

Kapitel 09:

Fortgeschrittene

Logikprogrammierung

Quellen:

© Apt, Blackburn, Covington, Deville, Nilsson;
Vorlesungsunterlagen von FAU-Inf8 /
Beckstein [jetzt FSU Jena]

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik



Ausblick: Kapitel 09

- **Lernziele:**

- Wie funktionieren Unifikation und Backtracking in einem Prolog-System?
- Wie lässt sich Backtracking verkürzen?
- Was sind Differenzlisten und wie kann damit effizienter sortiert werden?
- Wie dient die Metaprogrammierung der Fehleranalyse, Effizienzsteigerung und Einführung neuer Sprachmittel in Prolog?
- Auf welche Weise lässt sich die Programmentwicklung systematisieren?

Inhalt

- Arbeitsweise eines Prolog-Systems
- Rekursion über Terme
- Ablaufsteuerung
 - Cut
 - Sortieren und Differenzlisten
- Metaprogrammierung
- Systematische Programmentwicklung
- Alternative Ansätze der Logikprogrammierung

Zur Geschichte der Logikprogrammierung

- 1965 J. A. Robinson: Resolution in Klausellogik.
- 1972 R. A. Kowalski: Programmiersprachliche Interpretation der Klausellogik.
- 1973 A. Colmerauer et al. implementieren erstes PROLOG-System.
- 1976 First International Workshop on Logic Programming. Imperial College, London.
- 1977 K. L. Clark: Negation and *Finite Failure*.
- 1981 W. F. Clocksin, C. S. Mellish: *Programming in Logic*.
- 1984 Journal of Logic Programming.
- 1984 J. W. Lloyd: *Theoretical foundations of Logic Programming*.

Arbeitsweise eines Prolog-Systems

- Beispiel:

- Klauseln:

(1) big(bear).

(2) big(elephant).

(3) small(cat).

(4) brown(bear) .

(5) black(cat).

(6) gray(elephant).

(7) dark(Z) :- black(Z).

(8) dark(Z) :- brown(Z).

- Anfrage:

`:- dark(X), big(X).`

Arbeitsweise eines Prolog-Systems

1. Initiales Ziel: `dark(X)`, `big(X)`.
2. Scannen des Programms von "oben nach unten" nach einer Klausel, deren Kopf sich mit dem ersten Teilziel `dark(X)` unifizieren lässt. Gefunden wird Klausel (7). ($X=Z1$)

Ersetzen des ersten Teilziels mit dem instantiierten Rumpf der Klausel (7). Damit neues Ziel: `black(Z1)`, `big(Z1)`.
3. Scannen des Programms, um eine Klausel zu finden, die sich mit `black(Z1)` unifizieren lässt. Gefunden wird Klausel (5): `black(cat)`. Diese Klausel hat einen leeren Rumpf.

Damit verkleinert sich das Ziel auf `big(cat)`. ($Z1=cat$)
4. Scannen des Programms, um `big(cat)` zu bearbeiten. Keine passende Klausel zu finden: Backtracking auslösen zu Schritt 3. Rückgängigmachen der Substitution $Z1=cat$ und Wiederherstellen des vorherigen Ziels: `black(Z1)`, `big(Z1)`.

Arbeitsweise eines Prolog-Systems

Weiterscannen des Programms ab Klausel (5). Keine passende Klausel zu finden: Backtracking zu Schritt 2 and weiterscannen unterhalb Klausel (7). Klausel (8) kann zur Unifikation herangezogen werden: $\text{dark}(Z) :- \text{brown}(Z). (X=Z2)$

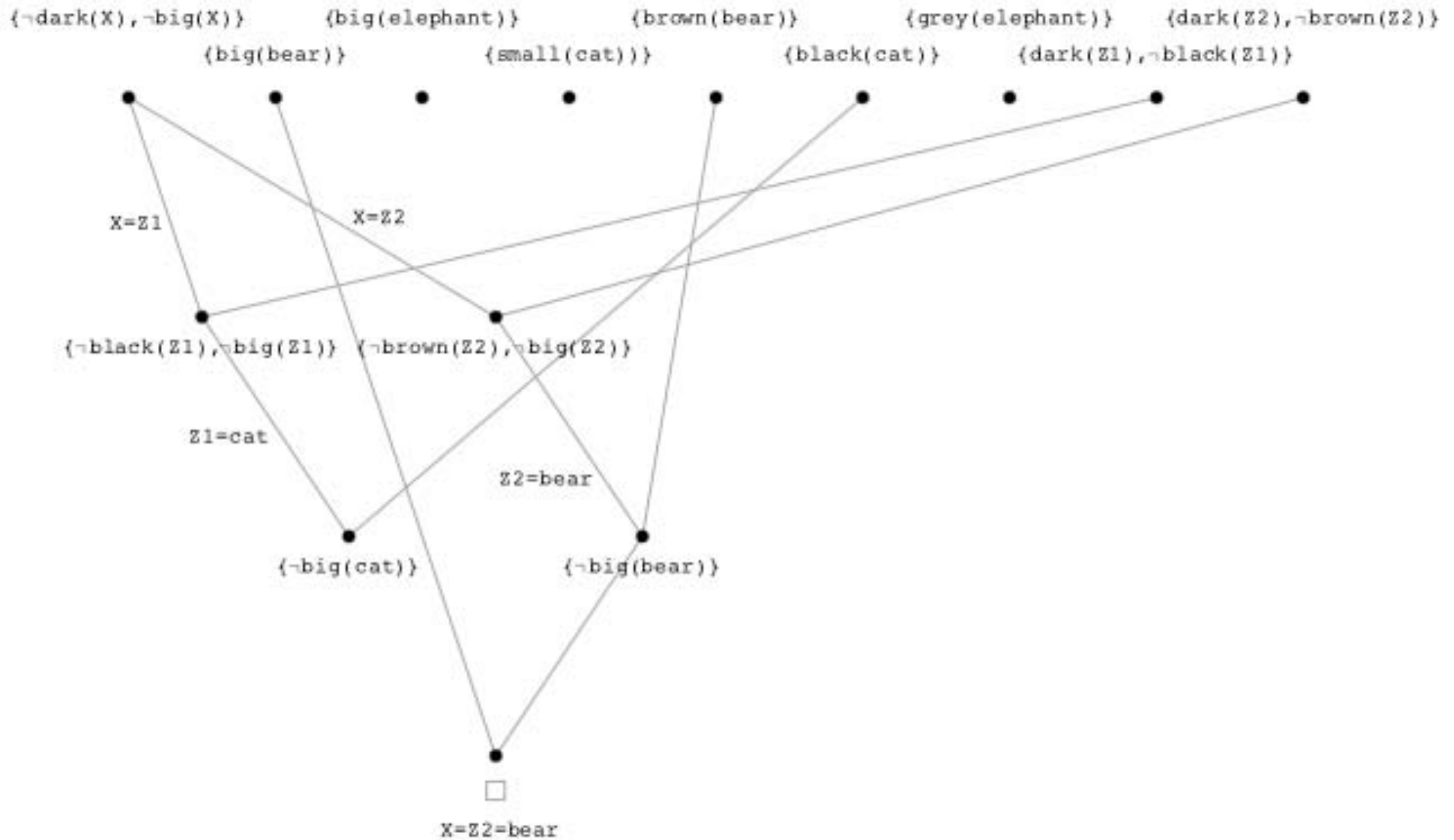
Ersetzen des ersten Teilziels durch $\text{brown}(Z2)$ ergibt neues Ziel $\text{brown}(Z2), \text{big}(Z2)$.

5. Scannen des Programms, um $\text{brown}(Z2)$ zu unifizieren. Finden der Klausel (4). Diese Klausel hat keinen Körper.

Das Ziel schrumpft zu $\text{big}(\text{bear}). (Z2=\text{bear})$

6. Scannen des Programms. Finden der Klausel (1). Diese Klausel hat keinen Körper. Ziel schrumpft zum leeren Ziel. Erfolgreiche Terminierung. Liefere korrespondierende Variablenbindung $X=Z2=\text{bear}$

Arbeitsweise eines Prolog-Systems



Unifikation + Backtracking

"arbeite Programm von links nach rechts ab"

Ziel: Z_1, \dots, Z_n

"Programmschritt":
ersetze Z_1
("Prozeduraufruf")
durch die Ziele

$B_{1,1}^i, \dots, B_{1,n_i}^i$
("Prozedurrumpf")

"passende
Prozedur
gefunden"
unifizierbar mit σ_1

$(B_{1,1}^i, \dots, B_{1,n_i}^i, Z_2, \dots, Z_n) \sigma_1$

Backtracking bei Fehlschlag
eines Teilziels

"falsche Prozedur wurde
aufgerufen, suche andere!"

Prolog-Programm

```

K1.
K2 :- B12.
.
.
.
.
.
Ki :- B1i, ..., Bnii.
.
.
.
.
.
Km :- B1m, ..., Bnmm.
    
```

"suche
Prozedur":
Durchsuchen
von oben nach
unten.

• • • □

Unifikation + Backtracking

- Terminierung:
 - wenn Ziel abgearbeitet (leere Klausel)
Ergebnis: yes + Antwortsustitution
 - wenn keine passende Prozedur gefunden werden kann
Ergebnis: no
 - keine Terminierung, wenn das Programm in einen unendlichen Pfad läuft ("Endlosschleife")

Unifikation: "Occurs Check"

- Aus Effizienzgründen wird in Prolog normalerweise der **occurs-check** weggelassen. (Logische Konsequenz: $\forall\Lambda$ und $\Lambda\forall$ ununterscheidbar.)
- Dies führt zu zirkulären Strukturen, wenn man eine Variable mit einer Struktur unifiziert, die diese Variable enthält. Beim Traversieren solcher Strukturen (z.B. für die Ausgabe) entstehen Endlosschleifen:

```
?- X = f(X).  
X = f(f(f(f(f(f(f(f(...))))))))))  
Yes  
?- X=[a,b,X].  
X = [a, b, [a, b, [a, b, [...|...]]]]  
Yes
```

- **unify_with_occurs_check/2** in ISO Prolog:

```
?- unify_with_occurs_check(X,f(X)).  
No  
?- unify_with_occurs_check(X,[a,b,X]).  
No
```

Rekursion über Terme: Symbolische Ableitung

```
nat(0).
nat(s(N)) :- nat(N).

diff(X,X,s(0)).
diff(X^s(N),X,s(N)*X^N).
diff(sin(X),X,cos(X)).
diff(cos(X),X,-sin(X)).
diff(exp(X),X,exp(X)).
diff(log(X),X,s(0)/X).
diff(N,X,0) :- nat(N).
```

```
diff(F+G,X,DF+DG) :-
    diff(F,X,DF),
    diff(G,X,DG).
diff(F-G,X,DF-DG) :-
    diff(F,X,DF),
    diff(G,X,DG).
diff(F*G,X,F*DG+DF*G) :-
    diff(F,X,DF),
    diff(G,X,DG).
diff(F/G,X,(G*DF-F*DG)/(G*G)) :-
    diff(F,X,DF),
    diff(G,X,DG).
diff(FU,X,DF*DU) :-
    FU =.. [F,U],
    diff(FU,U,DF),
    diff(U,X,DU).
```

**"Strukturelle
Induktion"**

Rekursion über Terme: Symbolische Ableitung

?- diff($x^{s(s(0))}$, x , Deriv).

$$\text{Deriv} = s(s(0)) * x^{s(0)}$$

?- diff($s(s(s(0))) * x^{s(s(0))} + s(s(0)) * x$, x , Deriv).

$$\text{Deriv} = s(s(s(0))) * (s(s(0)) * x^{s(0)}) + 0 * x^{s(s(0))} + (s(s(0))) * s(0) + 0 * x$$

?- diff($\sin(\cos(x))$, x , Deriv).

$$\text{Deriv} = \cos(\cos(x)) * -\sin(x)$$

Ablaufsteuerung: Bedingte Ausführung

- In Prolog üblicherweise durch alternative Definitionen von Prozeduren anstelle von if- oder case-Anweisungen.
- Aber: Sei folgendes Programm gegeben

R1: $f(X, 0) :- X < 3.$

R2: $f(X, 2) :- 3 \leq X, X < 6.$

R3: $f(X, 4) :- 6 \leq X.$

- Wir betrachten nun die Anfrage: $?- f(1, Y), 2 < Y.$

1. Das erste Teilziel kann erfüllt werden mit $Y/0.$

2. Dadurch wird das zweite Teilziel instantiiert zu $2 < 0$, es schlägt damit fehl

=> Backtracking zu **R2, R3.**

Aber: R_i schließen sich gegenseitig aus. Nur jeweils ein R_i kann erfüllt sein

=> **Backtracking** hier nutzlos!

Ablaufsteuerung: Cut

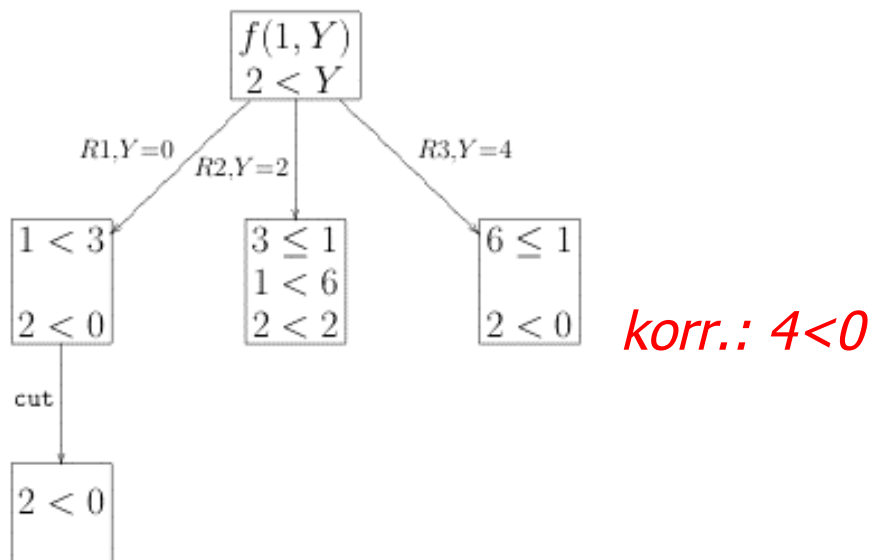
- Einführung eines neuen Operators Cut (!), der Backtracking verhindert:

R1: $f(X, 0) :- X < 3, !.$

R2: $f(X, 2) :- 3 \leq X, X < 6, !.$

R3: $f(X, 4) :- 6 \leq X.$

- **Neuer Suchgraph zur Anfrage** $?- f(1, Y), 2 < Y$



Ablaufsteuerung: Cut

- Weitere Optimierung:

Die Anfrage: $?- f(X, Y)$. liefert mit diesem Programm $Y = 4$ als Wert:

R1: $X < 3$ schlägt fehl (cut nicht erreicht).

R2: $3 \leq X$ erfolgreich, aber $X < 6$ schlägt fehl,
(cut nicht erreicht).

R3: $6 \leq X$ erfolgreich.

Ablaufsteuerung: Cut

Effizientere Formulierung:

if $X < 3$ then $Y = 0$,
otherwise if $X < 6$ then $Y = 2$,
otherwise $Y = 4$.

In Prolog:

R1': $f(X, 0) :- X < 3, !.$
R2': $f(X, 2) :- X < 6, !.$
R3': $f(X, 4).$

Achtung: Lässt man diese cuts fort, ändert sich die deklarative Bedeutung des Programms: Die Anfrage $?- f(1, Y).$ liefert hintereinander alle Lösungen:

$Y = 0, Y = 2, Y = 4.$

Ablaufsteuerung: Cut

- **Präzise Formulierung:** Sei A eine Programmklausel der Form

$$A \leftarrow B_1, \dots, B_k, !, B_{k+2}, \dots, B_n$$

und sei A durch das (Teil-)ziel G aktiviert. G heißt dann **Elternziel**.

- Wird bei der Abarbeitung der Cut (!) erreicht, dann wurden bereits Lösungen für die Teilziele B_1, \dots, B_k gefunden.
- Durch Ausführen des Teilziels ! wird diese Lösung **eingefroren**; alternative Lösungen für A werden nicht mehr betrachtet ("**prune**"):

Ablaufsteuerung: Cut

- D.h., andere in der Berechnung von B_i , $i \leq k$, verbleibende Wahlmöglichkeiten werden aus dem Suchbaum eliminiert (Löschen "nutzloser" Zweige).
- Falls B_i für $i > k$ scheitert, geht Backtracking nur bis zum ! zurück.
- Wenn Backtracking tatsächlich den Cut erreicht, scheitert der Cut.
- Zudem ist das Teilziel G auf die Klausel A festgelegt ("**commit**"); alle Versuche, G mit alternativen ProgrammklauseIn zu unifizieren, sind ausgeschlossen.

Ablaufsteuerung: Cut

- Beispiel:

$C :- P, Q, R, !, S, T, U.$

$C :- V.$

$A :- B, C, D.$

$?- A.$

- Der cut-Operator beeinflusst die Abarbeitung der Klausel C wie folgt:
 - **Backtracking** ist innerhalb der Teilziele P, Q und R möglich.
 - Beim Erreichen des cut-Operators werden alternative Lösungen für P, Q und R ausgeschlossen.
 - Die alternative Klausel zu C,

$C :- V.$

wird ebenso ignoriert.

Ablaufsteuerung: Cut

- **Backtracking** ist weiterhin möglich zum Erfüllen der Teilziele S, T, U.
- Das **Elternziel** der Klausel, die den cut enthält, ist die Klausel

$A :- B, C, D.$

- Der cut-Operator hat aber nur lokale Auswirkungen bei der Ausführung des Ziels C. Er ist von A aus **unsichtbar**.

Backtracking zum Erfüllen der Teilziele B, C, D erfolgt unabhängig davon, ob in diesen Teilzielen cut-Operatoren angewendet werden.

Grüne und rote Cuts

- Cuts, die verwendet werden, um Determinismus auszudrücken – d.h., dass für jedes anwendbare Ziel nur eine der Klauseln erfolgreich verwendet werden kann, um es zu beweisen – heißen "**grüne Cuts**".
⇒ Einsparung von Zeit und Speicherplatz.
 - Sie erhalten die deklarative Semantik; werden sie entfernt, liefert Prolog identische Ergebnisse, nur etwas ineffizienter (Grüne Cuts können überlesen werden).
- Wenn Cuts verwendet werden, um explizite Bedingungen wegzulassen, spricht man von "**roten Cuts**".

Grüne und rote Cuts

- **Schlechtes Beispiel:**

```
minimum(X,Y,X) :- X =< Y, !.  
minimum(X,Y,Y).
```

Was der Autor ausdrücken wollte: Wenn $X \leq Y$, dann ist das Minimum X . Andernfalls ist das Minimum Y und ein weiterer Vergleich zwischen X und Y ist überflüssig.

Aber: Dieses Programm ist erfolgreich mit dem Ziel `minimum(2,5,5)` !

- Eine logisch korrekte Definition des minimum-Programms:

```
minimum(X,Y,X) :- X =< Y, !.  
minimum(X,Y,Y) :- X > Y, !.
```

- Rote Cuts sind gefährlich und nicht semantikerhaltend.
Programme mit roten Cuts sind schwer lesbar.

Grüne und rote Cuts

- Cut-Beispiele:

$$p :- a, b.$$

$$p :- c.$$

$$p \Leftrightarrow (a \wedge b) \vee c$$

$$p :- a, !, b.$$

$$p :- c.$$

$$p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$$

$$p :- c.$$

$$p :- a, b.$$

$$p \Leftrightarrow c \vee (a \wedge b)$$

$$p :- c.$$

$$p :- a, !, b.$$

$$p \Leftrightarrow c \vee (a \wedge b)$$

Listen-Verarbeitung mit Cut

member-Prädikat, das nur das erste Vorkommen findet:

```
member(X,[X|L]) :- !.  
member(X,[Y|L]) :- member(X,L).
```

```
?- member(X,[a,b,c]).  
X = a ;  
no
```

add fügt nur Elemente in eine Liste ein, wenn diese noch nicht in ihr vorkommen:

```
add(X,L,L) :- member(X,L), !.  
add(X,L,[X|L]).
```

```
?- add(a,[b,c],L).  
L = [a,b,c]
```

```
?- add(X,[b,c],L).  
L = [b,c]  
X = b
```

```
?- add(a,[b,c,X],L).  
L = [b,c,a]  
X = a
```

Cut, Negation und die "Closed World Assumption"

- **Zur Erinnerung:** \neg ("Negative Ziele").

"Negation as Failure" ergibt mit der "Closed World Assumption" (CWA) eine eingeschränkte Form der logischen Negation (\neg deren Definition nicht-Horn ist).

- Die CWA besagt: $P \not\vdash A \Rightarrow \neg A$

Nicht-Beweisbarkeit ist im allgemeinen Fall unentscheidbar.

- NAF (Negation As [finite] Failure) ist schwächere Version der CWA:

$\leftarrow A$ hat einen endlich gescheiterten SLD-Baum $\Rightarrow \neg A$

- **Anmerkung:** Beide sind unkorrekt, da $\neg A$ keine **logische Folgerung** der (definiten) Programme sein kann. Denn die Herbrand-Basis, in der alle variablenfreien atomaren Formeln wahr sind, ist ein Modell des Programms.

Cut, Negation und die "Closed World Assumption"

- Mithilfe des Meta-Prädikats `call/1` und des Cut kann nunmehr ein derartiges not-Prädikat definiert werden:

```
not(P) :- call(P), !, fail.  
not(P).
```

- Sei P ein Ziel, z.B. `student(...)` – keine Variable!
 - In der ersten Klausel wird ein Beweis von P erzwungen; ist er erfolgreich, so wird der Cut überschritten und `fail` erreicht, d.h. `not(P)` scheitert.
 - Ist P nicht erfolgreich, gilt mit der zweiten Klausel `not(P)`.
- Bemerkung: `:- call(G)` ist erfolgreich, wenn `:- G` erfolgreich ist.

Sortieren und Differenzlisten

- Allgemeine Spezifikation des Sortierproblems:
 $\text{sort}(\text{Old}, \text{New}) :- \text{perm}(\text{Old}, \text{New}), \text{ordered}(\text{New}).$
wobei Old und New Listen seien.
- Wie kommt man von dieser hohen Abstraktionsebene zu einer konkreten Spezifikation?
 - Erster Schritt: Definition der Prädikate perm für Permutation und ordered für Ordnung.
 - Dabei wird zunächst rein deklarativ vorgegangen, d.h. ohne besondere Berücksichtigung der Ablaufsteuerung ("control").
 - Für eine effiziente Verarbeitung wäre jedoch eine prozedurale Verzahnung der beiden Prädikate notwendig!

Sortieren und Differenzlisten

```
mysort(Old, New) :- perm(Old, New), ordered(New).
```

```
perm([], []).
```

```
perm(L, [X|Y]) :- delete(X, L, L2), perm(L2, Y).
```

```
delete(X, [X|L], L).
```

```
delete(X, [Y|L], [Y|Z]) :- delete(X, L, Z).
```

% o.B.d.A. numerische Ordnung

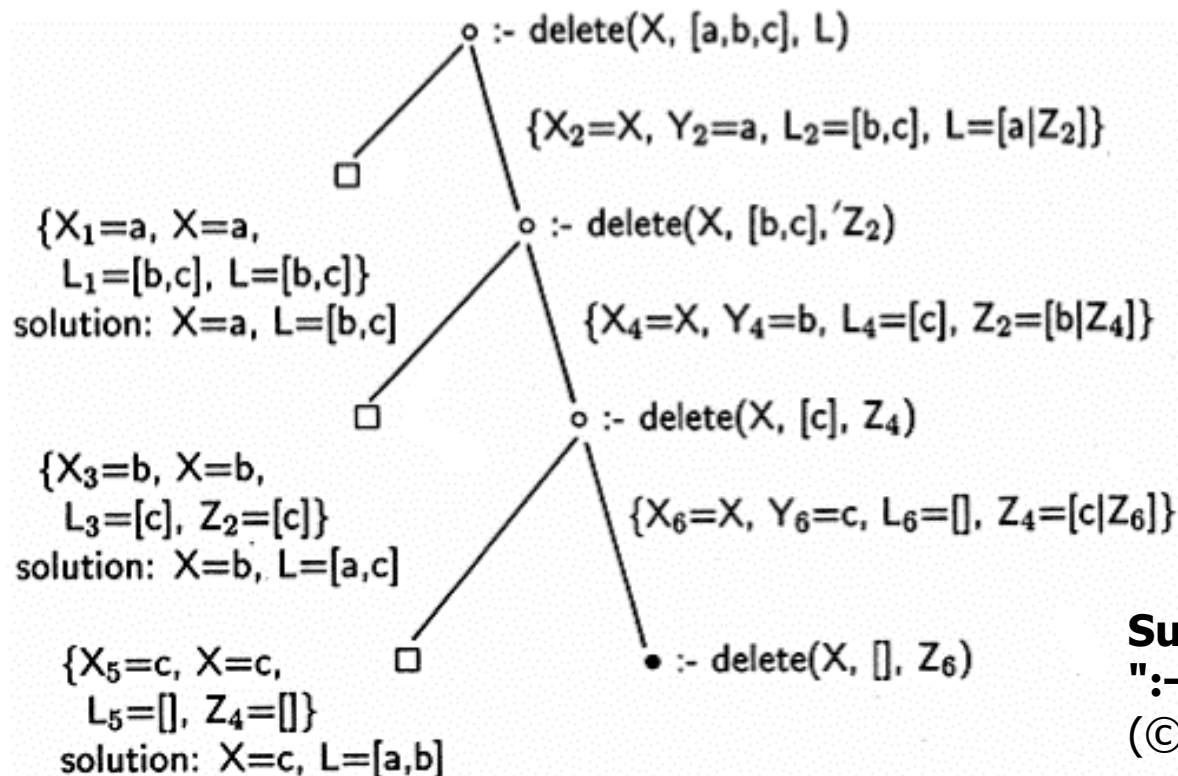
```
ordered([]).
```

```
ordered([X]).
```

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

Sortieren und Differenzlisten

- Wegen des inhärenten Nichtdeterminismus können bei der Konstruktion einer Permutation die Elemente der ursprünglichen Liste in beliebiger Reihenfolge gelöscht werden. Welches Element gelöscht wird, hängt jedesmal von der Auswahl einer der beiden delete-Klauseln ab.



Suchraum für
" :- delete(X , [a, b, c], L). "
 (©1985 IEEE)

Sortieren und Differenzlisten

- Erfüllt man z.B. die Bedingung $\text{delete}(X, L, L_2)$ durch Verwendung der ersten Klausel, wird X an den Kopf und L_2 an den Rest von L gebunden.
- Bei Wahl der zweiten Klausel und danach der ersten würde das ursprüngliche X an das zweite Element von L gebunden und L_2 an den Rest von L ohne das zweite Element.
- Ein effizienteres Sortierverfahren realisiert der Quicksort-Algorithmus von C.A.R. Hoare (geb. 1934):
 - Wähle ein beliebiges Listenelement X als Pivot.
 - Teile die restlichen Elemente in zwei Listen, eine für Elemente kleiner als X und die andere für Elemente größer als X .
 - Sortiere jede der beiden Listen rekursiv.
 - Ergebnis ist die Verkettung der beiden sortierten Listen mit dem Pivot-Element X in der Mitte.



Quicksort

```
quicksort([], []).
quicksort([X|Xs], Ys) :-
    !,
    partition(X, Xs, Littles, Bigs),
    quicksort(Littles, Ls),
    quicksort(Bigs, Bs),
    append(Ls, [X|Bs], Ys).
```

```
partition(_, [], [], []). % Empty list
partition(X, [Y | Xs], [Y | Ls], Bs) :-
    X > Y,
    !,
    partition(X, Xs, Ls, Bs).
partition(X, [Y | Xs], Ls, [Y | Bs]) :-
    X =< Y,
    !,
    partition(X, Xs, Ls, Bs).
```

Differenzlisten

- append ist ziemlich ineffizient. quicksort kann stattdessen mit einer Akkumulatorvariablen effizienter implementiert werden (s. z.B. Apt, K. 10; Covington, K. 7).
- Von der Verwendung einer Akkumulatorvariablen für Listen ist nur ein kurzer Weg zu einer alternativen Darstellung für Listen, den sog. **Differenzlisten**:
 - Verallgemeinerung des Listenkonzepts, das die Konkatenation mit konstantem Zeitaufwand erlaubt;
 - append benötigt hingegen eine zur Länge der ersten Liste proportionale Zeit.
- Eine Differenzliste ist ein Konstrukt der Form

$$[a_1, \dots, a_n \mid X] - X,$$

das die Liste $[a_1, \dots, a_n]$ in einer für Verkettung effizienten Form darstellt; "-" wird hier nur als (Infix-)Operatorzeichen benutzt!

Differenzlisten

- Terme der Form $A_s - B_s$ werden nicht evaluiert, d.h. die Differenz wird **nicht berechnet**. Es findet "lediglich" Unifikation statt.
- Seien zwei Differenzlisten
 $[a_1, \dots, a_n \mid X] - X$ und $[b_1, \dots, b_m \mid Y] - Y$ gegeben.
Dann wird ihre Verkettung durch die Differenzliste
 $[a_1, \dots, a_n, b_1, \dots, b_m \mid Y] - Y$ dargestellt.
- append in Differenzlistendarstellung:

$\text{append}(X - Y, Y - Z, X - Z).$

Differenzlisten

- Beispiel:

 ?- append([a, b | X]-X, [c, d | Y]-Y, U).

 U = [a, b, c, d | Y]-Y

 X = [c, d | Y].

- Also: Differenzlistendarstellung nutzt die Suffix-Eigenschaft

- Jede Liste ist ein Suffix von sich selbst;

- Hat die Liste die Form [H | T], so ist jeder Suffix von T ein Suffix der ganzen Liste.

- D.h.: Die Beziehung zwischen Listen und ihren Suffixen ist die reflexive transitive Hülle der Relation zwischen der Liste und ihren Resten.

Quicksort mit Differenzlisten

```
quicksort(Xs, Ys) :- quicksort_dl(Xs, Ys - []).
```

```
quicksort_dl([], Xs - Ys).
```

```
quicksort_dl([X | Xs], Ys - Zs) :-  
    partition(X, Xs, Littles, Bigs),  
    quicksort_dl(Littles, Ys - [X | Y1s]),  
    quicksort_dl(Bigs, Y1s - Zs).
```

- In beiden Fällen ergibt sich die Implementation direkt aus der (noch informellen) Spezifikation...
- Weitere Sortierverfahren: \Rightarrow "Grundlagen der Algorithmik"
 - Mergesort
 - Treesort
 - ...

Meta-Programmierung

- Ein **Meta-Interpreter** ist ein Prolog-Programm, das andere Programme interpretiert ("**Programme als Daten**", d.h. Terme).
- Wozu?
 - Fehleranalyse (Debugging)
 - Effizienzsteigerung
 - Einführung neuer Sprachmittel
- Für die Meta-Programmierung sind spezielle **Meta-Prädikate** zur Bearbeitung von (Programmen als) Termen erforderlich:
 - Vergleiche
 - Typtests
 - Zugriff auf Termaufbau, Fakten und Regeln
 - (alternative) Interpretation
 - "all solutions" – Prädikate

Vergleiche, Typtests, Zugriff

- ... s. Kap. 4
- Vergleiche: `==`, `\==`, `@<` etc.
- Typtests:
 - `var/1`, `nonvar/1`, `atomic/1`, `atom/1`, `compound/1`
 - `number/1`, `integer/1`, `float/1`
- Zugriff auf Terme; Erzeugung:
 - `functor(Term, F, N)`
 - `arg(N, Term, Arg)`
 - `Term =.. List`
 - `copy_term(T1, T2)`

Zugriff auf die Datenbasis

- `clause(Head, Body)` ist erfolgreich, wenn
 - `Head` mit dem Kopf einer Klausel in der Datenbasis und
 - `Body` mit dem Rumpf der Klausel unifizierbar ist (bzw. `true`, falls Fakt).
- **Nicht-monotone Operationen** auf der Datenbasis:
 - `asserta(Clause)` fügt `Clause` am Anfang der Datenbasis ein.
 - `assertz(Clause)` fügt `Clause` am Ende der Datenbasis ein.
 - `retract(Clause)` entfernt `Clause` aus der Datenbasis.
- Hinweis (nicht ISO!): Mit `listing.` kann die (aktuelle) Datenbasis aufgelistet werden.

"All solutions"–Prädikate

findall (+Template, +Generator, -List)

Für jede mögliche Lösung des Generator-Ziels wird eine Instanz des Templates in die Liste aufgenommen.

```
likes(bill,wine).      likes(dick,beer).  
likes(tom,beer).      likes(tom,wine).  
likes(harry,beer).    likes(jan,wine).
```

```
| ?- findall(X, likes(X, Y), L).  
X = _626, Y = _662,  
L = [bill, dick, tom, tom, harry, jan]
```

```
| ?- findall([X, Y], likes(X, Y), L).  
X = _630, Y = _647,  
L = [[bill, wine], [dick, beer], [tom, beer], [tom,  
wine], [harry, beer], [jan, wine]]
```

```
| ?- findall(1, likes(X, Y), L).  
X = _642, Y = _659,  
L = [1,1,1,1,1,1]
```

```
| ?- findall(Y, likes(tina, Y), L).  
Y = _626,  
L = []
```

"All solutions"–Prädikate

bagof (+Template, +Generator, -List)

Bei bagof können die freien Variablen des Generators (die außerdem nicht im Template vorkommen) selektiv existentiell quantifiziert werden.

Verbleibende freie Variablen führen zu alternativen Lösungsmengen.

```
| ?- bagof(X, Y^likes(X, Y), S).  
X = _626, Y = _643,  
S = [bill, dick, tom, tom, harry, jan]
```

Var^Goal :

^: Existenzquantifikation,

verhindert Variablenbindung

```
| ?- bagof(X, likes(X, Y), S).  
X = _626, Y = beer, S = [dick, tom, harry];  
X = _626, Y = wine, S = [bill, tom, jan]
```

```
| ?- bagof(Y, likes(tina, Y), L).  
no
```

"All solutions"–Prädikate

setof (+Template, +Generator, -List)

Bei setof sind die Einträge in der Lösungsliste sortiert und enthalten keine Mehrfachlösungen.

| ?- setof(X, Y^likes(X, Y), S).

X = _626, Y = _643,

S = [bill, dick, harry, jan, tom]

Einfacher "Top-Down" Meta-Interpreter

- Bei gegebenem Ziel
 - sucht der Meta-Interpreter nach einer Regel, deren Kopf mit dem Ziel unifizierbar ist und
 - versucht, rekursiv den Rumpf der Regel zu beweisen.
- Fallunterscheidung:
Ziel ist Fakt oder Regel oder zusammengesetzt

```
prove(true) :- !.  
prove((Goal1,Goal2)) :- !,  
    prove(Goal1),  
    prove(Goal2).  
prove(Goal) :-  
    clause(Goal,Body),  
    prove(Body).
```

Anwendung: Erzeugen von Beweisbäumen

```
prooftree (+Goal, -Prooftree)
```

```
:- op(500,xfy,<==).
```

```
prooftree(true, true) :- !.
```

```
    prooftree((Goal1, Goal2), (Proof1, Proof2)) :- !,  
    prooftree(Goal1, Proof1), prooftree(Goal2, Proof2).
```

```
prooftree(Goal, Goal<==Proof) :-  
    clause(Goal, Body),  
    prooftree(Body, Proof).
```

Anwendung: Erzeugen von Beweisbäumen

Beispiel:

```
member (X, [X|Y]) .  
member(X, [Y|Z]) :- member(X, Z).
```

```
| ?- prooftree((member(X, [a,b,c]),  
               member(X, [b,c]),  
               member(X, [c, d, e])), Tree).
```

```
X = c,  
Tree = member(c, [a, b, c])<== member(c, [b, c])<==  
       member(c, [c])<==true,  
  
       member(c, [b, c])<==member(c, [c])<==true,  
       member(c, [c, d, e])<==true ;
```

Inhalt

- Arbeitsweise eines Prolog-Systems
- Rekursion über Terme
- Ablaufsteuerung
 - Cut
 - Sortieren und Differenzlisten
- Metaprogrammierung
- Systematische Programmentwicklung
- Alternative Ansätze der Logikprogrammierung

Systematische Programmentwicklung

- Von einer **logischen Spezifikation** zu einem sie realisierenden **Logikprogramm** scheint ein direkter Weg zu führen...
- Anhand einer Aufgabe sollen einige Probleme deutlich werden, die auf diesem Weg zu lösen sind (Quelle: Deville):

Schreibe eine Prolog-Prozedur `efface(X, L, Leff)`,

- die das erste Auftreten von `X` in der Liste `L` entfernt und als Resultat die Liste `Leff` liefert, und
 - fehlschlägt, wenn `X` nicht in der Liste `L` enthalten ist.
- Es werden drei Realisierungen dieser Spezifikation angegeben, die zunächst gleichwertig erscheinen, aber an Testfällen verschiedenes "Verhalten" zeigen (ohne "occurs check").

Beispiel: efface

Realisierung 1:

`efface1(X, [X|T], T).`

`efface1(X, [H|T], [H|Teff]) :- efface1(X, T, Teff).`

Realisierung 2:

`efface2(X, [X|T], T) :- !.`

`efface2(X, [H|T], [H|Teff]) :- efface2(X, T, Teff).`

Realisierung 3:

`efface3(X, [X|T], T).`

`efface3(X, [H|T], [H|Teff]) :- \+(X = H); efface3(X, T, Teff).`

Beispiel: efface – Tests

	Testdaten			Intend. Verh.	Tatsächl. Verh.		
	<i>X</i>	<i>L</i>	<i>E</i>		Proz. 1.1	Proz. 1.2	Proz. 1.3
1	2	[1, 2, 3, 2, 4]	[1, 3, 2, 4]	yes	korrekt	korrekt	korrekt
2	2	[1, 2, 3, 2, 4]	[1, 2, 3, 4]	no	yes	yes	korrekt
3	2	[2 3]	3	?	yes	yes	yes
4	2	[1, 2, 3, 2, 4]	<i>E</i>	$E = [1, 3, 2, 4]$	zusätzl. $E = [1, 2, 3, 4]$	korrekt	korrekt
5	0	[1, 2, 3, 2, 4]	<i>E</i>	no	korrekt	korrekt	korrekt
6	<i>X</i>	[1, 2, 3, 2, 4]	[1, 3, 2, 4]	$X = 2$	korrekt	korrekt	no
7	<i>X</i>	[1, 2, 3, 2, 4]	[1, 2, 3, 4]	no	$X = 2$	$X = 2$	korrekt
8	2	<i>L</i>	[1, 3, 2, 4]	$L = [2, 1, 3, 2, 4]$ $L = [1, 2, 3, 2, 4]$ $L = [1, 3, 2, 2, 4]$	zusätzl. $L = [1, 3, 2, 4, 2]$	$L = [2, 1, 3, 2, 4]$	korrekt
9	<i>X</i>	[1, 2, 3, 2, 4]	<i>E</i>	$X = 1, E = [2, 3, 2, 4]$ $X = 2, E = [1, 3, 2, 4]$ $X = 3, E = [1, 2, 2, 4]$ $X = 4, E = [1, 2, 3, 2]$	zusätzl. $X = 2, L = [1, 2, 3, 4]$	$X = 1, E = [2, 3, 2, 4]$	$X = 1, E = [2, 3, 2, 4]$
10	2	[2, 1 <i>L</i>]	<i>L</i>	no	yes	yes	yes
11	<i>X</i>	<i>L</i>	[1, 3]	$L = [X, 1, 3]$ $L = [1, X, 3], X \neq 1$ $L = [1, 3, X], X \neq 1, 3$	$L = [X, 1, 3]$ $L = [1, X, 3]$ $L = [1, 3, X]$	$L = [X, 1, 3]$	$L = [X, 1, 3]$
12	2	<i>L</i>	<i>E</i>	$L = [2 E]$ $L = [E_1, 2 T], E = [E_1 T]$ $E_1 \neq 2$	$L = [2 E]$ $L = [E_1, 2 T], E = [E_1 T]$ ⋮	$L = [2 E]$	$L = [2 E]$
13	<i>X</i>	<i>L</i>	<i>E</i>	$L = [X E]$ $L = [E_1, X T], E = [E_1 T]$ $E_1 \neq X$	$L = [X E]$ $L = [E_1, X T], E = [E_1 T]$ ⋮	$L = [X E]$	$L = [X E]$

Beispiel: efface – Diskussion

- `efface1` weicht oft vom intendierten Verhalten ab: Es eliminiert alle Vorkommen von X in L .
- Auch bei den Tests der Varianten 2 und 3 gibt es deutliche Unterschiede; daher kann Variante 2 auch nicht als optimierte Version der Variante 3 gelten.
- Die Schwierigkeit, ein Prolog-Programm anzugeben, das effizient ist und die Testfälle korrekt löst, liegt u.a. an der **ungenauen** und **unvollständigen** Spezifikation:
 - Es wird nicht angegeben, wie das Programm verwendet werden soll, d.h. was Eingabe- und Ausgabe-Argumente sein sollen;
 - Es wird nicht festgelegt, was beim Aufruf mit Argumenten falschen Typs geschehen soll (vgl. Testsatz 2).

Deklarative und prozedurale Semantik

- Die deklarative Semantik von Prolog auf der Grundlage der Quantorenlogik behandelt **logische Folgerungen** eines Programms.
- Die prozedurale Semantik auf der Grundlage eines Beweissystems regelt die **Berechnung von Antworten** (Beweissystem = SLD-Resolution + Berechnungsregel + Suchstrategie).
- Beide Semantiken sind durch **Korrektheits- und Vollständigkeitssätze** miteinander verknüpft.
- Logikprogrammierung ist zunächst systematische Konstruktion einer **logischen Beschreibung** des Problems – ausschließlich unter deklarativen Aspekten.
- Diese Beschreibung kann nach Umwandlung in ein Logikprogramm unter Ausnutzung der prozeduralen Aspekte zur Berechnung von Ergebnissen verwendet werden.

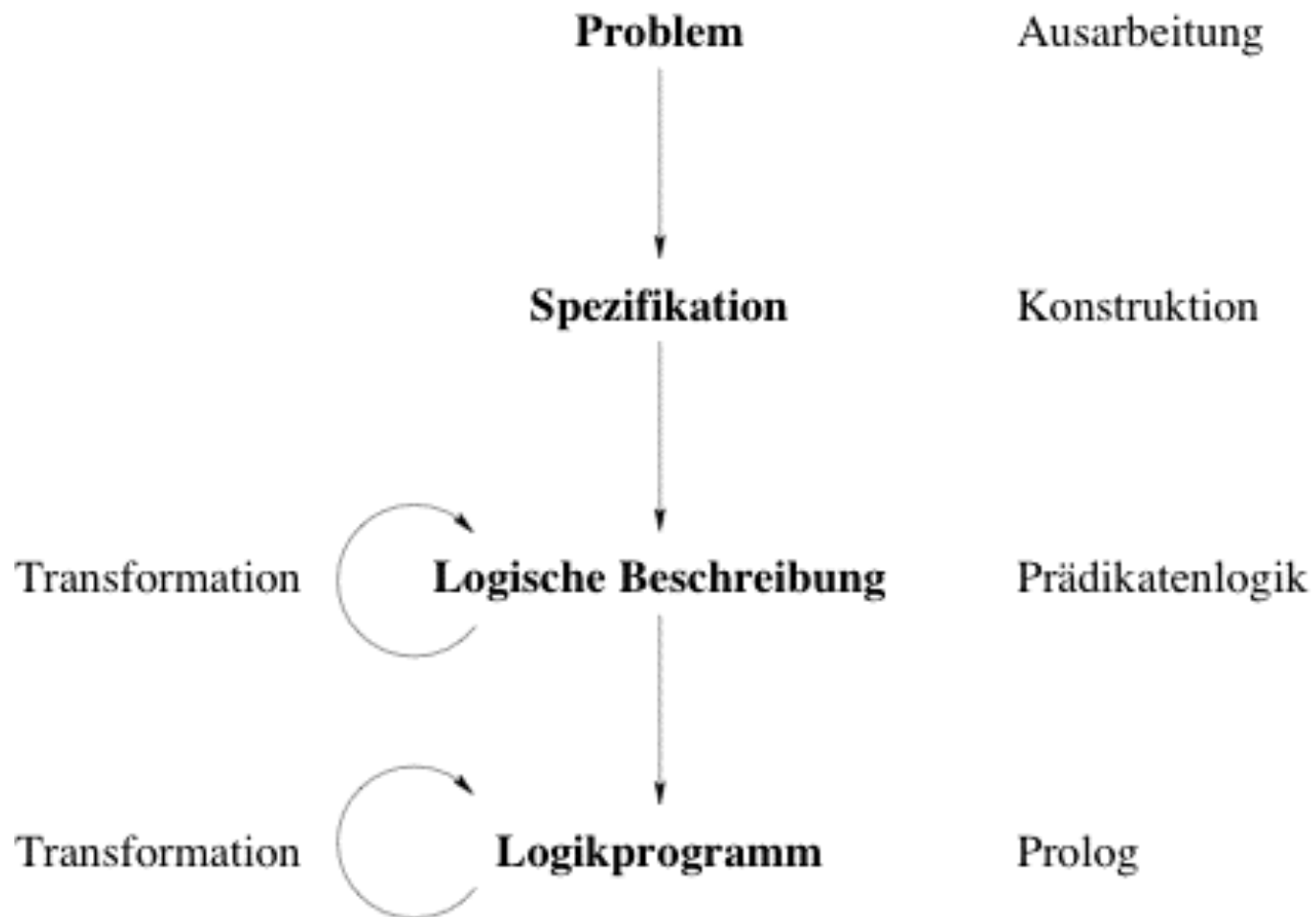
Prolog ist unvollständig

- Beispiel:
 - C1:** $p(a,b).$
 - C2:** $p(c,b).$
 - C3:** $p(X, Z) :- p(X, Y); p(Y, Z).$
 - C4:** $p(X, Y) :- p(Y, X).$

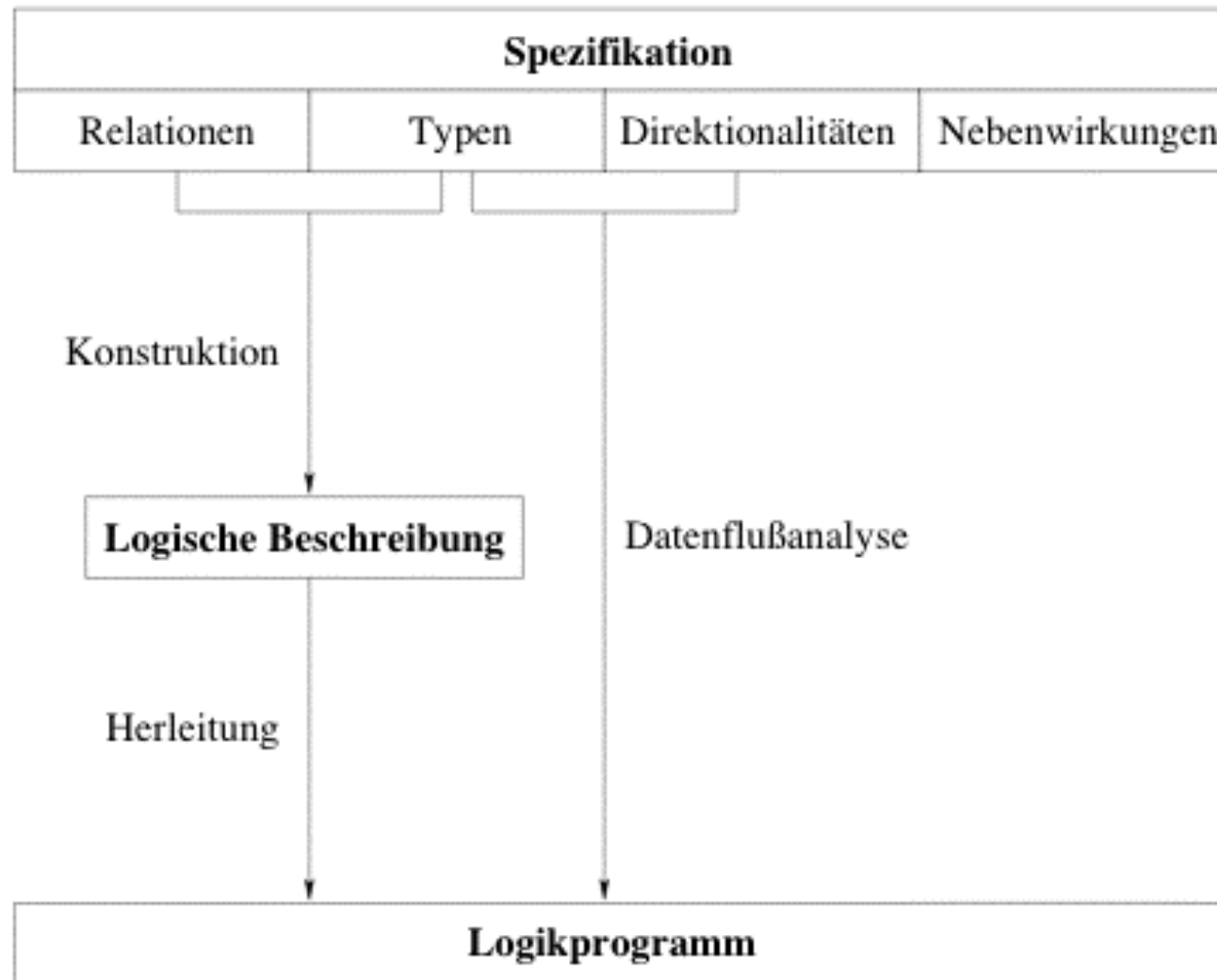
$p(a, c)$ ist logische Konsequenz des Programms.

- Kein Prolog-System (NAF-Regel + **Tiefensuche**) kann diese Konsequenz finden – unabhängig von der Anordnung der Regeln **C1** bis **C4** : **Unvollständigkeit**.
- **Unkorrektheit** liegt vor, wenn Antworten berechnet werden, die nicht logische Konsequenz eines Programms sind; sie wird typischerweise verursacht durch
 - den Einsatz **außerlogischer Prädikate** und das
 - **Unterlassen von occurs checks**.

Systematische Entwicklung von Logikprogrammen



Schema der Entwicklungsmethodik



Beispiel:

Systematische Entwicklung von efface

- Spezifikation:

Prozedur: `efface(X, L, Leff)`

Typen: `X`: ein beliebiger Term; `L, Leff`: Listen

berechnete Relation:

`X` ist ein Element von `L` und

`Leff` ist `L` ohne das erste Auftreten von `X`.

Verwendung (zulässige Direktionalitäten):

`in(any, ground, any) : out(ground, ground, ground),`

`in(ground, any, ground) : out(ground, ground, ground)`

- `in` spezifiziert die Instantiierungsbedingungen, die bei einer Aktivierung von `efface` vorliegen müssen,
- `out` die, die hinterher zu gelten haben.

Beispiel: Systematische Entwicklung von *efface*

- Logische Beschreibung durch strukturelle Induktion über \mathbb{L} :

$$\begin{aligned} \mathit{efface}(X, L, Leff) &\iff (\exists H, T, Teff) \\ &L = [] \wedge \mathit{false} \\ \vee L = [H \mid T] \wedge & (H = X \wedge Leff = T \\ &\vee H \neq X \wedge \mathit{efface}(X, T, Teff) \\ &\wedge Leff = [H \mid Teff]) \end{aligned}$$

Beispiel: Systematische Entwicklung von efface

- Aus der formalen Beschreibung abgeleitetes (reines) Prolog-Programm:

$$\begin{aligned} \text{efface}(X, L, \text{Leff}) &\leftarrow L = [H \mid T] \wedge H = X \wedge \text{Leff} = T. \\ \text{efface}(X, L, \text{Leff}) &\leftarrow L = [H \mid T] \wedge \text{Leff} = [H \mid \text{Teff}] \wedge \\ &\quad \text{efface}(X, T, \text{Teff}) \wedge \text{not}(H = X). \end{aligned}$$

- Um ein korrektes Programm zu erhalten, muss eine Permutation der Literale gefunden werden, die die Berechnung für die geforderten Direktionalitäten sicher macht. Dies ist im Beispiel erfüllt, es kann jedoch noch optimiert werden:

$$\begin{aligned} &\text{efface}_4(X, [X \mid T], T). \\ \text{efface}_4(X, [H \mid T], [H \mid \text{Teff}]) &\leftarrow \\ &\quad \text{efface}_4(X, T, \text{Teff}) \wedge \text{not}(H = X). \end{aligned}$$

Beispiel: Systematische Entwicklung von efface

- Ausschließlich für

$\text{in}(\text{ground}, \text{ground}, \text{any}) : \text{out}(\text{ground}, \text{ground}, \text{ground})$

$$\text{efface}_5(X, [X \mid T], \text{Leff}) \leftarrow ! \wedge \text{Leff} = T.$$

$$\text{efface}_5(X, [H \mid T], [H \mid \text{Teff}]) \leftarrow \text{efface}_5(X, T, \text{Teff}).$$

Prolog als Beweissystem?

- Sprache der Hornformeln:
 - Anfragen: Nur Adjunktion von Konjunktionen von Primformeln;
 - keine negierten Formeln können behauptet oder inferiert werden;
 - keine Adjunktion kann inferiert werden, solange nicht eines der Adjunktionsglieder inferiert werden kann.
- Damit gibt Hornlogik zwei Haupteigenschaften der Logik auf, die das Schließen mit unvollständigem Wissen erlauben:
 - zu sagen, dass eine von zwei Aussagen wahr ist, ohne zu wissen, welche;
 - zu unterscheiden zwischen dem Wissen, dass eine Aussage falsch ist und nicht zu wissen, dass sie wahr ist.

Prolog als Beweissystem?

- Beispiel von R.C. Moore:

Drei Bauklötze **A** – B – **C** seien in dieser Anordnung gegeben.

- A ist grün, C ist blau und die Farbe von B ist nicht bekannt.
- Gibt es in dieser Anordnung einen grünen Klotz direkt neben einem Klotz, der nicht grün ist?
- **Ja**, denn
 - wenn B grün ist, dann liegt ein grüner Klotz direkt neben dem nicht-grünen Klotz C;
 - Wenn B nicht grün ist, liegt direkt neben ihm der grüne Klotz A.

Prolog als Beweissystem?

- Um diese Aufgabe zu lösen, muss ein Beweissystem
 - festhalten, dass B entweder grün oder nicht-grün ist, und
 - diese Tatsache nutzen, um zu schließen, dass einige Klötze in einer Beziehung zu anderen stehen, ohne inferieren zu können, welche Klötze dies sind.
- Das ist nicht möglich mit einem System, das die CWA macht.

Alternative Ansätze der Logikprogrammierung

- **Concurrent Logic Programming**

- Das einfache Sortierbeispiel hat gezeigt, dass für eine effiziente Verarbeitung eine Verzahnung der beiden Prädikate `perm` und `ordered` notwendig wäre.

Man kann beide Teilziele als zwei **Prozesse** ansehen, die über die gemeinsame Variable `New` miteinander kommunizieren.

- Anstelle Teilziele i.a. sequentiell mit der SLD-Resolution zu lösen, könnte man auch eine parallele Lösung in Betracht ziehen:

"AND-Parallelismus".

Dies ist unproblematisch, wenn dabei keine gemeinsamen Variablen im Spiel sind bzw. wenn höchstens ein Teilziel eine gemeinsame Variable bindet. Andernfalls muss ein besonderer Kommunikations-/ Synchronisations-Mechanismus eingeführt werden.

Concurrent Logic Programming

- Die Prozess-Sicht auf Teilziele legt nahe, die Verarbeitung in Analogie zu Produzenten-Konsumenten-Modellen aus der Parallelverarbeitung zu organisieren:
 - Teilziele können agieren als **Prozesse**, die Eingaben aus einem oder mehreren "Datenströmen" **konsumieren** und Bindungen für logische Variablen **produzieren**.
 - Andere Teilziele wiederum können Variablenbindungen konsumieren und Ausgabeströme für Variable produzieren.
- Dieser Ansatz korrespondiert mit der strom-orientierten Verarbeitung in der funktionalen Programmierung: Die Datenströme rücken ins Zentrum der Modellierung von Systemen (Nachrichtentechnik!)
- Im Rahmen der Logikprogrammierung wurden solche Spracherweiterungen als "**Concurrent Logic Programming**" implementiert (s. z.B. Nilsson, Kap. 12).

Logikprogrammierung mit Gleichheit

- Prolog verfügt nur über eine äußerst rudimentäre Gleichheitstheorie (s. Kap. 4): Funktionen können durch Relationen dargestellt, Funktionsterme aber nicht ausgewertet werden.
- Für eine "volle" Implementierung der Gleichheit müssen ein spezielles Gleichheitsprädikat und Gleichungen als neuer Typ von Formeln eingeführt werden.
- Ein geeignetes Inferenzsystem muss den korrekten Schluss von gegebenen auf neue Formeln mithilfe von Ersetzungsregeln gestatten. Dies macht eine Erweiterung der Unifikation / Resolution erforderlich, die eine Theorie der Gleichheit realisiert, d.h. um entsprechende Schlussregeln erweitert wird.
- Problem: Effizienz!
Details s. z.B. Nilsson, Kap. 13; KK, Kap. 5.

Constraint Logic Programming

- **Beispiel** zur Motivation:

Gesucht ist ein Summen-Prädikat (beschränkt auf Integer-Argumente zwischen 1 und 20), das alle Kombinationen von Instantiierungen seiner Variablen zulässt.

$\text{sum}(X, Y, Z) :- \text{to}(1, 20, X), \text{to}(1, 20, Y), Z \text{ is } X+Y.$

Dabei ist `to` ein Generator-Prädikat für ein Intervall.

?- sum(3, 5, 8).

yes

?- sum(3, 5, X).

X = 8 ;

no

?- sum(X, Y, 5).

X = 1, Y = 4 ;

X = 2, Y = 3 ;

X = 3, Y = 2 ;

X = 4, Y = 1 ;

no

Constraint Logic Programming

- Das "Constraint" $Z \text{ is } X+Y$ kann nur geprüft werden, wenn X und Y instantiiert sind \Rightarrow "Generate-and-Test".
- Es wird nur benutzt, um im Nachhinein Pfade im Suchbaum als Sackgassen auszufiltern.
- Besser wäre eine Umformulierung des Programms je nach Instantiierung der Variablen.

Constraint Logic Programming

```
sum(X, Y, Z) :-  
    to(1, 20, X),  
    (nonvar(Z) -> (Y is Z-X, 1=<Y, Y=<20))  
    ;  
    (nonvar(Y) -> Z is X+Y  
    ;  
    to(1, 20, Y),  
    Z is X+Y)).
```

Hier: gleiche Kontrollkomponente (Prolog-Interpreter)

Reformulierung des Problems

CLP: gleiche Formulierung des Problems, aber

Veränderung der Kontrollkomponente

(Integration eines Constraint-Lösers in den Prolog-Interpreter)

Constraint Logic Programming

- Zwei Ansätze (Details s. z.B. Nilsson, Kap. 14):
 - Vollständiger Constraint-Löser für eine eingeschränkte Klasse von Constraints, z.B. Arithmetik von Real-Zahlen: u.a. Colmerauers Prolog II, III, IV.
Auch: Boolesche Constraints, Integer- und rationale Constraints, String-Constraints, Tree-Constraints.
Bezeichnung: CLP(domain).
 - Beliebige Constraints, deren Handhabung der Benutzer mittels Steueranweisungen selbst festlegen muss; z.B. P.v.Hentenryck.

Constraint Logic Programming

- Einsatzgebiete von CLP-Sprachen:
 - Schaltkreisentwurf
 - Optimierungsprobleme (Simplex)
 - Finanzwesen
 - Design
 - ...

Objekt-orientierte Programmierung

- Benötigte Sprachmittel:
 - **Definition von Objektklassen:** `object (Object, Methods)`
`object(rectangle(Length, Width),`
 `[(area(A) :- A is Length*Width),`
 `(describe :- write('Rectangle of size '),`
 `write(Length * Width))]).`
 - **Vererbung von Methoden:** `isa(Class, Superclass)`
`object(square(Side).`
 `[(describe :- write('Square with side'),`
 `write(Side))]).`

`isa(square(Side), rectangle(Side, Side)).`

Objekt-orientierte Programmierung

- **Instantiierung und Nachrichtenversenden:**

```
| ?- Rec1 = rectangle(4, 3),  
      send(Rec1, describe),  
      send(Rec1, area(Area)).
```

```
Rectangle of size 4 * 3  
Area = 12
```

Objekt-orientierte Programmierung

- Ein Meta-Interpreter für (derartige) objektorientierte Programme muss das Versenden von Nachrichten und die Vererbung handhaben:

```
send(Object,Message) :-
    get_methods(Object,Methods),    % Suche die Methoden des Obj.
    process(Message,Methods).       % Wende passende Methode an

get_methods(Object,Methods) :-      % Suche unter den
    object(Object,Methods).         %          privaten Methoden
get_methods(Object,Methods) :-      % Suche in den Oberklassen
    isa(Object,SuperObject),
    get_methods(SuperObject,Methods).

process(Message, [Message|_]).      % Es ist ein Fakt
process(Message, [(Message :- Body)|_]) :- % Es ist eine Regel
    call(Body).

process(Message, [_|Methods]) :-    % Weitersuchen
    process(Message,Methods).
```