgenden Satz gerechtfertigt:

Satz (van Emden, Kowalski)

$$M_P = \{A \in B_P | A \text{ logische Konsequenz von } P\}$$

Beweis: A logische Konsequenz von P
$\iff$ $P \cup \{\neg A\}$ unerfüllbar (Satz 5.2)
$\iff$ $P \cup \{\neg A\}$ hat kein Herbrand-Modell (Satz 5.4)
$\iff$ $\neg A$ ist falsch in allen Herbrand-Modellen für P
$\iff$ A ist wahr in allen Herbrand-Modellen für P

$\iff A \in M_P$.

Charakterisierung von M_P als kleinster Fixpunkt (Beachte: 2^{B_P} ist vollständiger Verband.)

Definition (Operator T_P)
Die Abbildung $T_P : 2^{B_P} \to 2^{B_P}$ auf Herbrand-Interpretationen sei definiert durch
$$T_P(I) := \Big\{A \in B_P \mid A_1, ..., A_n \Rightarrow A \text{ Grundinstanz einer Klausel in } B_P \text{ mit } \{A_1, ..., A_n\} \subseteq I\Big\}$$
Hierbei: t_1 heißt *Grundinstanz* von t_2, falls es eine Substitution σ mit $\sigma(t_2) = t_1$ gibt und t_1 variablenfrei ist.

Beispiel:
$\Rightarrow p(A)$
$p(X) \Rightarrow p(f(X))$
Sei $I_0 := \emptyset$.
$I_1 = T_P(I_0) = \{p(a)\}$.
$I_2 = T_P(I_1) = \{p(a), p(f(a))\}$.
$I_3 = T_P(I_2) = \{p(a), p(f(a)), p(f(f(a)))\}$.
T_P beschreibt *Ein-Schritt-Ableitungen* mit Regeln.

Klar: T_P ist monoton: $I_1 \subseteq I_2 \Rightarrow T_P(I_1) \subseteq T_P(I_2)$.

Satz 5.11: T_P ist stetig.

Beweis: Sei X eine gerichtete Teilmenge von 2^{B_P}. Zu zeigen ist: $T_P(\text{lub}(X)) = \text{lub}(T_P(X))$. Wir stellen zunächst fest: $\{A_1, ..., A_n\} \subseteq \text{lub}(X) \iff \{A_1, ..., A_n\} \subseteq I$ für ein $I \in X$ ($\to$ Übung).
$A \in T_P(\text{lub}(X))$
$\iff A_1, ..., A_n \Rightarrow A$ Grundinstanz einer Klausel aus P und $\{A_1, ..., A_n\} \subseteq \text{lub}(X)$ (Definition T_P)
$\iff A_1, ..., A_n \Rightarrow A$ Grundinstanz einer Klausel aus P und $\{A_1, ..., A_n\} \subseteq I$ für ein $I \in X$
$\iff A \in T_P(I)$ für ein $I \in X$ (Definition T_P)
$\iff A \in \text{lub}(T_P(X)) = \bigcup_{I \in X} T_P(I)$.

Satz 5.12
Sei I eine Herbrand-Interpretation von P. Dann gilt:
I ist Modell für $P \iff T_P(I) \subseteq I$.

Beweis: I Modell
$\iff$ Für alle Grundinstanzen $A_1, ..., A_n \Rightarrow A$ von Klauseln aus P gilt: $\{A_1, ..., A_n\} \subseteq I \Rightarrow A \in I$
$\iff T_P(I) \subseteq I$.

Satz 5.13 (Fixpunktcharakterisierung des kleinsten Herbrand-Modelles)

$$M_P = \mathrm{lfp}(T_P) = T_P \uparrow \omega$$

Beweis: $M_P = \bigcap\{I | I$ Herbrand-Modell für $P\}$
$= \mathrm{glb}\{I | I$ Herbrand-Modell für $P\}$
$= \mathrm{glb}\{I | T_P(I) \subseteq I\}$ (Satz 5.12)
$= \mathrm{lfp}(T_P)$ (Satz 5.7, Tarski)
$= T_P \uparrow \omega$ (Satz 5.8, Kleene).

$\Rightarrow$ Das kleinste Herbrand-Modell M_P ist effektiv berechenbar (im Unendlichen).

Wichtiger Aspekt: Variablen in Anfragen.

Definition (Antwort)
Sei A eine Anfrage der Form $A_1, ..., A_n \Rightarrow$. Eine *Antwort* für A bezüglich P ist eine Substitution σ der in A vorkommenden Variablen.
Eine Antwort σ heißt *korrekt*, falls $\forall(\sigma(A_1 \wedge ... \wedge A_n))$ logische Konsequenz aus P ist.
Hierbei ist der *Allabschluß* $\forall(F)$ von F definiert durch $\forall(F) := \forall X_1 ... \forall X_n : F$, falls $X_1, ..., X_n$ alle Variablen aus F sind.

Korrekte Antworten $\approx$ intuitive Vorstellung von Prolog-Systemen.

Beispiel:
$P := \{\Rightarrow anhang([], E, \cdot(E, [])),\ anhang(R, E, RE) \Rightarrow anhang(\cdot(K, R), E, \cdot(K, RE))\}$
Anfrage: $anhang(\cdot(K, []), b, X) \Rightarrow$
Antwort: $\sigma = \{K/E, X/\cdot(E, \cdot(b, []))\}$
Diese Antwort ist korrekt, falls $\forall E : anhang(\cdot(E, []), b, \cdot(E, \cdot(b, [])))$ logische Konsequenz aus P ist.

deklarative Semantik, *statische Semantik*: korrekte Antworten

Aber: Prolog ist ein Beweissystem und hat eine *prozedurale (operationelle, dynamische) Semantik*: $\rightarrow$ Resolutionsprinzip

5.5 Beweisen mit logischen Programmen

P: logisches Programm

Präzisierung der Resolution: *SLD-Resolution*

(SLD: Linear Resolution for Definite Clauses with Selection Function)

Auswahlregel S: Wählt ein Literal aus einer Anfrage aus.

Beispiel:

- FIRST: Wähle das erste Literal
- LAST: Wähle das letzte Literal

Im folgenden sei S immer eine feste, aber beliebig gewählte Auswahlregel.

Definition (SLD-Resolutionsschritt)
Gegeben:

1. Anfrage $A_1, ..., A_{k-1}, A_k, A_{k+1}, ..., A_n \Rightarrow$
2. Klausel $B_1, ..., B_l \Rightarrow B$
3. Auswahlregel S wählt A_k aus

Ist σ mgu für A_k und B, dann heißt

$$\sigma(A_1), ..., \sigma(A_{k-1}),\ \sigma(B_1), ..., \sigma(B_l),\ \sigma(A_{k+1}), ..., \sigma(A_n) \Rightarrow$$

ableitbar aus der Anfrage und Klausel bezüglich S bzw. *Resolvente* aus der Anfrage und Klausel bezüglich S.

Definition (SLD-Ableitung)
Sei A eine Anfrage. Eine *SLD-Ableitung* von $P \cup \{A\}$ bezüglich S ist eine endliche Folge

$$G_1 \vdash_{\sigma_1} G_2 \vdash_{\sigma_2} G_3 ...$$

von Anfragen (goals) mit

1. $G_1 = A$
2. G_{i+1} ist Resolvente aus G_i und C_i mit mgu σ_i bezüglich S, wobei C_i eine Klausel aus P ist, in der eventuell einige Variablen umbenannt worden sind (vermeide Namenskonflikte für Variablen in G_i und C_i).

Beispiel: P:

1. $\Rightarrow p(a)$
2. $\Rightarrow p(b)$
3. $q(a) \Rightarrow p(a)$
4. $p(X) \Rightarrow q(X)$

SLD-Ableitung (S = LAST):
$p(b), \underline{q(a)} \Rightarrow$
$\vdash_{\{X/a\},(4)} p(b), \underline{q(a)} \Rightarrow$
$\vdash_{\{\},(1)} \underline{p(b)} \Rightarrow$
$\vdash_{\{\},(2)} \Rightarrow$ (bzw. $\Box$).

Definition (SLD-Widerlegung)
Eine *SLD-Widerlegung* ist eine SLD-Ableitung, die in $\Box$ endet. Eine *uneingeschränkte SLD-Widerlegung* ist eine SLD-Widerlegung, bei der beliebige Unifikatoren (statt mgu's) erlaubt sind.
Drei Möglichkeiten:

- *erfolgreiche* SLD-Ableitungen enden in $\Box$,
- *fehlgeschlagene* SLD-Ableitungen sind endlich, enden aber nicht in $\Box$ (z.B. $q(c) \Rightarrow \vdash_{\{X/c\}} p(c) \Rightarrow$),
- *unendliche SLD-Ableitungen.*

Eine erfolgreiche SLD-Widerlegung endet also in der leeren Klausel $\Box$, d.h. leere Disjunktion $\equiv$ false. Am Ende: Widerspruch abgeleitet
Bedeutung: $P \cup \{A\} \Rightarrow$ Widerspruch, damit ist $P \cup \{A\}$ unerfüllbar, d.h. $\neg A$ ist logische Konsequenz aus P (vergleiche Anmerkungen zu Satz 5.2).
Konstruktion von *Widerspruchsbeweis*en (typisch für Resolutionsbeweise)

Operationelle Semantik von Logikprogramm: SLD-Resolution

Definition (S-berechnete Antwort)
Eine *S-berechnete Antwort* σ für $P \cup \{A\}$ ist eine Substitution der Variablen in A, für die eine SLD-Widerlegung $A \vdash_{\sigma_1} A_1 \vdash \sigma_2 A_2 \vdash \ldots \vdash_{\sigma_n} \Box$ existiert, so daß $\sigma(x) = \sigma_n(\ldots(\sigma_2(\sigma_1(x))\ldots)$ für alle Variablen x in A.

Beispiel: Anfrage: $q(Z) \Rightarrow \vdash_{\sigma_1=\{X/Z\}} p(Z) \Rightarrow \vdash_{\sigma_2=\{Z/a\}} \Box$.
$\sigma_2 \circ \sigma_1 = \{X/a, Z/a\}$. Eine S-berechnete Antwort ist $\sigma = \{Z/a\}$; σ ist auch eine korrekte Antwort.

Satz 5.14 (Korrektheit der S-berechneten Antwort)
Jede S-berechnete Antwort σ für $P \cup \{A\}$ ist eine korrekte Antwort.

Beweis: Seien $A: \; A_1, \ldots, A_k \Rightarrow$ und $\sigma_1, \ldots, \sigma_n$ mgu's in der SLD-Widerlegung von $P \cup \{A\}$. Zu zeigen ist: Der Allabschluß $\forall(\sigma_n \circ \ldots \circ \sigma_1(A_1 \wedge \ldots \wedge A_k))$ ist logische Konsequenz aus P.
Induktion über n:

- $n = 1$: Dann: $A: \; A_1 \Rightarrow$, und es gibt ein Faktum $\Rightarrow B$ mit $\sigma_1(A_1) = \sigma_1(B)$. Somit ist $\forall \sigma_1(B)$ logische Konsequenz aus P, und damit auch $\forall \sigma_1(A_1)$.

- $n > 1$: Betrachte ersten Ableitungsschritt von A: Sei A_m das ausgewählte Literal und $B_1, ..., B_l \Rightarrow B$ die angewendete Klausel, d.h. $\sigma_1(A_m) = \sigma_1(B)$.
  $\Rightarrow$ Erste Resolvente: $G = \sigma_1(A_1, ..., A_{m-1}, B_1, ..., B_l, A_{m+1}, ..., A_k) \Rightarrow$
  Dann ist nach Induktionsvoraussetzung
  $$\forall(\sigma_n \circ ... \circ \sigma_2 \circ \sigma_1(A_1 \wedge ... \wedge A_{m-1} \wedge B_1 \wedge ... \wedge B_l \wedge A_{m+1} \wedge ... \wedge A_k))$$
  logische Konsequenz aus P.
  Daraus folgt, daß $\forall(\sigma_n \circ ... \circ \underbrace{\sigma_1(B)}_{\sigma_1(A_m)})$ logische Konsequenz aus P.
  Insgesamt ist also $\forall(\sigma_n \circ ... \circ \sigma_1(A_1 \wedge ... \wedge A_m \wedge ... \wedge A_k))$ logische Konsequenz aus P.

Vollständigkeit der SLD-Resolution:

Ist jede korrekt Antwort auch eine berechnete?

- Im allgemeinen nicht: Berechnung mit allgemeinsten Unifikatoren liefert allgemeinste Antworten
  Beispiel: $\Rightarrow p(X)$. Anfrage: $p(Z) \Rightarrow$.
  korrekt: $\sigma = \{Z/a\}$; berechnet: $\sigma' = \{Z/X\}$.
- Wir werden zeigen, daß jede korrekte Antwort eine Instanz einer berechneten Antwort ist.

Lemma 5.15 (mgu-Lemma)
Sei
$$A \vdash_{\sigma_1} A_1 \vdash_{\sigma_2} A_2 \vdash ... \vdash_{\sigma_n} \Box$$
eine uneingeschränkte SLD-Ableitung (d.h. $\sigma_1, ..., \sigma_n$ nicht notwendig mgu's sondern nur Unifikatoren). Dann existiert auch eine SLD-Widerlegung
$$A \vdash_{\sigma'_1} A'_1 \vdash_{\sigma'_2} A'_2 \vdash ... \vdash_{\sigma'_n} \Box$$
und eine Substitution φ mit $\varphi \circ \sigma'_n \circ ... \circ \sigma'_1 = \sigma_n \circ ... \circ \sigma_1$.

Beweis: Induktion über n:

- $n = 1$: Dann: $A \vdash_{\sigma_1} \Box$. Also: $A :\ L \Rightarrow$, und es gibt ein Faktum $\Rightarrow B$ mit $\sigma_1(L) = \sigma_1(B)$. Dann folgt mit dem *Unifikationssatz* aus Kapitel 3.2, daß es einen mgu σ'_1 für L und B und eine Substitution φ mit $\varphi \circ \sigma'_1 = \sigma_1$ gibt. Damit ist $A \vdash_{\sigma'_1} \Box$ eine SLD-Widerlegung.
- $n > 1$: Betrachte den ersten Schritt: Sei L ausgewähltes Literal und $B_1, ..., B_m \Rightarrow B$ die verwendete Klausel, d.h. $\sigma_1(L) = \sigma_1(B)$. Dann folgt aus dem Unifikationssatz, daß es einen mgu σ'_1 und eine Substitution φ mit $\varphi \circ \sigma'_1 = \sigma_1$ gibt.
  Also ist $A \vdash_{\sigma'_1} A'_1$ ein SLD-Schritt mit $\varphi(A'_1) = A_1$.
  Weiterhin: $A'_1 \vdash_{\sigma_2 \circ \varphi} A_2 \vdash_{\sigma_3} A_3 \vdash ... \vdash_{\sigma_n} \Box$ ist uneingeschränkte SLD-Ableitung

der Länge $n-1$.
Aus der Induktionsvoraussetzung folgt die Existenz einer SLD-Widerlegung $A'_1 \vdash_{\sigma'_2} A'_2 \vdash_{\sigma'_3} \ldots \vdash_{\sigma'_n} \Box$ und einer Substitution γ mit $\gamma \circ \sigma'_n \circ \ldots \circ \sigma'_2 = \sigma_n \circ \ldots \circ \sigma_2 \circ \varphi$. Also $\gamma \circ \sigma'_n \circ \ldots \circ \sigma'_2 \circ \sigma'_1 = \sigma_n \circ \ldots \circ \sigma_2 \circ \underbrace{\varphi \circ \sigma'_1}_{\sigma_1}$.

Zunächst: Variablenfreie Anfragen.

Satz 5.16
Sei L ein positives Literal ohne Variablen und logische Konsequenz aus P. Dann gibt es eine SLD-Widerlegung für $P \cup \{L \Rightarrow\}$.

Beweis: Da $L \in B_P$ und L logische Konsequenz ist, gilt $L \in M_P$ (Satz 5.10). Daraus folgt mit Satz 5.13: $L \in T_P \uparrow \omega$, und somit: $\exists n : L \in T_P \uparrow n$.
Induktion über n: Es gibt eine SLD-Widerlegung für $P \cup \{L \Rightarrow\}$.

- $n = 1$: $L \in T_P \uparrow 1$. Dann folgt aus der Definition von T_P, daß es ein Faktum $\Rightarrow A$ und eine Substitution σ mit $\sigma(A) = L$ gibt, d.h. $L \vdash_\sigma \Box$ ist uneingeschränkte SLD-Ableitung. Nun kann das mgu-Lemma 5.15 angewendet werden.

- $n > 1$: $L \in T_P \uparrow n$: Nach Definition von T_P gibt es eine Grundinstanz $\sigma(B_1, \ldots, B_k \Rightarrow B)$ mit $\sigma(B) = L$ und $\{\sigma(B_1), \ldots, \sigma(B_k)\} \subseteq T_P \uparrow (n-1)$. Dann existiert nach Induktionsvoraussetzung eine SLD-Widerlegung für $P \cup \{\sigma(B_i) \Rightarrow\}$.
  Da $\sigma(B_i)$ variablenfrei ist, gibt es eine SLD-Widerlegung für
  $P \cup \{\sigma(B_1), \ldots, \sigma(B_k) \Rightarrow\}$.
  Damit existiert eine uneingeschränkte SLD-Widerlegung für $P \cup \{L \Rightarrow\}$. Nun kann wieder das mgu-Lemma angewendet werden.

Nächster Schritt: Identische Substitutionen sind berechnete Antworten von logischen Konsequenzen.

Lemma 5.17
Sei L ein positives Literal. Falls $\forall(L)$ logische Konsequenz aus P ist, dann ist die Identität $\{\}$ eine S-berechnete Antwort für $P \cup \{L \Rightarrow\}$.

Beweis: L enthält Variablen $X_1, \ldots, X_n$. Seien $c_1, \ldots, c_n$ neue Konstanten (nicht in P oder L), und sei $\sigma = \{X_1/c_1, \ldots, X_n/c_n\}$.
$\forall(L)$ ist logische Konsequenz aus P, damit ist auch $\sigma(L)$ logische Konsequenz aus P. Mit dem vorangehenden Satz 5.16 folgt die Existenz einer SLD-Widerlegung für $P \cup \{\sigma(L) \Rightarrow\}$.
Da die c_i nicht in L oder P vorkommen, ersetze in dieser Ableitung c_i durch X_i, dadurch erhält man eine SLD-Widerlegung für $P \cup \{L \Rightarrow\}$, bei der kein X_i ersetzt wird.

Lemma 5.18 (Lifting-Lemma)
Gegeben P, A, σ. Falls eine SLD-Widerlegung für $P \cup \{\sigma(A)\}$ mit mgu's $\sigma_1, \ldots, \sigma_n$ existiert, dann existieren auch eine SLD-Widerlegung für $P \cup \{A\}$ mit mgu's $\sigma'_1, \ldots, \sigma'_n$

und eine Substitution φ mit $\sigma_n \circ ... \circ \sigma_1 \circ \sigma = \varphi \circ \sigma'_n \circ ... \circ \sigma'_1$.

Beweis: Betrachte den ersten Schritt $\sigma(A) \vdash_{\sigma_1} A_1$. Sei $B_1, ..., B_k \Rightarrow B$ die verwendete Klausel und $\sigma(L)$ das ausgewählte Literal. o.B.d.A sei $\sigma(B) = B$. Dann gilt: $\sigma_1(\sigma(L)) = \sigma_1(B) = \sigma_1(\sigma(B))$, also ist $A \vdash_{\sigma_1 \circ \sigma} A_1$ ein uneingeschränkter SLD-Schritt (da σ nicht notwendig ein mgu ist). Wegen des mgu-Lemmas 5.15 gibt es eine gewünschte SLD-Widerlegung.

Satz 5.19 (Vollständigkeit der SLD-Resolution)
Für jede korrekte Antwort σ für $P \cup \{A\}$ existieren eine S-berechnete Antwort σ' für $P \cup \{A\}$ und eine Substitution φ, so daß $\sigma(x) = \varphi(\sigma'(x))$ für alle Variablen x in A.

Beweis: Sei $A: \quad L_1, ..., L_k \Rightarrow$. Dann ist $\forall(\sigma(L_1 \wedge ... \wedge L_k))$ logische Konsequenz aus A. Nach Lemma 5.17 ist die Identität $\{\}$ eine S-berechnete Antwort für jedes $P \cup \{\sigma(L_i) \Rightarrow\}$. [1] Kombiniere diese zu einer SLD-Widerlegung für die Anfrage $P \cup \{\sigma(A)\}$ mit S-berechneter Antwort $\{\}$. Seien $\sigma'_1, ..., \sigma'_n$ die mgu's in dieser Ableitung, $\sigma'_n \circ ... \circ \sigma'_1$ die Identität auf der Anfrage. Dann existieren nach dem Lifting-Lemma 5.18 eine SLD-Widerlegung für $P \cup \{A\}$ mit mgu's $\sigma_1, ..., \sigma_n$ und eine Substitution φ mit $\varphi \circ \sigma_n \circ ... \circ \sigma_1 = \underbrace{\sigma'_n \circ ... \circ \sigma'_1}_{\{\}} \circ \sigma$. Sei σ' die Einschränkung von $\sigma_n \circ ... \circ \sigma_1$ auf Variablen in A, dann folgt $\varphi(\sigma'(x)) = sigma(x)$.

Die Bedeutung der Sätze 5.14 und 5.19 ist die *Äquivalenz der deklarativen und operationellen Semantik*.

Die Auswahlregel S war bisher beliebig. Im folgenden: FIRST (in Prolog)

Wie findet man SLD-Widerlegungen?

Beispiel:
(1) $q(X, Y), p(Y, Z) \Rightarrow p(X, Z)$, (2) $\Rightarrow p(X, X)$, (3) $\Rightarrow q(a, b)$.
Anfrage: $p(S, b) \Rightarrow$
SLD-Ableitungen:

1. $p(S, b) \Rightarrow \vdash_{(1)} q(S, Y), p(Y, b) \Rightarrow \vdash_{(3),[S/a,Y/b]} p(b, b) \Rightarrow \vdash_{(1)}$ (Umbennenung der Variablen in Klausel 1) $\underline{q(b, Y_1)}, p(Y_1, b) \Rightarrow$ Fehlschlag.

2. $p(S, b) \Rightarrow \vdash_{(1)} q(S, Y), p(Y, b) \Rightarrow \vdash_{(3),[S/a,Y/b]} p(b, b) \Rightarrow \vdash_{(2)} \square$.
   Erfolg: $\sigma = \{S/a\}$.

Systematische Methode zum Finden von SLD-Widerlegung (Beschreibung: *markierte Bäume*.

[1] Bemerkung: nicht $\sigma(P)$, da alle Regeln in P $\forall$-quantifiziert sind.

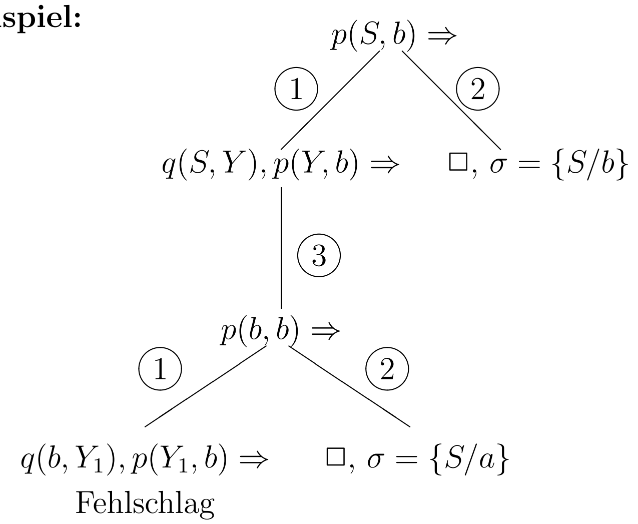
Definition (SLD-Baum)
Ein *SLD-Baum* für $P \cup \{A\}$ ist ein markierter Baum, dessen Kanten mit Anfragen markiert sind, mit:

1. Die Wurzel ist mit A markiert.

2. Ist S ein Knoten und sind $S_1, ..., S_n$ alle Resolventen aus S und Klauseln in P, dann hat S die Söhne $S_1, ..., S_n$.

3. Die leere Klausel hat keinen Sohn.

Beispiel:



Es ergeben sich drei verschiedene SLD-Ableitungen (inclusive einem Fehlschlag).

Definition
Ein *Erfolgszweig* ist eine Kantenfolge von der Wurzel zum Blatt $\square$,
ein *Mißerfolgszweig* ist eine Kantenfolge von der Wurzel zu einem Blatt $\neq \square$,
und ein *unendlicher Zweig* ist eine unendliche Knotenfolge von der Wurzel.

Nach Vollständigkeitssatz 5.19 gilt: Wenn $P \cup \{A\}$ unerfüllbar ist, dann enthält der SLD-Baum für $P \cup \{A\}$ einen Erfolgszweig. Wie findet man diesen?

Definition (SLD-Strategie)
Eine *SLD-Strategie* (*Suchregel*) ist eine Methode zum Durchsuchen eines (SLD-) Baumes.

Erfolgreiche SLD-Strategie: *Breitendurchlauf* durch Ebene für Ebene des Baumes.

Beispiel:

```
       w
    /  |  \
   a   b   c
 / | \    / \
 x y z   r   s
```

Breitendurchlauf: w a b c x y z r s
Sichere SLD-Strategie: Erfolgszweige werden immer gefunden.
Nachteil: Hoher Verwaltungsaufwand. Speichere jeweils eine Ebene. Wenn jeder Knoten zwei Söhne hat, dann hat Ebene 10 (10 SLD-Schritte) 1024 Knoten! $\Rightarrow$ riesiger Speicheraufwand. Diese Methode ist also für eine praktische Implementierung ungeeignet.

Daher verzichtet man in Prolog auf eine sichere Strategie.
Unterschied zwischen Prolog und rein logischer Programmierung!

5.6 Beweisen in Prolog

Syntaktischer Vergleich:

logische Programme	Prolog
Konstanten: a,b,c,...	Atome: `a,b,c,...`
	Zahlen: `0,1,...`
Terme: $f(...)$	Struktur, Term: `f(...)`
Prädikate: $p(...)$	Prädikate: `p(...)`
Klauseln: $\Rightarrow A$	Faktum: `A.`
$A_1, ..., A_n \Rightarrow A$	Regel: `A :- A1,...,An.`
Anfrage: $A_1, ..., A_n \Rightarrow$	Anfrage: `?- A1,...,An.`

Semantischer Vergleich:

Deklarative Semantik $\iff$ operationelle Semantik	nur operationelle Semantik
Programm: *Menge* von Klauseln	Programm: *Folge* von Klauseln (Ordnung: Reihenfolge der Eingabe)

Suchstrategie von Prolog: *Tiefendurchlauf* durch SLD-Baum
Voraussetzung: Ordnung der Klauseln entspricht Ordnung der Söhne im SLD-Baum.

```
DFS(Knoten) = IF <Knoten ist Blatt>
              THEN <besuche Knoten>
              ELSE     { Knoten hat Soehne K1,...,Kn }
                <besuche Knoten>
                DFS(K1);
                ...
                DFS(Kn);
```

Für den Beispiel-Baum w: $DFS(w)$: w a x y z b c r s

Problem bei DFS: Unendliche Zweige, die links von Erfolgszweigen stehen.

Beispiel: (1) `p :- p.`, (2) `p.`
DFS terminiert nicht, d.h. findet keine Lösung. Vermeidung durch Vertauschen von (1) und (2).

In Prolog sind zwei Reihenfolgen relevant:

1. Reihenfolge der Klauseln
2. Reihenfolge der Literale in rechten Regelseiten

Dadurch ergibt sich ein Unterschied zur logischen Programmierung (bei der keine Reihenfolgen existieren).

6 Nichtdeklarative Bestandteile von Prolog

Prolog: Programmieren in Logik

Programm: Beschreibung logischer Sachverhalte. Der Computer zieht Schlußfolgerungen.

Diese Idealvorstellung ist praktisch nicht anwendbar. In diesem Kapitel fragen wir: wo sind Abstriche von der reinen Logik?

6.1 Beweisstrategie

Backtracking Verfahren:

- Lösungssuche in mehreren Schritten.
- Bei Alternative: probiere eine aus,
- bei Sackgassen: mache Beweisschritte rückgängig und probiere eine andere Alternative aus.

Prolog verwendet Backtracking zur Lösungssuche.

Resolutionsprinzip (ohne Unifikation)
Um ?- $A_1, ..., A_{k-1}, A_k, A_{k+1}, ..., A_n$. zu beweisen, reicht es aus,
?- $A_1, ..., A_{k-1},\ L_1, ..., L_p,\ A_{k+1}, ..., A_n$. zu beweisen, falls L :- $L_1, ..., L_p$ und $A_k = L$.
Auch auf Fakten anwendbar: $p = 0 \Rightarrow$ Verkürzung der Anfrage.
Kann eine Anfrage auf die leere Anfrage $\Box$ verkürzt werden, wird die Antwort „yes“ gegeben.

Das Resolutionsprinzip ist nicht eindeutig:
1) Welches Literal einer Anfrage wird abgearbeitet?
2) Welche/s Regel/Faktum wird zur Abarbeitung verwendet?

Zu 1) In Prolog wird grundsätzlich das erste (=linke) Literal abgeleitet. Kapitel 5: Dies beeinflußt nicht die Vollständigkeit.
Zu 2) Kapitel 5: Eine sichere Methode ist, alle Fakten/Regeln gleichzeitig zu untersuchen. Der Nachteil ist, daß dies sehr aufwendig ist. In Prolog wird daher auf diese sichere Methode verzichtet! Stattdessen wird das Backtracking-Verfahren verwendet.

1. Klauseln haben eine Reihenfolge (wie im Programmtext).
2. Im Beweisschritt: Wähle erste passende Klausel zur Ableitung des linken Literals.
   Bei Sackgassen: Mache Beweisschritt rückgängig und wähle nächste Klausel zur Ableitung.

3. Vor einem Beweisschritt: Benenne die Variablen aus der Klausel um, so daß sie verschieden von den Variablennamen in der Anfrage sind.

4. Bei einem Beweisschritt: Ersetze die Variablen durch Terme (Unifikation). Die Variablen werden *gebunden* bzw. *instanziiert*.

Die Punkte 1 bis 4 beschreiben *Prologs Beweisstrategie*.

Beispiel: Programm:

```
p(a).
p(b).
q(b).
```

`?- p(X),q(X).`
$\vdash_{\{X/a\}}$ (1. Klausel)
`?- q(a).`
Sackgasse. Ableitung rückgängig machen
`?- p(X),q(X).`
$\vdash_{\{X/b\}}$ (2. Klausel)
`?- q(b).`
$\vdash_{\{\}}$
$\Box$, Antwort: `X=b`

Probleme der Beweisstrategie:
Das Backtracking-Verfahren ist *unvollständig*.

Beispiel: Programm `p:-p.` `p.` Die Anfrage `?- p.` terminiert nicht, obwohl `?- p.` beweisbar ist.

Beispiel für Relevanz der Reihenfolge der Klauseln: Programm

```
last([K|R],E) :- last(R,E).
last([E],E).
```

`?- last(L,3).`
$\vdash_{\{L=[K_1|R_1],E_1=3\}}$ (1. Klausel)
`?- last(R1,3).`
$\vdash_{\{R_1=[K_2|R_2],E_2=3\}}$
`?- last(R2,3).`
$\vdash$ ... terminiert nicht.
Bei Vertauschung der Klauseln:
`?- last(L,3).`
$\vdash_{\{L=[E],E=3\}}$ (Faktum)
$\Box$, Antwort: `L=[3]`

Beispiel für Relevanz der Reihenfolge der Literale in rechten Regelseiten: Programm

```
groesser_gleich (X,Z) :- groesser_gleich (Y,Z),
                         addiere_1 (Y,X).
groesser_gleich (X,X).
addiere_1 (0,S(0)).
addiere_1 (S(X),S(Y)) :- addiere_1 (X,Y).
```

terminiert nicht bei `?- groesser_gleich (G,0).` wegen *linksrekursiv*er Regel (allgemein: `p(...) :- p(...),...`), terminiert aber bei Vertauschung der ersten beiden Regeln.

Fazit: Das Beweisverfahren von Prolog ist unvollständig. Nicht alles, was logisch beweisbar ist, kann Prolog herleiten. Durch Kenntnis der Beweisstrategie können Endlosbeweise vermieden werden:

1. Spezialfälle vor Allgemeinfällen
2. Vorsicht bei linksrekursiven Regeln

Bei Problemen mit Endlosbeweisen: *Debugging*.

6.2 Der „Cut"-Operator

Prolog-Beweisstrategie: Backtracking.
Bei Implementierung muß man sich die aktuellen Informationen an allen Alternativpunkten speichern. ⇒ bei langen Beweisen kann es zu einem Speicherüberlauf kommen.
Durch „!" (*Cut*) kann der Benutzer das Backtracking teilweise unterdrücken. ⇒ geringerer Speicherbedarf.
Der Cut schneidet Teile der SLD-Baumes ab. Diese Teile können Erfolgszweige enthalten!
Bisher: eventuell Endlosbeweise statt „yes".
Mit Cut: eventuell „no" statt „yes".
Gründe für Cut:

1. Effizienz (Speicherbedarf, Laufzeit)
2. Kennzeichnen von Funktionen
3. Vermeidung von Laufzeitfehlern mit Arithmetik („is")

Konzept: „!" darf anstelle von Literalen auf rechten Regelseiten stehen: `p :- q,!,r`
Bedeutung (prozedural): Falls diese Klausel zum Beweis verwendet wird, dann gilt:

1. Falls `q` nicht beweisbar ist, wähle nächste Klausel für `p`.
2. Falls `q` beweisbar ist, dann ist `p` **nur dann** beweisbar, wenn `r` beweisbar ist.
   (Also kein Alternativbeweis für `q`, keine andere Klausel für `p`.)

Beispiel:

```
ja :- ab(X),!,X=b.
ja.
ab(a).
ab(b).
```

Dann ist rein logisch `?- ja.` auf zwei Arten beweisbar. Dennoch antwortet Prolog mit „no“.
⇒ Vorsicht mit Cut!

Sinnvolleres Beispiel:

```
istKindVon (K,E) :- weiblich (E), !, istMutterVon (E,K).
istKindVon (K,E) :- istVaterVon (E,K).
```

Häufig wird der Cut zur *Fallunterscheidung* genutzt:

```
p :- q, !, r.
p :- s.
```

entspricht „if q then r else s.“

Beispiel:

```
max (X,Y,Z) :- X>=Y, !, Y=Z.
max (X,Y,Z) :- Y=Z.
```

6.3 Negation

Reine Logikprogramme erlauben keine Negation (negative Bedingungen). Für praktische Anwendungen ist Negation aber notwendig.

Beispiel:

```
istSchwesterVon (S,P) :- weiblich (S),
                         istMutterVon (M,S),
                         istMutterVon (M,P),
                         "not S=P".
```

Zur Formulierung negativer Aussagen bietet Prolog *Negation als Fehlschlag*:
`\+p` ($\not\vdash p$)
ist beweisbar, falls alle Beweise für p fehlschlagen.

Beispiel:

```
?- \+ monika=susanne.
yes.
?- \+ maennlich (andreas).
no.
```

Negation als Fehlschlag ist **nicht** logische Negation.

Beispiel: `p :- \+p.`
Bedeutung bei logischer Negation: $\neg p \Rightarrow p \;\equiv\; \neg(\neg p) \vee p \equiv p$. Das bedeutet: `?- p.` müßte beweisbar sein. Prolog gerät jedoch in eine Endlosschleife.

Was ist die Bedeutung von `\+ p.` ?
Ist p falsch (in allen Modellen)? Nein! Die Herbrandbasis B_P ist Modell von P, d.h. es gibt ein Modell, in dem p wahr ist.
Dies bedeutet: `\+p` ist **nie** logische Konsequenz aus P.

Lösung:
Closed World Assumption (*CWA*) (*Reiter* 1978):
Falls ein variablenfreies Prädikat p nicht logische Konsequenz aus P ist, dann gilt ¬ p. (Also: alles, was nicht zwingend wahr ist, ist falsch.)
CWA ist eine neue *Inferenzregel*. Da die „logische Konsequenz“-Eigenschaft unentscheidbar ist, müssen wir das abschwächen:

Negation as (finite failure) (*NF*) (*Clarke* 1978):
Falls alle Beweise für p fehlschlagen und endlich sind, dann gilt `\+p`.

NF ist schwächer als CWA: Falls der SLD-Baum für p fehlgeschlagen ist und einen unendlichen Zweig enthält, dann gilt `\+p` bezüglich CWA, aber nicht bezüglich NF.

Andererseits ist NF effektiv implementierbar:

Beispiel: `p :- \+q.` ist implementierbar durch

```
p :- q, !, fail.
p.
```

Was ist die logische Entsprechung zu NF? Die bisherige Semantik ist ungeeignet (Klauseln als Implikationen). Betrachte die *Vervollständigung* (*Clark's Completion*) eines Programmes:

1. Sei = ein binäres Prädikatssymbol.

2. Sei $p(t_1, ..., t_n) :- L_1, ..., L_m$ eine Klausel aus P.
   Dann transformiere diese in die Formel

   $$(\exists Y_1 ... \exists Y_l \; X_1 = t_1 \wedge ... \wedge X_n = t_n \wedge L_1 \wedge ... \wedge L_m) \Rightarrow p(X_1, ..., X_n).$$

   Dabei seien $X_1, ..., X_n$ neue Variablen und $Y_1, ..., Y_l$ die ursprünglichen Variablen.

3. Seien $F_1 \Rightarrow p(X_1, ..., X_n), ..., F_k \Rightarrow p(X_1, ..., X_n)$ alle transformierten Klauseln für ein Prädikat p. Dann heißt

$$\forall(F_1 \vee ... \vee F_k \iff p(X_1, ..., X_n))$$

die *vervollständigte Definition* von p.

4. Die *Gleichheitstheorie* besteht aus folgenden Formeln:

   - $\forall X : X = X$
   - $\forall X \forall Y \forall Z : X = Y \wedge Y = Z \Rightarrow X = Z$
   - $\forall X \forall Y : X = Y \Rightarrow Y = X$
   - $\forall(X_1 = Y_1 \wedge ... \wedge X_n = Y_n \Rightarrow f(X_1, ..., X_n) = f(Y_1, ..., Y_n))$ für alle Funktionssymbole f
   - $\forall(X_1 = Y_1 \wedge ... \wedge X_n = Y_n \wedge p(X_1, ..., X_n) \Rightarrow p(Y_1, ..., Y_n))$ für alle Prädikatssymbole p
   - $\forall(f(X_1, ..., X_n) \neq g(Y_1, ..., Y_m))$ für alle Funktionssymbole $f \neq g$
   - $\forall(X \neq t)$, falls X in t vorkommt
   - $\forall(X_1 \neq Y_1 \vee ... \vee X_n \neq Y_n \Rightarrow f(X_1, ..., X_n) \neq f(Y_1, ..., Y_n))$ für alle Funktionssymbole f (wobei $s \neq t$ für $\neg(s = t)$ steht)

5. Die *Vervollständigung* (*Clark's Completion*) eines Programmes P ist die Formelmenge bestehend aus der Gleichheitstheorie und den vervollständigten Definitionen der Prädikate. Bezeichnung: $comp(P)$.

Intuitiv werden bei der Vervollständigung die Implikationen durch Äquivalenzen ersetzt. Dadurch erhalten wir die Entsprechung zur CWA-Regel. Wir definieren nun „logische Konsequenz“ und „korrekte“ Antworten bezüglich $comp(P)$.
Außerdem: Einschränkung der Selektionsfunktion S: S selektiert nur dann ein negatives Literal `\+L`, falls L variablenfrei ist.
Problem: *Floundering*: Die Anfrage enthält nur noch negative Literale mit Variablen.

Definition (SLDNF-Resolution)
Wie SLD-Resolution, jedoch mit folgender Abweichung:
Bei Selektion eines negativen Literals $\backslash + L$: Berechne ?–L. Falls diese Berechnung nur Fehlschläge enthält und endlich ist, dann ist $\backslash + L$ bewiesen.

SLDNF-Resolution ist korrekt bezüglich $comp(P)$, aber im allgemeinen nicht vollständig wegen

1. Floundering

2. unendlichen Beweisen bei negativen Literalen

Beispiel zu 1):

```
p(X).
q(a).
r(b).
?- p(X), \+q(X).
```

Rein logisch ist $\{X/b\}$ eine korrekte Antwort. Diese Lösung kann aber nicht bewiesen werden.

Beispiel zu 2):

```
r :- \+p.
r :- p.
p :- p.
```

Hier ist die Identität $\{\}$ eine korrekte Antwort bezüglich der Anfrage ?–r. Diese ist nicht berechenbar.

Vollständigkeitsergebnisse für eingeschränkte Klassen von Programmen und Zielen ⇒ *Lloyd*-Buch

Achtung: Selektionsfunktion nicht sicher! Negative Literale werden auch selektiert, wenn L noch Variablen enthält.

Negation as failure:

- keine Variablenbindung (Ausweg: „*konstruktive Negation*“)
- nur variablenfreie Literale negieren (sonst falsche Antworten!)

Beispiel:

```
p(a,a).
p(a,b).
?- \+p(b,b).
yes.
?- \+\+p(b,b).
no.
?- \+p(X,b).
```

`no.` (← rein logisch: $X = b$!) `?- \+\+p(X,b).`
`X=_0` (← rein logisch falsche Antwort, da z.B. falsch für $X = b$.)

⇒ Beim Beweis negativer Literale: Literal muß variablenfrei sein!

Beispiel:

```
istSchwesterVon (S,P) :- weiblich (S),
                         istMutterVon (M,S),
                         istMutterVon (M,P),
                         \+ S=P.     % darf nicht erste Bedingung sein
```

Ansonsten (falls `\+ S=P` erste Bedingung) würde

```
?- istSchwesterVon (angelika,P).
```

„no.“ ergeben (z.B. $P = angelika$)

Ausweg: *verzögerte Negation* (in den meisten Prolog-Systemen möglich; nicht in SWI-Prolog)
Idee: Verzögere den Beweis von negativen Literalen `\+L` solange, bis L variablenfrei ist.

Beispiel:

```
istSchwesterVon (S,P) :- \+S=P, weiblich(S),
                         istMutterVon (M,S),
                         istMutterVon (M,P).
```

```
  ?- istSchwesterVon (angelika,P).
⊢  ?- \+ angelika = P, weiblich (angelika),
    istMutterVon (M,angelika), istMutterVon (M,P).
⊢  ?- \+ angelika = P, istMutterVon (M,angelika), istMutterVon (M,P).
⊢  ?- \+ angelika = P, istMutterVon (christine,P).
⊢  ?- \+ angelika = herbert
```

$\rightarrow$ „yes“.

- Verzögerung zulässig wegen flexibler Selektionsstrategie
- Implementierung: Prolog-System mit *Coroutining*, z.B. Sicstus-Prolog:
  statt `\+L` nun: `when(ground(L),\+L)`
  $\Rightarrow$ logisch korrekte Negation

6.4 Zyklische Strukturen

Unifikation: $X = t$: Binde X an t nur, falls X in t nicht vorkommt (*occur check*, *Vorkommenstest*)
Fast alle Prolog-Systeme haben keinen occur check. Dann ist $X = f(X)$ unifizierbar mit $\sigma = \{X/f(f(f(...))\}$: *zyklische Struktur*.

Probleme: Ausgabe und Unifikation zyklischer Strukturen; Logik wird zerstört.

```
test :- X=1+X.
?- test.
yes.
```

obwohl rein logisch `test` nicht beweisbar ist.

Unendliche Unifikation:

```
?- X=f(X), Y=f(Y), X=Y.
```

führt zu Endlosschleife. (Dies tritt in der Praxis aber kaum auf!)

6.5 Arithmetik

Bisher: $3 * 5$ verschieden von 15.

Definition (arithmetischer Ausdruck)
Ein *arithmetischer Ausdruck* ist ein Term mit Funktoren $+, -, /, *, mod$ (Infixoperatoren), $+, -$ (Präfixoperatoren) und Zahlen und Variablen ($\leftarrow$ an arithmetische Ausdrücke gebunden).

Beispiel: $3 * 5$, $X * 5 + Y$

`is(X,Y)` ist vordefiniert. `is` ist ein Infixoperator, und `is(X,Y)` ist beweisbar, wenn

1. Y ein variablenfreier arithmetischer Ausdruck ist und
2. der ausgerechnete Wert von Y (also eine Zahl) mit X unifizierbar ist.

```
?- 16 is 3*5+1.
yes.
?- 2+1 is 2+1.
no.
```

Anmerkungen:

1. `is` ist ein partielles Prädikat: es gibt eventuell eine Fehlermeldung, falls Y kein variablenfreier arithmetischer Ausdruck ist.
2. Die Reihenfolge bei der Verwendung von `is` ist relevant:
   ```
   ?- X=2, Y is 3-X.
   X=2, Y=1
   ?- Y is 3-X, X=2.
   ```
   $\rightarrow$ Fehlermeldung

Beispiel: Fakultätsfunktion (Reihenfolge wichtig!)

```
fak (0,1).
fak (N,F) :- N1 is N-1, fak (N1,F1), F is F1*N.
?- fak (6,F).
F=720
```

Wird nun mit „;“ die Berechnung einer zweiten Lösung aufgerufen, gerät Prolog in eine Endlosschleife (durch Berechnung von 0! mit der zweiten Regel).
Dies kann vermieden werden, indem die erste Klausel in
`fak (0,F) :- !, F=1.`
umgeschrieben wird.

Vergleichsprädikate

- `X =:= Y`: Wertgleichheit
- `X =\= Y`: Wertungleichheit
- `X > Y`
- `X < Y`
- `X >= Y`
- `X =< Y`

Diese rechnen X und Y aus (wie bei `is`) und vergleichen die Ergebnisse.

Weitere Arithmetik:

- Gleitkommazahlen (Punktnotation)
- Logarithmus, trigonometrische Funktionen

Die Arithmetik ist logisch unvollständig:

```
?- 5 is 3+2.
yes
?- X is 3+2.
X=5
?- 5 is Y+2.
```

→ Fehler
(Lösung: → *Constraint* Logic Programming)

6.6 Ein-/Ausgabe von Daten

Bisher: $\texttt{append(}\underbrace{\texttt{[a,f],[f,e]}}_{\texttt{Eingabe}}\texttt{,L).}$
liefert Ausgabe `L=[a,f,f,c]`.

Der *write*-Befehl: `write (X)`

- X beliebiger Term,

- `write (X)` ist immer beweisbar (genau einmal)
- Nebeneffekt: Ausgabe von X auf dem Bildschirm, wobei die Operatoren berücksichtigt werden (Infix)
- Beim Backtracking kein Rücksetzen der Ausgabe

Beispiel:

```
?- write (+(2,3)), a=b.
2+3  no.
```

Der *nl*-Befehl: `nl`
Wie `write`, aber Zeilenvorschub:

```
?- write (a), nl, write (b), nl.
a
b
yes.
```

Der *read*-Befehl: `read (X)`

- Nebeneffekt: Nächster Term wird gelesen (abgeschlossen mit „.“ und Blank/Zeilenvorschub)
- beweisbar genau dann, wenn der gelesene Term mit X unifizierbar ist
- Es gibt nur einen Beweis.
- Kein Rücksetzen der Eingabe bei Fehlschlag ($\rightarrow$ nichtlogisches Prädikat)

Beispiel: Kleine Interaktion für Verwandtschaftsbeispiel: Interaktive Ausgabe des Vaters:

```
vater :- write ('Wessen Vater?'), nl,
         read (P),
         istVaterVon (V,P),
         write ('Vater von'), write (P),
         write (' ist '), write (V), write ('.'), nl.
?- vater.
Wessen Vater?
maria.
Vater von maria ist anton.
yes.
?-
```

Ein-/Ausgabe einzelner Zeichen: `get0(N)`
$\rightarrow N =$ ASCII-Wert des nächsten gelesenen Zeichens.

Weiter: Dateizugriff, Anschlüsse an Fenstersysteme

6.7 Zerlegung und Konstruktion von Termen

Der =..-Befehl: `T=..L` ist beweisbar, falls

- T Struktur, L Liste
- $f(t_1, ..., t_n) = ..[f, t_1, ..., t_n]$
- T und L dürfen beim Beweis nicht gleichzeitig Variablen sein.

Anwendung:

1. *Zerlegung von Termen*:
```
?- f(a,g(b)) =.. L.
L = [f,a,g(b)]
```
2. *Konstruktion von Termen*:
```
?- T =.. [datum,1,3,90].
T = datum(1,3,90)
```

Beispiel: `sucheKonst (T,L)`:
(T Grundterm, L Liste der Konstanten in T)

```
sucheKonst (T,L) :- addKonst (T,[],L).
addKonst (T,L,[K|L]) :- T=..[K], !.     % T ist Konstante
addKonst (T,L,NewL) :- T=..[F|Arg], addKonstInListe (Arg,L,NewL).
addKonstInListe ([],L,L).
addKonstInListe ([T|TListe],L,NewL) :-
        addKonst (T,L,LmitT),
        addKonstInListe (TListe,LmitT,NewL).
?- sucheKonst (3+4*5-f(a,b),L).
L = [b,a,5,4,3]
```

Das *var*-Prädikat: `var (X)` beweisbar genau dann, wenn X eine freie Variable ist.

Erweiterung von `sucheKonst` auf Terme mit Variablen: Zusätzlich erste Klausel
```
addKonst (T,L,L) :- var (T), !.
```

6.8 Daten als Programme

Die Klausel `p :- q,r,s.` entspricht dem Term `:-(p,',','(q,','(r,s)))`.

Umgekehrt: Terme $\rightsquigarrow$ Programme

`call (X)`

- X instanziiert mit Term, der als Anfrage interpretiert wird (ohne `?-`)
- beweisbar, falls `?- X.` beweisbar ist

Beispiel:

```
?- X=3, call(is(Y,X*2)).
X=3
Y=6
?- T =.. [is, Y, 2*5], call (T).
T=10 is 2*5
Y=10
```

`map (P,L1,L2)`
P Prädikat (zwei-stellig), map wendet P auf Elemente von L_1 an.

Beispiel:

```
zweifach (X,Y) :- Y is 2*X.
?- map (zweifach,[2,5,3],L).
L=[4,10,6]

map (P,[],[]).
map (P,[E1|R1],[E2|R2]) :-
     T =.. [P,E1,E2],      % T = P(E1,E2)
     call (T),
     map (P,R1,R2).
```

Semantik: *Prädikatenlogik zweiter Stufe* (Quantifizierung über Prädikate)

6.9 Programme als Daten

Datenbank (des Prolog-Systems) = Folge aller Klauseln, die gegenwärtig dem System bekannt sind; Liste von Termen

Manipulation der Datenbank:
Einfügen in die Datenbank:

- `asserta (K)` Seiteneffekt: Füge Klausel K am Anfang der Datenbank ein
- `assertz (K)` Seiteneffekt: Füge Klausel K am Ende der Datenbanl ein

Bei Backtracking: Einfügung bleibt.

Beispiel: (Datenbank leer)

```
?- asserta (:-(p,',')(q,r))), asserta (p), fail.
no.
```

Inhalt der Datenbank jetzt: `p.` `p :- q,r.`
`?- listing.` listet alle gegenwärtig bekannten Klauseln auf (bei Sicstus Prolog: nicht bei compiliertem Code)

Beispiel:

```
neueKlauseln :- repeat,
                read (T),
                speicher_pruefe (T).
speicher_pruefe (end_of_file).
speicher_pruefe (T) :- assertz (T), !, fail.
repeat.
repeat :- repeat.
```

(repeat: unendliche viele Beweise → failure loop)

Löschen von Klauseln:

- `retract (K).` beweisbar, wenn in der Datenbank eine Klausel existiert, die mit K unifizierbar ist (K muß mindestens einen Prädikatnamen enthalten)
- Nebeneffekt: diese Klausel wird gelöscht.
- Backtracking: lösche weitere Klausel.

Beispiel: Löschen aller Klauseln eines Prädikats P $(= p(X_1, ..., X_n)\ )$

```
loescheKlauseln (P) :- retract (P), fail.
loescheKlauseln (P) :- retract (:- (P,Rumpf)), fail.
loescheKlauseln (P).

?- loescheKlauseln (letztes(L,E)).
```

- `clause (H,B)` beweisbar, wenn eine Klausel `H :- B` in der Datenbank existiert.
  (falls `H.` in Datenbank ⇒ $B = true$.)

7 Fortgeschrittene Programmiertechniken

7.1 Differenzlisten

`append (L1,L2,L3)`

- ist definiert über Aufbau der Liste L_1. ⇒ Laufzeit linear in Länge von L_1.
- Verbesserung: direkter Zugriff auf das Listenende

Differenzlisten: Liste darstellen als „Differenz“ zweier Listen:
`[a,b,c]` entspricht
`[a,b,c,d] - [d]`
`[a,b,c] - []`
`[a,b,c|L] - L` (allgemein)

Allgemein: Repräsentiere `[e1,...,en]` durch `[e1,...,en|L]-L`.

Vorteil: `M-L` Differenzliste ⇒ konstanter Zugriff auf das Ende (L)

Konkatenation von Differenzlisten:

```
append_dl (L-M,M-N,L-N).
?- append_dl ([1,2|L1]-L1, [3,4|L2]-L2, L3).
L3 = [1,2,3,4|L2]-L2
```

Nur ein Resolutionsschritt!

Anwendung: Umkehren einer Liste. Zunächst naiv:

```
rev ([],[]).
rev ([E|R],L) :- rev (R,UR), append (UR,[E],L).
```

Laufzeit: quadratisch in der Länge der ersten Liste.

Erste Verbesserung: Darstellung des zweiten Argumentes von `rev` als Differenzliste:

```
rev_dl ([],L-L).
rev_dl ([E|R],L-M) :- rev_dl (R,UR-T), append_dl (UR-T,[E|N]-N,L-M).
```

(`T=[E|N], M=N, L=UR`)

Zweite Verbesserung:

```
rev_dl ([],L,L).
rev_dl ([E|R],L,M) :- rev_dl (R,L,[E|M]).
rev (L,M) :- rev_dl (L,M,[]).
```

⇒ lineare Laufzeit

Differenzlisten sind im allgemeinen kein universeller Ersatz für Listen sondern Listen mit offenem Ende, die genau einmal konkateniert werden können:

Beispiel:

```
?- DL = [a|L]-L,
   append_dl (DL,[b|M]-M,X),
   append_dl (DL,[c|N]-N,Y).
no.
```

(`L=[b|M]` und `L=[c|N]`, Widerspruch) ⇒ Der Benutzer muß die „Einmal-Konkatenation“ sicherstellen!

7.2 Meta-Interpretierer

Programme $\longleftrightarrow$ Daten: Programmierung eines Interpreters für Prolog in Prolog: *Meta-Interpretierer*

Vorteil: Erweitern von Meta-Interpretierern:

```
\item{Beweisstrategie "andern}
\item{Sprachumfang erweitern}
\item{neue Programmierumgebungen (Debugger)}
```

Meta-Interpretierer für Prolog: `prove (Goal)`

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

Beispiel:

```
app ([],L,L).
app ([E|R],L,[E|RL]) :- app (R,L,RL).

?- prove (app([1],[2],L)).
```

$\vdash$ `?- clause (app([1],[2],L),G), prove (G).`
$\vdash$ `?- prove (app([],[2],RL)).    % L = [1|RL]`
$\vdash$ `?- clause (app([],[2],RL),G1), prove (G1).`
$\vdash$ `?- prove (true).              % RL = [2]`
$\vdash \Box$; $L = [1, 2]$

Anmerkung:

- funktioniert so bei SWI-Prolog
- bei Sicstus Prolog: Definition aller Prädikate, auf die mit clause zugegriffen wird:
  :– dynamic $p_1/n_1, ..., p_k/n_k$

Erweitertet Meta-Interpretierer:
Berechnung der Beweislänge: `lprove (Goal, Length)`

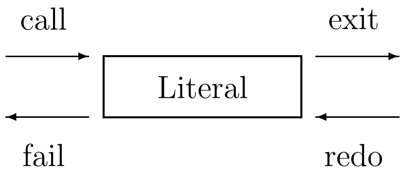
```
lprove (true,0).
lprove ( (A,B), L) :- lprove (A,LA), lprove (B,LB),
                      L is LA+LB.
lprove (A,L) :- clause (A,G), lprove (G,LG), L is LG+1.

?- lprove (app ([a,b,c],[d],[a,b,c,d]), L).
L = 4
```

Weitere Möglichkeiten:

- Ausgabe des Beweisverlaufs
- Interpretation mit Tiefenbeschränkung
- Vollständige Suche mit iterierter Tiefenbeschränkung
- Interpretation mit Lemma-Generierung

7.3 Debugging von Prolog-Programmen

Konzept: Jedes Literal, das bewiesen ist, entspricht einer *Box* (Box-Modell von *Byrd*):



Der *Debugger* zeigt jedes Betreten / Verlassen der Box an:

- call: erster Beweis des Literals
- redo: weitere Beweise
- exit: erfolgreicher Beweis
- fail: Fehlschlag

Beispiel:

```
p(a).
p(b).
q(b).
?- trace, p(X), q(X).
call:  p(_15)
exit:  p(a).
call:  p(a).
fail:  q(a).
redo:  p(
exit:  p(b).
call:  q(b).
exit:  q(b).
X=b
```

Tracer als Meta-Interpretierer:

```
tprove (true) :- !.
tprove ((A,B)) :- !, tprove (A), tprove (B).
tprove (A) :- callfail (A), clause (A,G),
              tprove (G), exitredo (A).
callfail (A) :- write ('call: '), write (A), nl.
callfail (A) :- write ('fail: '), write (A), nl, fail.
exitredo (A) :- write ('exit: '), write (A), nl.
exitredo (A) :- write ('redo: '), write (A), nl, fail.
```

8 Erweiterungen der reinen Logikprogrammierung

8.1 Flexible Berechnungsregeln

Probleme: bei links-rechts-Abarbeitung

- `not P`: warten, bis P variablenfrei
- Endlosschleifen:
  ```
  append ([],L,L).
  append ([E|R],L,[E|RL]) :- append (R,L,RL).
  ?- append ([1|V],W,L), append (L,X,[2|Y]).
  ```
  (das erste Literal hat unendlich viele Lösungen: $L = [1, \_, \_, \_, ...]$)
- Ineffiziente Berechnungen:
  ```
  sort (L,M) :- perm (L,M), ord (M).
  ```
  $\Rightarrow$ Generierung von $n!$ Permutationen ($n = |L|$)
- Laufzeitfehler bei Arithmetik:
  ```
  ?- X is Y+5, Y=3.
  ```
  $\rightarrow$ Error in arithmetic expression

Verzögerte Auswertung von Literalen (*when*-Deklaration):

```
?- append (X,Y,Z) when X or Z
```

(werte append nur aus, falls erstes oder drittes Argument an eine Struktur oder Konstante gebunden ist)
Auswirkung im append-Beispiel:

```
?- append([1|V],W,L), append (L,X,[2|Y]).
```

$\vdash_{\{L=[1|RL]\}}$ `?- append (V,W,RL), append ([1|RL],X,[2|Y]).`
(vorne V und RL beides Variablen; nicht auswerten!)
$\vdash$ fail

Effizienzverbesserung:

```
sort (L,M) :- ord (M), prem (L,M).
```

(ord(M): korrekte Lösungen, perm(M): potentielle Lösungen)

```
?- ord(X) when X.
?- sort([4,3,2,1],L).
```

$\vdash$ `?- ord(L), perm([4,3,2,1],L).`
$\vdash \ldots \vdash$ `?- ord([4,3|R]), perm([4,3,2,1],[4,3|R]).`
$\vdash$ fail, Backtracking

Konsequenz: Es werden nicht alle Permutationen erzeugt.

Statt „generate + test“ nun „test + generate“ und when-Deklaration

Vermeidung von Laufzeitfehlern:

```
?- X is E when ground(E).
?- X is Y+5, Y=3.
```

$\vdash$ `?- X is 3+5, Y=3.`
`X=8`

Flexible Berechnungsregeln:

- kein Standard (nicht implementiert in SWI)
- in vielen kommerziellen Systemen *unterschiedlich* realisiert
- Sicstus-Prolog: *Block-Deklarationen*
  `:- block append (-,?,-)`
  (an erster und dritter Stelle: ungebundene Variablen)
  $\Rightarrow$ blockiere Abarbeitung, falls erstes und drittes Argument ungebunden sind.
- Implementierung durch *Koroutinen*
- Bessere Beweiskontrolle durch automatische Transformation und Generierung von when-Deklarationen (NU-Prolog, *Naish* 1987)
- $\boxed{\text{Computation = Logic + Control}}$
  (Control: when-Deklarationen)
- *Andorra-Berechnungsmodell* (*Warren* 1987)
  Erst *deterministische* Literale auswerten (deterministisch $:\Longleftrightarrow$ höchstens eine Klausel anwendbar)
  1. Wende erst Resolutionsschritte auf alle deterministischen Literale an.
  2. Falls eine Anfrage keine deterministischen Literale enthält: Resolutionsschritt für linkes Literal

8.2 Constraints

Motivation:

```
?- X is 5+3.
X=8
?- 8 is Y+3.
error
```

Prolog-Arithmetik ist logisch unsauber.

Konsequenz: Prolog-Programme nur noch in bestimmten Modi ausführbar.
Lösung: Spezielle Verfahren zur Lösung von arithmetischen (Un)gleichungen

Arithmetische Constraints: (Un)gleichungen zwischen arithmetischen Ausdrücken ⇒ *Constraint Logic Programming*

Beispiel: Hypothekenberechnung
Parameter:

P: Kapital

T: Laufzeit

IR: monatlicher Zinssatz

B: Restbetrag

MP: monatliche Rückzahlung

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

Monatliche Rückzahlung:

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

Zeitdauer zur Finanzierung:

```
?- mortgage (100000,T,0.01,0,1400).
T = 125.901
```

Relation zwischen P,B,MP:

```
?- mortgage (P,150,0.01,B,MP).
P = 0.166783*B+83.3217*MP
```

CLP:

- Erweiterung der Logikprogrammierung
- Ersetze Terme durch constraint-Strukturen
- Ersetze Termunifikation durch Lösungsalgorithmen für Constraints.

Beispiel: CLP(R) (IBM, R für reelle Zahlen)

- Struktur: Terme + reelle Zahlen
- Constraints: (Un)gleichungen mit arithmetischen Ausdrücken
- Lösungsalgorithmus:
  - klassische Unifikation für Terme
  - Gauß-Elimination für Gleichungen
  - Simplex-Verfahren für Ungleichungen

Weitere Constraint-Strukturen:

- Boolesche Ausdrücke (A and (B or C))
  $\Rightarrow$ Hardware-Entwicklung und -verifikation
- unendliche zyklische Bäume ($\rightarrow$ Graphen)
- Listen
- endliche Bereiche $\leadsto$ Anwendungen in OR (Planungsaufgaben, z.B. Containerbeladung, Fertigung, Flughafenabfertigung, ...)

CLP(X): Rahmen für CLP, X beliebige aber *feste Constraint-Struktur*, bestehend aus

- Signatur Σ (Menge von Funktions- und Prädikatsymbolen)
- Struktur D über Σ (entspricht Interpretation, d.h. Wertemenge + entsprechende Prädikate und Funktionen)
- Klasse L von Formeln über Σ (Teilmenge der PL 1)
- Theorie T (Axiomatisierung der wahren Formeln aus L, d.h. $c \in L$ wahr gdw. $T \models c$

Übliche Einschränkungen:

1. Σ enthält das Prädikat = (Identität auf D)
2. Es gibt Constraints $\top, \bot \in L$ (top, bottom; entsprechen true und false)
3. L ist abgeschlossen unter Variablenumbenennung, Konjunktion, Existenzquantifizierung.

Klauseln über CLP(X): $p(\bar{t})$:– $c, p_1(\bar{t}_1), ..., p_k(\bar{t}_k)$
mit $\bar{t}, \bar{t}_i$ Folgen von Termen, $c \in L$
Anfragen über CLP(X): ?– $c, p_1(\bar{t}_1), ..., p_k(\bar{t}_k)$. ,
wobei $c \in L$ erfüllbar.
Resolutionsschritt mit $p_1(\bar{s})$:– $c', L_1, ..., L_m$ ergibt die Anfrage

$$?\!-\ c \wedge c' \wedge \bar{t}_1 = \bar{s}, L_1, ..., L_m, p_2(\bar{t}_2), ..., p_k(\bar{t}_k)$$

falls $c \wedge c' \wedge \bar{t}_1 = \bar{s}$ erfüllbar.

Wesentliche Ergebnisse (Jaffar / Lassez: Popl '87)
Standardresultate von Logikprogrammierung (Korrektheit und Vollständigkeit) gelten analog in CLP(X).
Notwendig: Algorithmus zur Entscheidung von Erfüllbarkeit in X (inkrementell: nicht in jedem Schritt wieder von vorne rechnen)
z.B. X=$I\!R$: inkrementelle Version von

- Gauß-Eliminationsverfahren
- Simplexverfahren

Bei „harten Constraints“: kein voller Erfüllbarkeitstest, sondern

- verzögerte Auswertung (z.B. nicht-lineare Constraints in CLP($I\!R$): $X * Y = 5$, warte, bis X oder Y einen festen Wert hat)
- nur lokale Konsistenzprüfung bei endlichen Domains (offensichtliche Widersprüche aufdecken) (CLP(FD)) (finite domains), da globale Konsistenzprüfung (Test auf Erfüllbarkeit) NP-vollständig → Van Heutenryck: Constraint Satisfaction in Logic Programming, MIT Press 1989

9 Implementierungen von Prolog

Möglichkeiten:

1. Interpretierer (portabel, ineffizient)
2. Übersetzer (effizient, weniger portabel)
3. Spezialhardware (möglicherweise effizient; Entwicklungszeit!, teuer

Meistverbreitet: Übersetzer
Keine direkte Übersetzung in Maschinencode sondern in eine abstrakte Maschine:

- orientiert an Prolog
- effizient implementierbar auf Standard-Hardware
- Zwischenebene der Übersetzung

„Standardmaschine“ für Prolog: *Warren's Abstract Machine* (*WAM* von D.H.D. Warren 1983; gute Beschreibung in Ait-Kaci, 1991)

Implementierung der WAM:

- Emulator in C (portabel, nicht sehr effizient)
- Übersetzung in C (portabel, gute Abstimmung mit Optimierung des C-Compilers nötig)
- Übersetzung in Maschinencode (effizient, aber nicht so portabel)

9.1 Grundideen der Warren Abstract Machine

Fasse Klauseln als Prozeduren auf

$$\underbrace{p(X,Y)}_{\text{Prozedurkopf}} \;\text{:-}\; \underbrace{q(X), r(Y,Z), s(X,Z)}_{\text{Prozedurrumpf}}$$

mit Parametern X, Y und lokaler Variable Z

Notation in imperativer Sprache:

```
procedure p(X,Y);
var Z;
begin
  call q(X);
  call r(Y,Z);
  call s(X,Z)
end
```

Alle Prozeduren sind rekursiv → Laufzeitstack zur Implementierung

- enthält lokale Variablen und Rücksprungadresse
- „push“ bei Prozeduraufruf
- „pop“ bei Prozedurende

Verfeinerung des vorigen Beispiels:

```
p(X,Y): allocate X,Y,Z   % push, Speicher fuer X,Y,Z
        call q(X)
        call r(Y,Z)
        call s(X,Z)
        deallocate       % pop
        return
```

Parameterübergabe:

- Reserviere hierfür *Argumentregister* $A_1, A_2, ...$
- bei Prozeduraufruf: i-tes aktuelles Argument steht in A_i
- vor Prozeduraufruf: Lade aktuelle Argumente in A_i
- zu Beginn des Prozedurrumpfes: Schreibe A_i in entsprechende Variablen

Neue Befehle:

- get V, A_i: speichere Wert von A_i in Variable V
- put V, A_i: speichere Wert von V in A_i

```
p/2: allocate X,Y,Z
     get X,A1
     get Y,A2
     put X,A1
     call q/1
     put Y,A1
     put Z,A2
     call r/2
     put X,A1
     put Z,A2
     call s/2
     deallocate
     return
```

Argumente können komplexe Terme sein.

- get T, A_i: Unifiziere Term T mit Term, der in A_i steht
- put T, A_i: Speichere Term T in A_i

Beispiel: $p(f(X), g(Y) :- q(h(g(X)))$.

```
p/2: allocate X,Y
     get f(X),A1
     get g(Y),A2
     put h(g(X)),A1
     call q/1
     deallocate
     return
```

Nachteil: In A_i stehen Terme (beliebig groß)
Lösung: Speichere alle Terme in speziellem Bereich, genannt *Heap*.
Register A_i und Variablen X, Y enthalten Verweise in den Heap.

Darstellung von Strukturen im Heap:

```
Funktor
1. Argument
...
n. Argument
```

($n + 1$ Speicherzellen; in Argumenten eventuell Verweise auf Unterstrukturen)

Optimierung: Variablen und Konstanten direkt in Argumenten speichern.
Notwendig: Markierung von Argumenten (Verweis? Konstante?)
WAM-Darstellung von Prolog-Termen: Paar, bestehend aus

- Etikett („tag")
- Wert

Etikett	Wert
reference	Adresse
structure	Adresse
list	Adresse der Liste
integer	Wert der Zahl
atom	Nummer des Atoms

Atome (und Funktoren) durch Compiler numerieren.
Listen: kein Abspeichern des Funktors →

```
1. Argument  (Listenkopf)
2. Argument  (Listenrest)
```

Variablen:

- gebundene Variablen: Verweis auf Bindung
- ungebundene Variablen: Verweis auf sich selbst

Beispiel: Darstellung von $f(g(X), X)$ auf Heap:

```
...
1001: str   9
1002: ref   8
...
1039: str   f
1040: ref   1001
1041: ref   1002
```

- get T, A_i: unifiziere T mit A_i; falls notwendig, speichere T im Heap ab
- put T, A_i: erzeuge Darstellung von T im Heap, setze A_i auf die entsprechende Heap-Adresse

Implementierung des Backtracking

Backtracking = Zurücksetzen auf früheren Berechnungszustand
Naive Lösung:

- Vor einem Resolutionsschritt, bei dem es mehrere Alternativen gibt, speichere aktuellen Maschinenzustand und Verweis auf Alternative.
- Im Failure-Fall: Zurückspeichern des Maschinenzustandes und Sprung zur Alternative
  → Stack für Maschinenzustände

Beispiel:

```
1)  p :- q,r.
2)  p :- s.
3)  q.
4)  q :- u.
?- p.
```

Backtrack-Stack vor Aufruf von r:

```
Zustand vor p, Alternative 2
Zustand vor q, Alternative 4
```

Nachteil: Sicherung eines Zustandes sehr aufwendig
Lösung: Statt den Heap vollständig im Backtrack-Stack zu speichern, protokolliere nur die Änderungen im Heap. Somit enthält der Backtrack-Stack nur einige Maschinenregister (z.B. Argumentregister $A_1, ...$) und Verweis auf nächste Alternative.

Neues Problem: Im Umgebungsstack stehen Umgebungen, die üblicherweise nach Verlassen der Prozedur (Klausel) gelöscht werden. Dadurch kann beim Backtracking eine passende Umgebung schon gelöscht worden sein:

Beispiel:

```
1)  p :- q(X),r(X).
2)  q(X) :- s(X).
3)  r(1).
4)  s(x) :- u(x).
5)  s(x) :- v(x).
6)  u(2).
7)  v(1).
?- p.
```

Stackzustände vor `u(x)` in (4):

Umgebung für (1)		Zustand vor s(x), Alternative (5)

Umgebung für (2)
Umgebung für (4)

Somit hat der Umgebungsstack vor Aufruf von v(2) die folgende Form:

Umgebung für (1)

Da der Beweis von v(2) fehlschlägt, muß der Zustand vor s(x) wiederhergestellt werden. Aber die Umgebung für die Klausel ist schon gelöscht worden.

Einfache Lösung: Speichere aktuellen Umgebungsstack im Backtracking-Stack.
Nachteil: Viele Umgebungen werden mehrfach abgespeichert.
Praktikable Lösung: Verschmelze Backtrack-Stack und Umgebungsstack zu einem Stack, so daß dieser sowohl Umgebungen als auch Backtrackpunkte enthält.

Beispiel: Stackzustand vor Aufruf von u(x):

Umgebung für (1)
Umgebung für (2)
Zustand vor u(x), Alternative (5)
Umgebung für (4)

Prinzip: Eine Umgebung wird nur dann gelöscht, wenn sich darüber kein Choicepoint befindet.

9.2 Die Warren Abstract Machine (WAM)

9.2.1 Speicherbereiche und Register der WAM

Die WAM enthält folgende Speicherbereiche:

- *Code-Bereich*: WAM-Instruktionen des auszuführenden Programmes
- *Stack-Umgebungen*:
  1. Felder für Variablen
  2. Verweis in den Rumpf einer anderen Klausel
  3. Verweis auf eine andere Umgebung
- *Stack*: Backtrackpunkte: Informationen, um früheren Maschinenzustand wiederherzustellen
- *Heap*: Enthält nur Terme, die bei der Unifikation entstehen
- *Trail*: Protokolliert Variablen, die bei der Unifikation gebunden werden und die beim Backtracking auf „ungebunden“ gesetzt werden müssen

- *PDL* (*Push Down List*): kleiner Stack zur Implementierung der Unifikation

Code-Bereich → | Heap → | Stack → | Trail → ... ← PDL

niedrige Adressen   hohe Adressen

Wichtig: Beim Schrumpfen des Stacks dürfen Variablenbindungen nicht in undefinierte Bereiche zeigen. Daher: bei einer Variablenbindung muß der Zeiger immer von der hohen zur niedrigen Adresse zeigen.

Register:

Bezeichner	Name
P	program pointer
CP	continuation program pointer
E	last environment
B	last backtrack point
TR	top of trail
H	top of heap
S	structure pointer (zu modifizierende Terme)
RW	read-write register
$A_1, A_2, ...$	argument register
$X_1, X_2, ...$	temporary variables

Eine Umgebung besteht aus folgenden Komponenten:

- ECP: letzter Wert von CP
- EE: letzter Wert von E
- $Y_1, Y_2, ...$: permanente Variablen

Ein Backtrackpunkt enthält folgende Komponenten:

- $BA_1, ..., BA_n$: gesicherte Werte der Argumentregister
- BCP: gesicherter Wert von CP
- BCE: gesicherter Wert von E
- BB: gesicherter Wert von B
- BTR: gesicherter Wert von TR
- H: gesicherter Wert von H
- BP: Verweis auf die nächste alternative Klausel

Definition (temporäre Variable)
Eine Variable in einer Klausel ist *temporär*, wenn gilt:

1. Die Variable kommt im Kopf der Klausel, in einer Struktur oder im letzten Literal der Klausel vor.
2. Die Variable kommt nicht in zwei verschiedenen Literalen des Rumpfes vor.
3. Wenn die Variable im Klauselkopf vorkommt, dann kommt sie höchstens noch im ersten Literal des Rumpfes vor.

Definition (permanente Variable)
Eine Variable ist *permanent*, wenn sie nicht temporär ist.

9.2.2 Instruktionssatz

Der *Instruktionssatz der WAM*:

Faktum	Rumpf (Literal)	Rumpf (mehrere Literale)
$p \leftarrow$	$p \leftarrow q$	$p \leftarrow q, r, s$
get-Instr. für p proceed	get-Instr. für p put-Instr. für q execute q	allocate get-Instr. für p put-Instr. für q call q,N put-Instr. für r call r,N1 % N1 $\leq$ N put-Instr. für s deallocate execute s

- **proceed**: am Ende eines Faktums: $P := CP$
  (Programmzähler bzw. Continuation Pointer)
- **execute L**: am Ende eines Rumpfs (mit einem Literal): $P := L$ ($\equiv$ Jump)
- **allocate**: erzeugt neue Umgebung:
  $Ce := E$
  $E :=$ *top_of_stack* % kann berechnet werden aus E,B und letztem Aufruf
  $E.ECP := CP$
  $E.EE := Ce$
- **deallocate**: gibt aktuelle Umgebung frei:
  $CP := E.ECP$
  $E := E.EE$
- **call Pr,N**: Aufruf eines Prädikats:
  CP :=<Codeadresse nach dieser Anweisung>
  $P := Pr$
  ($N = \#$ noch benötigter permanenter Variablen)
  $\boxed{\text{ECP}} \quad \leftarrow \text{E}$

EE
Y_1
...
Y_n
top_of_stack

Instruktionen zur Verknüpfung von Klauseln:
Seien $K_1, ..., K_m$ alle Klauseln für p/n.
Übersetzung:

`p/n: try_me_else L2,n` ($\leftarrow$ Erzeuge neuen Backtrack-Punkt, sichere Register $A_1, ..., A_n, CP, E, B, TR, H; BP := L2$)
        < Code für K_1 >
`L2:  retry_me_else L3` ($\leftarrow$ Ändere nächste Alternative: $B.BP := L3$)
        < Code für K_2 >
`L3:  retry_me_else L4`
    ...
`Lm:  trust_me_else fail` ($\leftarrow$ lösche aktuellen Backtrack-Punkt: $B := B.BB$)

Rückspeicherung aus dem Backtrack-Punkt: bei Fehlschlag (implizit in get-Instruktion)

Indizierung von Klauseln:

```
1)  append ([],L,L).
2)  append ([E|R],L,[E|RL]) :-  ...
               ↑
         Fallunterscheidung
```

```
append/3: switch_on_term L1,L1c,L1l,fail
L1:       try_me_else L2,3
L1c:      < Code fuer (1) >
L2:       trust_me_else fail
L2l:      < Code fuer (2) >
```

(L1c für Konstanten, L1l für Listen)

Zu `switch_on_term L1,L2,L3,L4` : Betrachte Argument A_1:

- freie Variable $\rightarrow P := L1$
- Konstante $\rightarrow P := L2$
- Liste $\rightarrow P := L3$
- Struktur $\rightarrow P := L4$

Weitere Indizierungsinstruktionen, Fallunterscheidung nach verschiedenen Konstanten

Verfeinerung der get/put-Instruktionen

Struktur der Argumente ist zur Compile-Zeit bekannt $\Rightarrow$ get/put-Instruktionen spezialisieren ($\rightarrow$ Literatur)

Beispiel: $p(g(X), f(a), X)$.
WAM-Code:

`p/3:`	`get_structure g/1,A1`	% unifiziere A_1 mit Strukturkopf $g/1$
	`unify_variable X1`	% unifiziere erstes Argument von A_1 mit der temporären Variable X_1
	`get_structure f/1,A2`	% unifiziere A_2 mit f
	`unify_constant a`	% unifiziere erstes Argument von A_2 mit a
	`get_value X1,A3`	% unifiziere A_3 mit Wert von X_1
	`proceed`	

Weitere Optimierungen:

- bessere Registerallokation
- Vermeidung von Variablenumspeicherung
- Sortierung permanenter Variablen, so daß Speicherplatz frühzeitig frei wird (vergleiche call $Pr, \underline{N}$) („environment trimming“)

Beispiel:

```
append ([],L,L).
append ([E|R],L,[E|RL]) :- append (R,L,RL).
```

Optimierter Code:

```
append/3: switch_on_term L1,L1c,L2l,fail
L1:       try_me_else L2,3
L1c:      get_nil A1
          get_value A2,A3   % Unif. von A2,A3
          % optimiert: nicht mehr in Hilfsvariablen laden
          proceed
L2:       trust_me_else fail
L2l:      get_list A1
          unify_variable X4   % X4=E
          unify_variable A1   % R=A1  (Optimierung!)
          get_list A3
          unify_value X4      % Unif.
          unify_variable A3   % RL=A3
             % Optimierung: bessere Register-Allokation
          execute append/3
```

Eigenschaften:

- kein Anlegen von Umgebungen
- kein rekursiver Aufruf (execute = jump) (tail recursion optimization)
- Aufbau von Backtrack-Punkten nur, falls ein Argument freie Variable
- ?– append($[1, 2, 3, ..., n], [....], L$). $\Rightarrow$ Abarbeitung wie bei while-Schleife
- Einzelne WAM-Instruktionen sind leicht implementierbar $\rightarrow$ MC-Code

9.3 Globale Analyse

Zur Compilezeit: Laufzeitinformationen ausnutzen: Modi von Prädikaten (Argumente innen gebunden oder frei) $\rightarrow$ Spezialisierung von get-Instruktionen
Typ von Argumenten: immer Listenstruktur
Problem: Laufzeitaspekte unentscheidbar.
Lösung: Approximation $\rightarrow$ abstrakte Interpretation.
Systeme: Aquarius-Prolog (Peter van Roy) $\rightarrow$ > 1.000.000 Regelschritte pro Sekunde; schneller als optimierender C-Compiler.

Index

=.., 55
Äquivalenz der deklarativen und operationellen Semantik, 40
Übersetzer, 19
äquivalent, 29

ableitbar, 36
Ableitungen
  Ein-Schritt-, 34
Abtrennungsregel, 10
Allabschluß, 35
allgemeines Resolutionsprinzip, 14
allgemeingültig, 26
allgemeinster Unifikator, 12
Allquantor, 23
Andorra-Berechnungsmodell, 62
Anfrage, 6, 10, 24
Anonyme Variable, 9
Antisymmetrie, 31
Antwort, 35
  S-berechnet, 37
Argumentregister, 67
Arithmetik, 52
Arithmetische Constraints, 63
arithmetischer Ausdruck, 52
Assoziativität, 8
Atome, 7
Aufzahlung des Suchbaums, 15
Ausdruck
  arithmetisch, 52
Auswahlregel, 35

Backtracking, 44
  Implementierung, 69
  unvollstandig, 45
Beweisen mit Resolutionsprinzip, 15
Block-Deklarationen, 62
bottom, 32
Box, 60
Breitendurchlauf, 42
Byrd, 60

call, 55
Clark's Completion, 48, 49
Clarke, 48
Closed World Assumption, 48
CLP, 63
CLP(X), 64
Code-Bereich, 70
Colmeraner, 3
Constraint, 53
Constraint Logic Programming, 63
Constraint-Struktur, 64
Constraints, 62
Coroutining, 51, 62
Cut, 46
  Fallunterscheidung, 47
CWA, 48

Datenbank, 56
Datenstrukturen, 20
Debugger, 60
Debugging, 46
Definition
  vervollstandigt, 49
deklarative Semantik, 35
Differenzieren
  symbolisch, 18
Differenzlisten, 57
disagreement set, 12
Disjunktion, 23
dynamische Semantik, 35

Effizienz
  Beweisverfahren, Hornklauseln, 31
  Cut, 46
Ein-Schritt-Ableitungen, 34
Elementare Programmiertechniken, 15
Elimination von Existenzquantoren, 29
erfüllbar, 26
Erfüllbarkeitsäquivalenz, 30
erfolgreiche SLD-Ableitung, 37
Erfolgszweig, 42
Existenzquantor, 23
Existenzquantoren

  Eliminierung (Skolem), 29

Fakten, 10
  Prolog, 5
Faktum, 24
Fallunterscheidung, 47
fehlerhafte Unifikation, 13
fehlgeschlagene SLD-Ableitung, 37
Fixpunktcharakterisierung des kleinsten Herbrand-Modelles, 35
Floundering, 49
Formel, 23
  geschlossen, 23
Formeln
  Transformation in Klauseln, 28
frei, 23
Funktionen
  als Relationen, 19
  Kennzeichnen mit Cut, 46
Funktor
  Prolog, 7

gebunden, 23, 45
gerichtet, 32
geschlossene Formel, 23
get0, 54
glb(A), 32
Gleichheit
  Prolog, 15
Gleichheitstheorie, 49
Globale Analyse, 75
größte untere Schranke, 32
größter Fixpunkt, 32
größtes Element, 32
greatest lower bound, 32
Grundinstanz, 34
Grundterm, 9

Heap, 67, 70
Herbrand-Basis, 27
Herbrand-Interpretation, 27
Herbrand-Modell, 28
  kleinstes, 33
Herbrand-Universum, 27
Homomorphismus, 11
Hornklausel, 24

Implementierung des Backtracking, 69
Implikation, 23
Inferenzregel, 48
Infixoperator, 8
instanziiert, 45
Instruktionssatz der WAM, 72
Interpretation, 25
  syntaktische: Herbrand-, 28
  Term-, 28
is, 52

Klammern, 8
Klausel, 6, 24
Kleene
  Satz von, 33
kleinste obere Schranke, 31
kleinster Fixpunkt, 32
kleinstes Element, 32
kleinstes Herbrand-Modell, 33
KNF, 30
Konjunktion, 23
Konjunktive Normalform, 30
Konstanten
  Prolog, 7
Konstruktion von Termen, 55
konstruktive Negation, 50
Kopf, 24
Koroutinen, 62
korrekt, 35
Korrektheit
  Unifikationsalgorithmus, 13
Kowalski, 3
  Satz von, 33

least upper bound, 31
leere Klausel, 24
Lifting-Lemma, 39
linksassoziativ, 8
linksrekursiv, 46
Listen, 7
  Prolog, 7
Literal, 10, 24
Lloyd, 50
Logikprogrammierung
  Prinzip der, 3

theoretische Grundlagen, 22
logisch äquivalent, 28, 29
logische Konsequenz, 26
logische Programme, 22
logisches Programm, 24
lub(A), 31

Manipulation der Datenbank, 56
map, 56
markierte Bäume, 40
Meta-Interpretierer, 59
mgu, 12
mgu-Lemma, 38
Mißerfolgszweig, 42
Modell, 26
Modus ponens, 10
monoton, 32
most general unifier, 12
Muster, 17

Naish, 62
Negation, 11, 23
verzogert, 51
Negation als Fehlschlag, 47
Negation as (finite failure), 48
NF, 48
Nichtdeterminismus, 11
Nichtterminierung, 11
nl, 54

obere Schranke, 31
Objekte
Prolog, 7
occur check, 13, 51
fehlerhafte Unifikation, 13
zyklische Terme, 13
Operationelle Semantik, 37
operationelle Semantik, 35
Operator
Infix-, 8
Postfix-, 8
Prafix-, 8
Operator T_P, 34
Operatoren
Prolog, 8
Ordinalzahlen, 33

partielle Ordnung, 31
Pattern, 17
PDL, 71
Pereira, 3, 19
permanent, 72
permanente Variable, 72
Postfixoperator, 8
Prädikat, 22
Prädikatenlogik erster Stufe, 22, 23
Pränex-Form, 29
Präzedenz, 8
Pradikat
Prolog, 6
Pradikatenlogik
erster Stufe, 6
Prafixoperator, 8
Programmiertechniken
elementare, 15
Programmklausel, 24
Prolog, 3
Anfrage, 6
Gleichheit, 15
Klausel, 6
Objekte, 7
Pradikat, 6
Regeln, 5
Relation, 6
Zahlen, Atome und Strukturen, 7
Prolog-Syntax, 4
Prologs Beweisstrategie, 45
prozedurale Semantik, 35
Push Down List, 71

quantifiziert, 23

read, 54
rechtsassoziativ, 8
Reflexivität, 31
Regeln, 10
in Prolog, 5
Register der WAM, 71
Reiter, 48
Relation
Prolog, 6
Resolutionsprinzip, 3, 44

- allgemeines, 14
- beweisen mit, 15
- einfach, 10
- einfaches, 10

Resolvente, 36
Robinson, 3, 12
- Unifikationssatz, 13

Roussel, 3
Rumpf, 24

S-berechnete Antwort, 37
Satz von Kleene, 33
Satz von Tarski, 32
Satz von van Emden und Kowalski, 33
Schranken, 31
Semantik
- deklarativ, 35
- deklarative, 33
- dynamische, 35
- operationelle, 35
- prozedurale, 35
- statisch, 35

Sichere SLD-Strategie, 42
Skolemform, 30
Skolemisierung, 29
SLD-Ableitung, 36
- erfolgreich, 37
- fehlgeschlagen, 37
- unendlich, 37

SLD-Baum, 40, 41
SLD-Resolution, 35
- Korrektheit, 37
- Vollständigkeit, 38, 40

SLD-Resolutionsschritt, 36
SLD-Strategie, 42
- sicher, 42

SLD-Widerlegung, 37
- uneingeschränkt, 37

SLDNF-Resolution, 49
Stack, 70
Stack-Umgebungen, 70
statische Semantik, 35
Stelligkeit, 23
stetig, 32
Strukturen
- Prolog, 7

strukturerhaltend, 11
Substitution, 11
Suchbaum, 15
Suchraum, 16
Suchregel, 42
symbolisches Differenzieren, 18
Syntax
- Prolog, 4

Tarski
- Satz von, 32

Term, 9, 22
Term-Interpretation, 28
Termgleichheit, 9
Terminierung
- Unifikationsalgorithmus, 13

Theorembeweiser, 31
theoretische Grundlagen der Logikprogrammierung, 22
Tiefendurchlauf, 43
top, 32
Tracer, 60
Trail, 70
Transformation in Pränex-Form, 29
Transformation von Formeln in Klauseln, 28
Transitivität, 31

Umkehrfunktion, 19
Umkehrrelation, 19
uneingeschränkte SLD-Widerlegung, 37
unendliche SLD-Ableitung, 37
unendlicher Zweig, 42
unentscheidbar
- Eigenschaft ‘logische Konsequenz’, 48
- Pradikatenlogik erster Stufe, 6

unerfüllbar, 26
Unerfüllbarkeitstests, 30
Unifikation, 10, 12
- fehlerhafte, 13

Unifikationsalgorithmus, 12
- Korrektheit, 13
- Terminierung, 13

Unifikationssatz, 38
Unifikationssatz von Robinson, 13
Unifikator, 12
    allgemeinster (mgu), 12
unifizierbar, 12
Universum, 25
Unstimmigkeitsmenge, 12
untere Schranke, 31
unvollstandig
    Backtracking, 45

van Emden
    Satz von, 33
var, 55
Variable
    anonyme, 9
    frei, 23
    gebunden, 23
    permanent (WAM), 72
    temporar (WAM), 71
Variablen, 5, 9
Variablenbelegung, 25
Variablennamen, 9
Verband, 32
Vergleich Prolog vs. imperative Sprachen, 6
Vier-Farben-Problem, 15
vollständiger Verband, 32
Vollständigkeit der SLD-Resolution, 38, 40
Vorkommenstest, 13, 51

wahr, 25
Wahrheit von Formeln, 25
WAM, 66
    Register, 70, 71
    Speicherbereiche, 70
WAM-Darstellung von Prolog-Termen, 68
Warren, 3, 19, 62
Warren's Abstract Machine, 66
Warren, D.H.D., 66
when, 61
Widerspruchsbeweis, 37
write, 53

Zahlen
    Prolog, 7
Zeichen
    Prolog, 7
Zerlegung von Termen, 55
zyklische Struktur, 51
zyklische Terme, 13