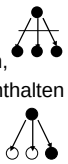
Lösungsbaum

Endlicher Teilbaum des Und-oder-Baums mit folgenden Eigenschaften:

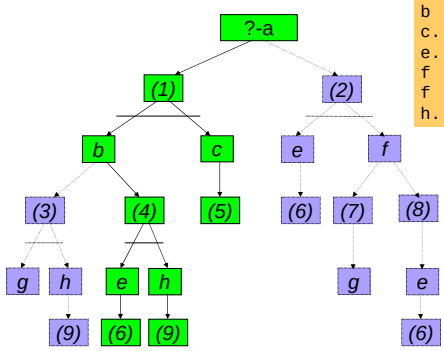
- Enthält nur lösbare Knoten,
- enthält Wurzelknoten,
- bei Und-Verzweigungen sind alle Nachfolger enthalten,
- bei Oder-Verzweigungen ist (genau) ein Nachfolger enthalten



Modell für „Beweisbaum“ in PROLOG

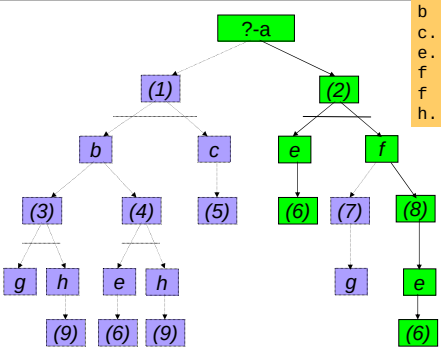
Lösungsbäume

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```

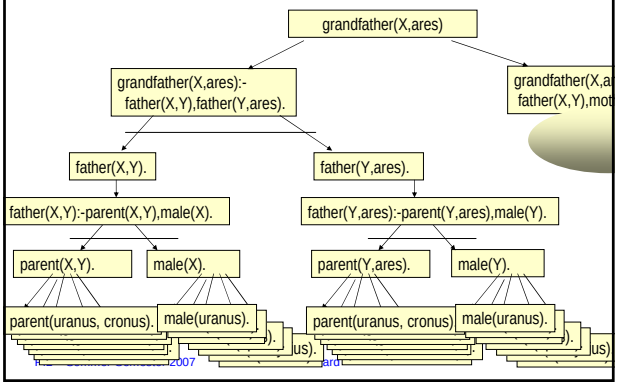


Lösungsbäume

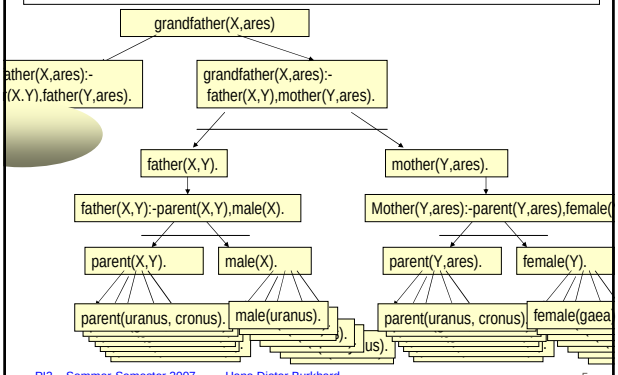
```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



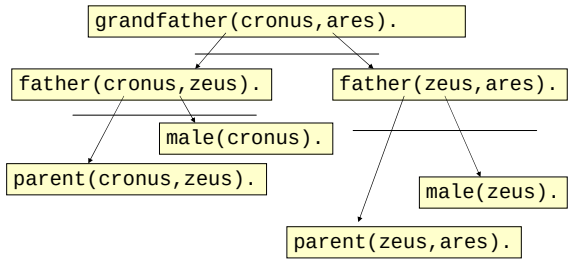
Und-oder-Baum modelliert Beweisversuche



Und-oder-Baum modelliert Beweisversuche



Lösungsbaum (Beweisb.) modelliert Beweis



Richtungen für Suchverfahren

Bottom-up/Backward-Suche:
Vom Ziel zum Anfang

Top-down/Forward-Suche
Vom Anfang zum Ziel

Bottom-up-Suche für Prolog ungeeignet

Modell für nicht-deterministische Suche

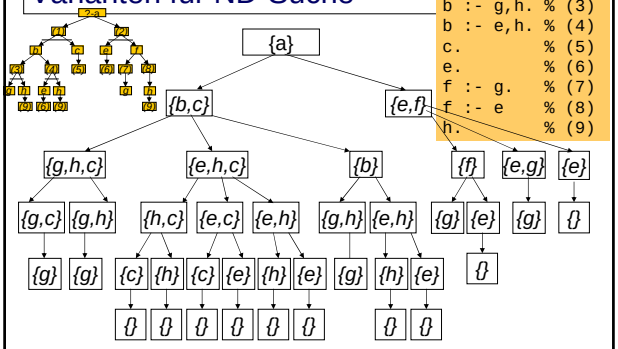
Vereinfachtes Schema (ohne Unifikation: „AK“)

1. subgoals=$\{g_1,...,g_n\}$
2. Falls subgoals = $\emptyset$: Erfolg.
3. Wähle $g \in$ subgoals .
4. Wähle Klausel k: $g :- g'_1,...,g'_m$ der Prozedur für g .
   Falls kein solches k existiert: Mißerfolg (des Versuchs).
5. subgoals := $(\{g_1,...,g_n\} - \{g\}) \cup \{g'_1,...,g'_m\}$.
   Weiter bei 2.

Alle Varianten (in 3 und 4) probieren,
falls keine zum Erfolg führt: „Nicht beweisbar“

Varianten für ND-Suche

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Alle Varianten (**jeden Weg von der Wurzel aus**) probieren,
falls keine zum Erfolg führt: „Nicht beweisbar“

Prolog-Interpreter

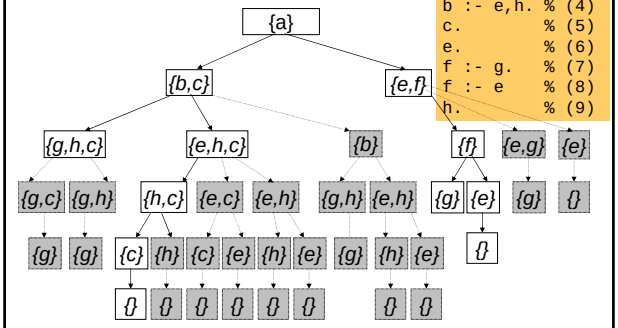
Einschränkung der Varianten

- Reihenfolge innerhalb einer Klausel (Und-Verzweigung)
  (alle subgoals müssen erfüllt werden)
  links vor rechts
- Reihenfolge innerhalb einer Prozedur (Oder-Verzweigung)
  (Alternativen für Beweis)
  oben vor unten

Zu zeigen wäre (Vollständigkeit):
Wenn Beweis existiert, dann auch schon hierbei.

Einschränkung der Varianten

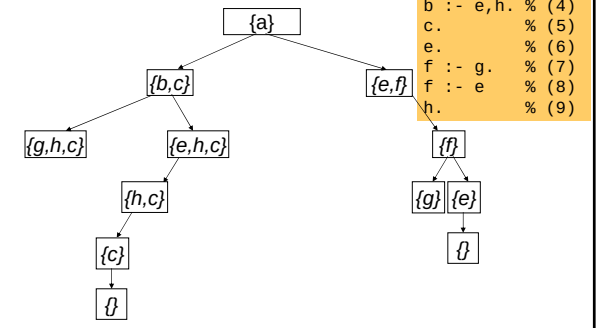
```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Subgoals sind alle zu beweisen,
Reihenfolge links vor rechts

Einschränkung der Varianten

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Subgoals sind alle zu beweisen,
Reihenfolge links vor rechts

Backtracking

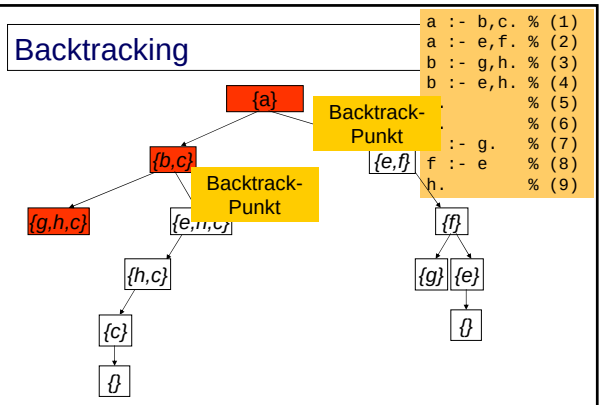
Effizienzgewinn

Suchpfade werden nicht vollständig von der Wurzel aus neu probiert, sondern nur stückweise.

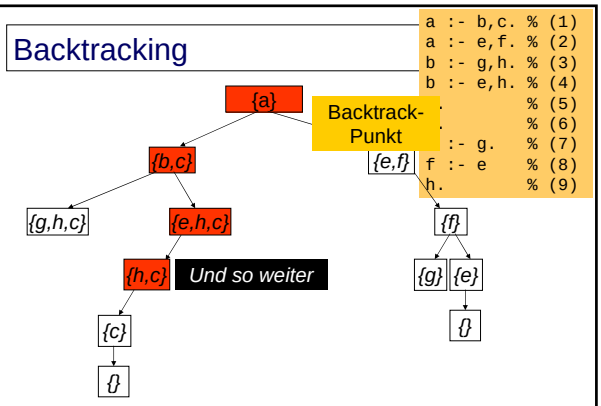
Bei Alternativen:
Einfügen eines „Backtrack-Punktes“ („Choicepoint“)

„Backtracking“:
Bei Fehlschlag am jüngsten „Backtrack-Punkt“ andere Alternative verfolgen

Backtracking



Backtracking



Modelle für Prolog-Suche

Vereinfachtes Schema (ohne Unifikation: „AK“)

1. subgoals=[g_1,...,g_n] . Liste
2. Falls subgoals = [ ] : Erfolg .
3. k sei nächste Klausel der Prozedur für g_1 :
   g_1 :- g'_1,...,g'_m .
   Falls kein solches k existiert: Backtracking.
   Falls kein Backtracking möglich: Misserfolg.
4. subgoals := [g'_1,...,g'_m ,g_2,...,g_n].
   Weiter bei 2.

Prolog-Interpreter

- Und-Verzweigung: links vor rechts
- Teilziele der Reihe nach vollständig abarbeiten

Verfolgen eines Zweiges in die Tiefe

- Oder-Verzweigung: oben vor unten

Linke Zweige zuerst

- Backtracking bei Fehlschlag:
  Rückkehr zu Alternative an oder-Verzweigung

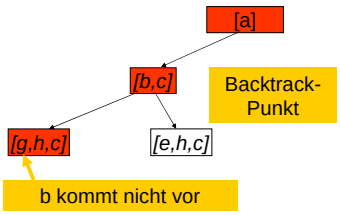
Nächster Zweig einer Oder-Verzweigung

Modelle für Prolog-Suche

1. subgoals=[g_1,...,g_n] .
2. Falls subgoals = [ ] : Erfolg .
3. k sei nächste Klausel der Prozedur für g_1 :
   g_1 :- g'_1,...,g'_m .
   Falls kein solches k existiert: Backtracking.
   Falls kein Backtracking möglich: Misserfolg.
4. subgoals := [g'_1,...,g'_m ,g_2,...,g_n].
   Weiter bei 2.

Problem: Backtrack-Punkt in subgoal –Liste nicht repräsentiert

Modelle für Prolog-Suche

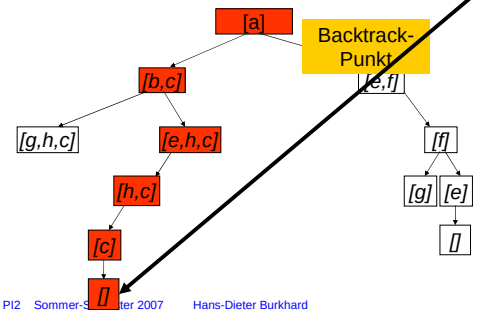


Problem: Backtrack-Punkt in subgoal –Liste nicht repräsentiert

Modelle für Prolog-Suche

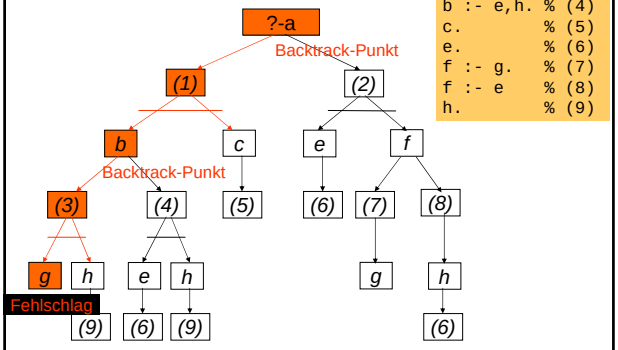
Quelle für Missverständnisse: *Warum?*

Bei erfolgreichem ersten Beweis ist subgoal-Liste leer



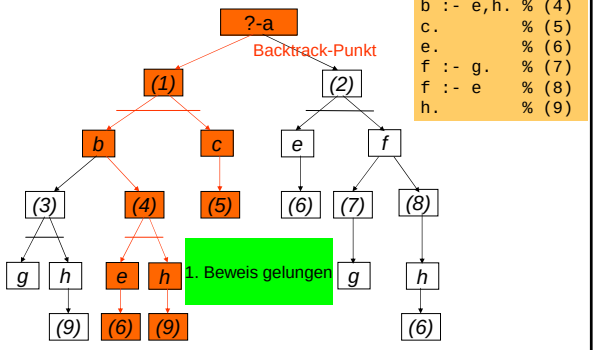
Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```

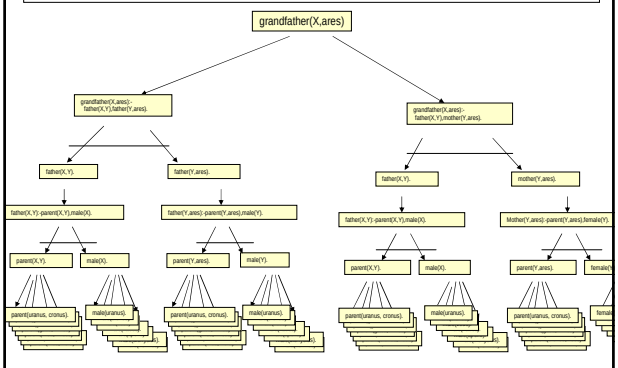


Abarbeitung im Und-oder-Baum

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Algorithmus für systematische Suche



Algorithmus für systematische Suche

PROCEDURE solve(unsolved_goals: GOALLIST);

goals = $[g_1,...,g_n]$ Bezeichnet Liste von goals g_i

top(goals) = g_1 Bezeichnet erstes Element

tail(goals) = $[g_2,...,g_n]$ Bezeichnet Rest-Liste

NIL = [ ] Bezeichnet leere Liste

concatenate($[g_1,...,g_n]$, $[g'_1,...,g'_m]$) = $[g_1,...,g_n , g'_1,...,g'_m]$

Bezeichnet Verkettung von Listen

Algorithmus für systematische Suche

PROCEDURE solve(unsolved_goals: GOALLIST);

unsolved_goals — Liste ungelöster subgoals

klauseln(g) — Klauseln der Prozedur für g

```
ackermann(o,N,s(N)).
ackermann(s(M),o,V):- ackermann(M,s(o),V).
ackermann(s(M),s(N),V):- ackermann(s(M),N,V1), ackermann(M,V1,V).
```

subgoals(k) — Subgoals der Klausel k

```
ackermann(s(M),s(N),V):- ackermann(s(M),N,V1), ackermann(M,V1,V).
```


Algorithmus für systematische Suche

```
PROCEDURE solve(unsolved_goals: GOALLIST);
                VAR k:KLAUSEL, g: GOAL;
BEGIN
  IF unsolved_goals = NIL THEN HALT(yes)
  ELSE g:= top(unsolved_goals);
    FORALL k ∈ klauseln(g) DO
                (* Klauseln für g nacheinander rekursiv probieren*)
      solve(concatenate(subgoals(k),tail(unsolved_goals)))
                (* subgoals der Klausel k weiter verfolgen *)
    END (*FORALL*)
  END (*IF*)
END solve;
```

Warum?

Backtrack-Punkte

Reihenfolge: oben vor unten

Reihenfolge: links vor rechts

Aufruf mit solve([goal]); HALT(no).

Algorithmus für systematische Suche

Rekursive Prozedur-Aufrufe mit Subgoal-Listen.

Bei leerer Subgoalliste:
- Resultat „yes“
- Abbruch der Prozedur-Kette.

Nach vollständiger Abarbeitung einer Aufruf-Kette Rückkehr zur jüngsten Möglichkeit gemäß FORALL ... (Backtracking).

Wenn alle (FORALL-)Varianten erfolglos versucht:
- Resultat „no“
- Prozedur-Ketten vollständig abgearbeitet.

Algorithmus für systematische Suche

Algorithmus verwendet eigentlich zwei Listen
(gemäß LIFO-Prinzip: Keller/stacks)

- Liste unsolved_goals (offene Teilziele)

- Liste der offenen Prozedur-Aufrufe (Prozedurkeller) mit Alternativen für Backtracking'

Betrachtung als verschachtelte Listen
$\mathcal{L} = [L_1,...,L_m]$
$= [[g_{11}, ..., g_{1n}] , ..., [g_{i1}, ..., g_{i,ni}], ... , [g_{m1}, ..., g_{m,nm}]]$

Algorithmus für systematische Suche

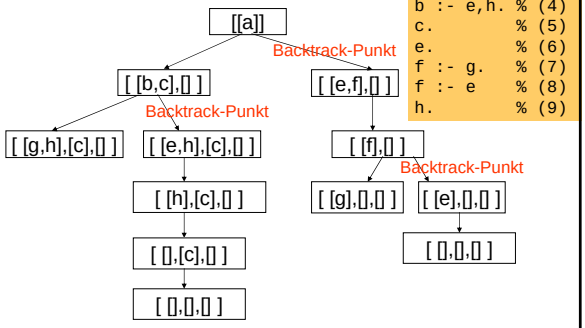
(0) (Start) $\mathcal{L} := [[Ausgangsproblem(e)]]$.
(1) Falls $\mathcal{L} = [[ ], ...,[ ]]$: EXIT(yes) .
(2) Sei L_i erste nicht-leere Subgoal-Liste aus $\mathcal{L} = [L_1,...,L_m]$.
Sei g_{i1} erstes Element aus $L_i = [g_{i1}, ..., g_{i,ni}]$:
- g_{i1} aus L_i entfernen: $L'_i := [g_{i2}, ..., g_{i,ni}]$.
- Falls keine Klauseln für g_{i1}existieren: weiter bei **(4)** .

(3) Sei k die nächste abzuarbeitende Klausel für g_{i1} .
Falls k Fakt: weiter bei **(1)** .
Falls k Regel: g_{i1} :- $g_1, ..., g_n$:
$\mathcal{L} := [[g_1, ..., g_n], L_1,.., L'_i,..., L_m]$.
Falls weitere Klauseln für g_{i1} existieren:
Backtrack-Punkt setzen. Weiter bei **(2) .**

(4) Backtracking: Rücksetzen zum jüngsten *Backtrack-Punkt* :
$\mathcal{L}$ zurücksetzen auf Stand vor *Backtrack-Punkt*, weiter bei **(3)** .
Falls kein *Backtrack-Punkt* existiert: $\mathcal{L} = [ ]$, EXIT(no).

Algorithmus für systematische Suche

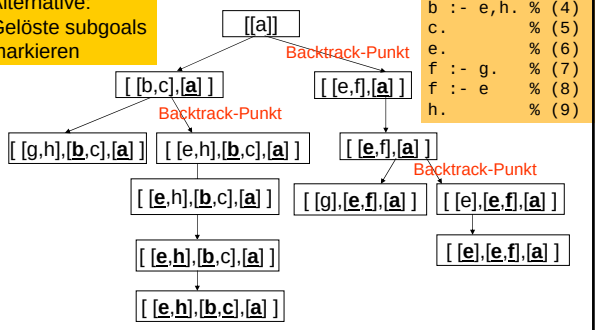
```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Algorithmus für systematische Suche

Alternative: Gelöste subgoals markieren

```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



Effizientere Implementation

Bekannt sind:

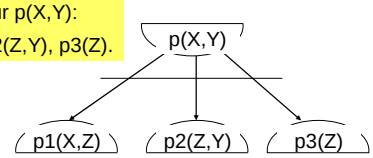
- Reihenfolge von Klauseln in Prozeduren
- Reihenfolge von subgoals in Klauseln

Zur Laufzeit nicht gesamte Listen speichern, sondern nur Referenz auf jeweils nächsten Eintrag

g_i statt $[g_i, ..., g_n]$,

Modell für Relationen (mit Variablen)

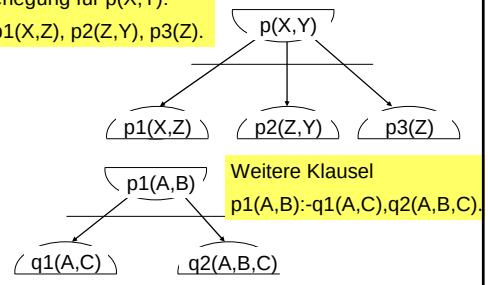
Problemzerlegung für p(X,Y):
p(X,Y) :- p1(X,Z), p2(Z,Y), p3(Z).



„Ferguson-Diagramm“

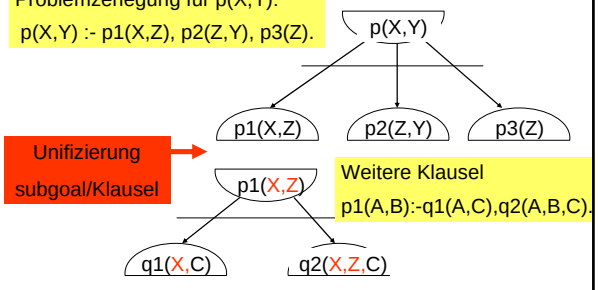
Modell für Relationen (mit Variablen)

Problemzerlegung für p(X,Y):
p(X,Y) :- p1(X,Z), p2(Z,Y), p3(Z).



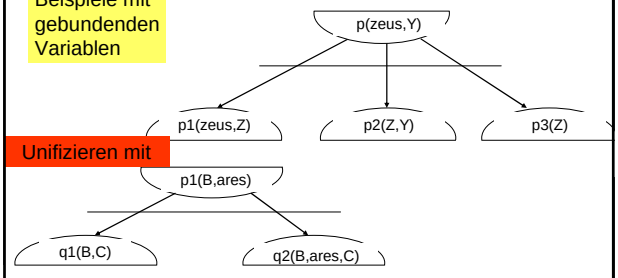
Modell für Relationen (mit Variablen)

Problemzerlegung für p(X,Y):
p(X,Y) :- p1(X,Z), p2(Z,Y), p3(Z).



Modell für Relationen (mit Variablen)

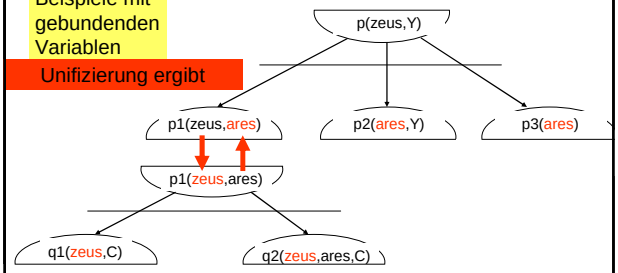
Beispiele mit gebundenden Variablen



Modell für Relationen (mit Variablen)

Beispiele mit gebundenden Variablen

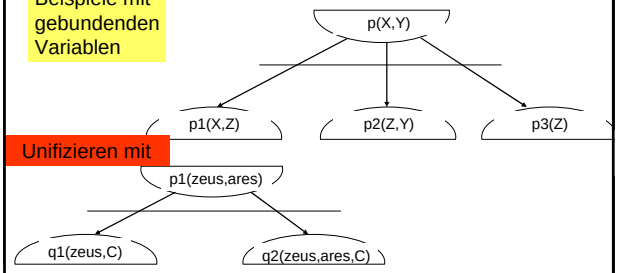
Unifizierung ergibt



Modell für Relationen (mit Variablen)

Beispiele mit gebundenden Variablen

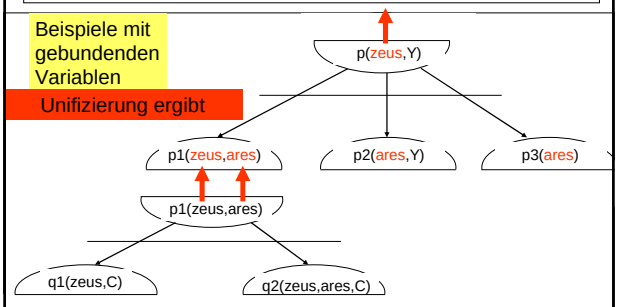
Unifizieren mit



Modell für Relationen (mit Variablen)

Beispiele mit gebundenden Variablen

Unifizierung ergibt



Algorithmus mit Variablen

```
PROCEDURE solve(unsolved_goals: GOALLIST);
            VAR k:KLAUSEL, g: GOAL;
BEGIN
 IF unsolved_goals = NIL THEN HALT(Result)
 ELSE g:= top(unsolved_goals);
   FORALL k ∈ klauseln(g) DO
            (* Klauseln für g nacheinander rekursiv probieren*)
     IF unify (g , head(k), σ) THEN
       solve(concatenate( σ(subgoals(k)), σ(tail(unsolved_goals)) ))
            (* subgoals der Klausel k weiter verfolgen *)
     END
   END (*FORALL*)
 END (*IF*)
END solve;
```

(für mehrfache Antworten: später)

σ ist die bei der Unifikation verwendete Substitution

Algorithmus mit Variablen

unify(g , head(k), σ) bewirkt

Prüfung auf Unifizierbarkeit von g und head(k)

Bindung von Variablen gemäß σ
durch Aktionen im Laufzeitsystem:

- Für die Klausel k wird Speicherplatz (frame) angelegt,
- darin Speicherplatz für Argumente (environment) mit Verweisen auf Bindungen.
- Beim Backtracking löschen der frames, die jünger als jüngster Backtrack-Punkt sind.

Algorithmus mit Variablen

Problem:
Effiziente Methode zur Bindung von Variablen beim Klauselaufruf und zum Auflösen von Bindungen beim Backtracking

Implementierungsidee:
Gegenseitige Referenzen der Variablen in Baumform

Anbindung an konstante/komplexe Strukturen an der Wurzel des Baumes

Dereferenzierung: Verfolgen einer Kette von Referenzen

Implementation: frames

Für jeden Klauselaufruf wird im Laufzeitsystem ein Speicherbereich reserviert: „frame“

Er enthält:

- Klausel einschließlich der Argumente.
  Effiziente Variante: „structure sharing“, in frame nur
  - Verweis auf template der Klausel
  - Argumente der Klausel („environment“).
- Verweis auf nächstes subgoal der aktuellen Klausel.
- Verweis auf Vater-Klausel der aktuellen Klausel.
- (Verweis auf jüngsten Backtrack-Punkt.)

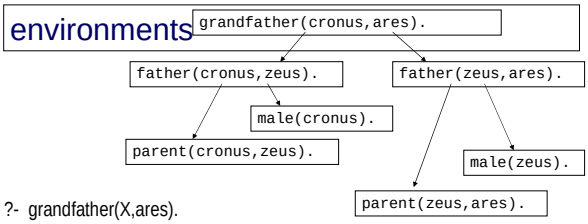
Implementation: frames

Frame enthält zusätzlich für Backtrack-Punkt :

- Nächste Klausel beim Backtracking.
- Davor liegender Backtrack-Punkt.
- Trail-Information (Protokoll der zu löschenden Variablenbindungen)

Dafür Reorganisation bei jedem Backtracking.

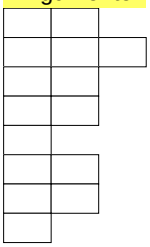
environments



?- grandfather(X,ares).
grandfather(X,ares):-father(X,Y),father(Y,ares).
father(X,Y):-parent(X,Y),male(X).
parent(cronus,zeus).
male(cronus).
father(zeus,ares):-parent(zeus,ares),male(zeus).
parent(zeus,ares).
male(zeus).

environments

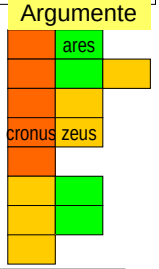
Argumente



?- grandfather(X,ares).
grandfather(X,ares):-father(X,Y),father(Y,ares).
father(X,Y):-parent(X,Y),male(X).
parent(cronus,zeus).
male(cronus).
father(zeus,ares):-parent(zeus,ares),male(zeus).
parent(zeus,ares).
male(zeus).

environments

Argumente

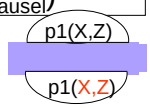


?- grandfather(X,ares).
grandfather(X,ares):-father(X,Y),father(Y,ares).
father(X,Y):-parent(X,Y),male(X).
parent(cronus,zeus).
male(cronus).
father(zeus,ares):-parent(zeus,ares),male(zeus).
parent(zeus,ares).
male(zeus).

Aneinandergebundene Variable in einer Klasse
Implementation als Baum von Referenzen
Endpunkte: referierter Term (z.B. in Konstanten-Tabelle)

Unify(Argument$^i_{goal}$,Argument$^i_{klausel}$)

T1 := Dereferenziere(Argument$^i_{goal}$)
T2 := Dereferenziere(Argument$^i_{klausel}$)



Unifikation erfolgreich, falls T1 und T2 unifizierbar.

Als „Seiteneffekte“ dabei Variablen binden:
Falls T1 und T2 Variable: T1 und T2 aneinander binden.
Prinzip der Rückwärtsreferenz:
Jüngere Variable an ältere Variable binden
Sonst: Variable Ti an den nicht-variablen Term binden
Falls jüngere Variable älter als jüngster Backtrack-Punkt:
Bindung im „trail“ protokollieren.

Backtracking

Rücksetzen der frames bis zum letzten Backtrack-Punkt.

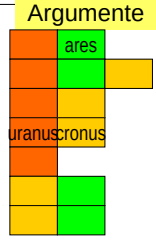
Dabei entfallen Bindungen der Variablen aus gelöschten Frames automatisch

Einige Bindungen für ältere Variablen müssen evtl. explizit gelöst werden: gemäß Protokoll im „trail“.

Fortsetzung mit der im Backtrack-Punkt als Alternative vorgemerkten nächsten Klausel

Backtracking

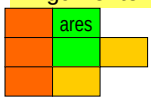
Argumente



?- grandfather(X,ares).
grandfather(X,ares):-father(X,Y),father(Y,ares).
father(X,Y):-parent(X,Y),male(X).
parent(uranus,cronus).
male(uranus).
father(cronus,ares):-parent(cronus,ares),male(cronus).
parent(cronus,ares). Fehlschlag

Backtracking

Argumente



?- grandfather(X,ares).
grandfather(X,ares):-father(X,Y),father(Y,ares).
father(X,Y):-parent(X,Y),male(X).

Algorithmus mit Variablen und mehrfachen Antworten

(0) (Start) $\mathcal{L}$: = [[Ausgangsproblem(e)]] .

(1) Falls $\mathcal{L}$ = [[ ], ...,[ ]] : REPORT(Result), weiter bei **(4)** .

(2) Sei L_i erste nicht-leere Subgoal-Liste aus $\mathcal{L}$ = [L_1,...,L_m] .
Sei g_{i1} erstes Element aus L_i = [g_{i1}, ..., $g_{i,ni}$] :
- g_{i1} aus L_i entfernen: L'_i := [g_{i2}, ..., $g_{i,ni}$] .
- Falls keine unifizierbaren Klauseln für g_{i1}: weiter bei **(4)** .

(3) Sei k die nächste unifizierbare Klausel für g_{i1} mit Unifikator: σ .
Falls k Fakt: weiter bei **(1)** .
Falls k Regel: g_{i1} :- g_1, ..., g_n :
$\mathcal{L}$:= σ([[g_1, ..., g_n], L_1,.., L'_i,..., L_m]) .
Falls weitere Klauseln für g_{i1} existieren: *Backtrack-Punkt* setzen
Weiter bei **(2)** .

(4) Backtracking: Rücksetzen zum jüngsten *Backtrack-Punkt* :
$\mathcal{L}$ zurücksetzen auf Stand vor *Backtrack-Punkt* , weiter bei **(3)** .
Falls kein *Backtrack-Punkt* existiert: EXIT(no).

Weitere Implementationsprobleme

Für Vorwärtsreferenzen bei komplexeren Strukturen.

Für Eingriffe in Beweisablauf (cut).

Effiziensteigerung (vorzeitige Speicherfreigabe), z.B.
- last call Optimierung („lco“)
- deterministische Klauseln („dco“)

Später mehr dazu

Prolog-Compiler:
Übersetzung in optimiertes Programm
(WAM = Warren abstract machine)