Alternative Programmierkonzepte (T3INF4271)

Logische Programmierung

07
Prolog
Constraint Logic Programming

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Antonius van Hoof

AVH 2021

Prolog ist wie geschaffen für CLP

- **CLP ist sehr geeignet für kombinatorische und Optimierungs-Aufgaben**
  - Generell sind die Problem mit einem hohen asymptotischen Aufwand
  - Ohne CLP naiv in Prolog lösbar ("generate-and-test" Lösungen),
    aber dann bei größeren Problemen nicht mehr wirklich lösbar, weil zu ineffizient
- **CLP ist eine natürliche Erweiterung von Prolog** (wenn auch einbaubar in anderen Sprachen)
  - Logik ist selbst ein Art von Constraint System rundum Erfüllbarkeit (Satisfiability)
  - Wird so natürlich erweitert um Constraints in anderen Gebieten:
    - CLPB – Boolean Constraints (Variablen mit Wert aus {t,f})
    - CLPFD – Constraints auf ganzen Zahlen (Zahlenraum Z), Finite Domäne
    - CLPQ und CLPR – Constraints auf den Zahlenräumen Q und R
  - Zahlen-Constraints sind natürlich geeignet für arithmetische Probleme
    → deklarative Arithmetik
    aber auch sonst einsetzbar für nicht direkt numerische Probleme
- **Obige Constraint-Systeme sind fertige Solver-Systeme**
  - Man kann aber eigene Solver-Systeme bauen mit CHR
  - CHR: Constraint Handling Rules (behandeln wir hier nicht weiter)

Anwendungen von CLP

- **Generell: Bestimmen und Optimieren von Ressource Allokationen**
  - Insb. Konstruktion von Zeit- und Raumbelegungsplänen
  - Fahrpläne
  - Tourenpläne
  - Platinenauslegung
  - Maschinenbelegungen
  - Prozessausgestaltungen
  - Usw.
- **In Prolog nicht nur Generieren solcher Pläne, sondern auch Test und Komplettierung**

Beispiel (der Klassiker): Map Colouring

- https://www.youtube.com/watch?v=6XD7vBbywMc

Beispiel: CLP und State Space Probleme
Hier: das Wolf-Ziege-Kohl-Problem

```
solve(Solution) :-
   wzk([[0,0,0,0]],RevSolution),
   flatten(RevSolution,FP),
   labeling([],FP),
   reverse(RevSolution,Solution).

wzk([[1,1,1,1]|PrevStates], [[1,1,1,1]|PrevStates]).
wzk([[B,W,Z,K]|PrevStates],AllStates) :-
   next([B,W,Z,K],NewState),
   maplist(dif(NewState),PrevStates),
   wzk([NewState,[B,W,Z,K]|PrevStates],AllStates).
```

```
next([B,W,Z,K],[NB,NW,NZ,NK]) :-
   [B,W,Z,K,NB,NW,NZ,NK] ins 0..1,
   NB #\= B,          % Der Bauer muss fahren
   % zusätzlich darf nur ein weiteres Objekt fahren
   % d.h. zwei bleiben, wo sie sind
   % wir benutzen hier die Reifikation der Constraints
   (  NW #= W #/\ NZ #= Z) #\/
   (  NZ #= Z #/\ NK #= K) #\/
   (  NW #= W #/\ NK #= K),
   % wer darf nicht ohne Bauer zusammenbleiben
   NZ #= NK #==> NB #= NZ,
   NW #= NZ #==> NB #= NW.
```


Beispiel: CLP und State Space Probleme
Hier: das Wolf-Ziege-Kohl-Problem
Jetzt: Als DCG

```
dcg_solve(Solution) :-
   phrase(dcg_wzk([0,0,0,0],[[0,0,0,0]]),Solution),
   flatten(Solution,FS),
   labeling([],FS).


dcg_wzk([1,1,1,1],_) --> [[1,1,1,1]].
dcg_wzk(State,Visited) --> [State],
   { next(State,NextState),
     maplist(dif(NextState),Visited)
   },
   dcg_wzk(NextState,[NextState|Visited]).
```

```
next([B,W,Z,K],[NB,NW,NZ,NK]) :-
   [B,W,Z,K,NB,NW,NZ,NK] ins 0..1,
   NB #\= B,          % Der Bauer muss fahren
   % zusätzlich darf nur ein weiteres Objekt fahren
   % d.h. zwei bleiben, wo sie sind
   % wir benutzen hier die Reifikation der Constraints
   (  NW #= W #/\ NZ #= Z) #\/
   (  NZ #= Z #/\ NK #= K) #\/
   (  NW #= W #/\ NK #= K),
   % wer darf nicht ohne Bauer zusammenbleiben
   NZ #= NK #==> NB #= NZ,
   NW #= NZ #==> NB #= NW.
```

Weitere Beispiel-Demos

- **Knights and Knaves (Boolean Solver)**
  - https://www.youtube.com/watch?v=oEAa2pQKqQU
- **Sudoku**
  - https://www.youtube.com/watch?v=5KUdEZTu06o
- **N-Queens**
  - https://www.youtube.com/watch?v=I_tbL9RjFdo
- **School Timetabling**
  - https://www.youtube.com/watch?v=uKvS62avplE

- **Einige dieser und weitere, zum Selbst Spielen auch zu finden unter:**
  - https://swish.swi-prolog.org/example/examples.swinb

Führen jeweils zu extrem kurze, knappe, rasend schnelle und meist flexibel einsetzbare Implementierungen
→ Richard O'Keefe: **"Elegance is not optional"**
[The Craft of Prolog, S.4ff]

Alternative Programmierkonzepte (T3INF4271)

Logische Programmierung

08
Prolog
Meta-Interpretation
Expertensysteme

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Antonius van Hoof

Was ist ein Meta-Interpreter?

- **Ein Meta-Interpreter für eine Sprache ist ein Interpreter für die gleiche Sprache in der diese Interpreter geschrieben ist**
  - Hier: ein Prolog Meta-Interpreter ist ein Interpreter für Prolog, der selbst in Prolog geschrieben ist
- **Meta-Programme benutzen andere Programme als Daten, die analysiert, transformiert und ausgeführt/simuliert werden**
- **Aufgrund der Homoikonizität von Prolog ist es besonders einfach Meta-Programme und insbesondere Meta-Interpreter in Prolog zu schreiben**

Einfachster Meta-Interpreter für Pure Prolog

```
solve(true):- !.
solve((FirstQuery,RestOfQueries)):-
   !,
   solve(FirstQuery),
   solve(RestOfQueries).
solve(Query):-
   clause(Query,Tail),
   solve(Tail).
```

Damit ist noch nicht viel gewonnen, interessant wird es, wenn man während des Ausführungsvorgangs noch weiteres anstellt

Beispiel:
Interpreter, der einen Beweisbaum liefert

```
:- op(1100, xfx, 'BECAUSE').
:- op(1000, xfy, 'AND').

% %   solve(+Goal,-Proof) is semidet.
%
%       prove Goal in Pure Prolog and show its Proof
%
solve( true,'SIMPLY TRUE') :- !.
solve((A,B),ProofA 'AND' ProofB) :-
        solve(A,ProofA),
        solve(B,ProofB).
solve(Goal,(Goal 'BECAUSE' Proof)) :-
        clause(Goal,Body),
        solve(Body,Proof).
```


Beispiel:
A Meta-Interpreter for full Prolog

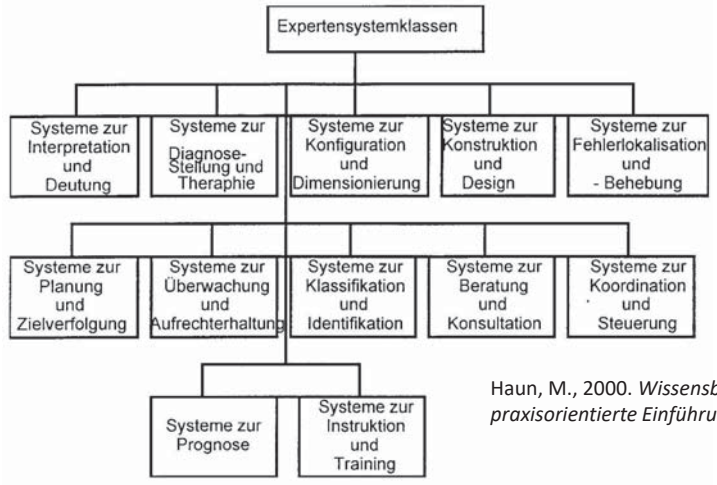
```
% %       solvefp(+Goal,-Proof) is semidet.
%
%       prove Goal in full Prolog and show its Proof
%
solvefp( true,'SIMPLY TRUE') :- !.
solvefp((A,B),ProofA 'AND' ProofB) :-
        !, solvefp(A,ProofA), solvefp(B,ProofB).
solvefp(!,'CUT') :-
        !, retract('$choice'(ChoicePoint)),
        prolog_cut_to(ChoicePoint).
solvefp(\+ Goal, 'NOT'(Goal)) :-
        !, \+ solvefp(Goal,_).
solvefp(setof(X,Goal,Xs), one_of(X,Xs)) :-
        !, setof(X,P^solvefp(Goal,P),Xs).      % dito for bagof/3 and findall/3
solvefp(A,'SYSTEMCALL'(A)) :-
        systempred(A), !, call(A).
solvefp(Goal,Proof) :-
        reduce(Goal,Proof).

reduce(Goal,(Goal 'BECAUSE' Proof)) :-
        prolog_current_choice(ChoicePoint), asserta('$choice'(ChoicePoint)),
        clause(Goal,Body), solvefp(Body,Proof).
```

```
systempred(call(_)).
systempred(read(_)).
systempred(write(_)).
systempred(nl).
systempred(clause(_,_)).
systempred(_ < _).
% usw.
```

Expertensysteme

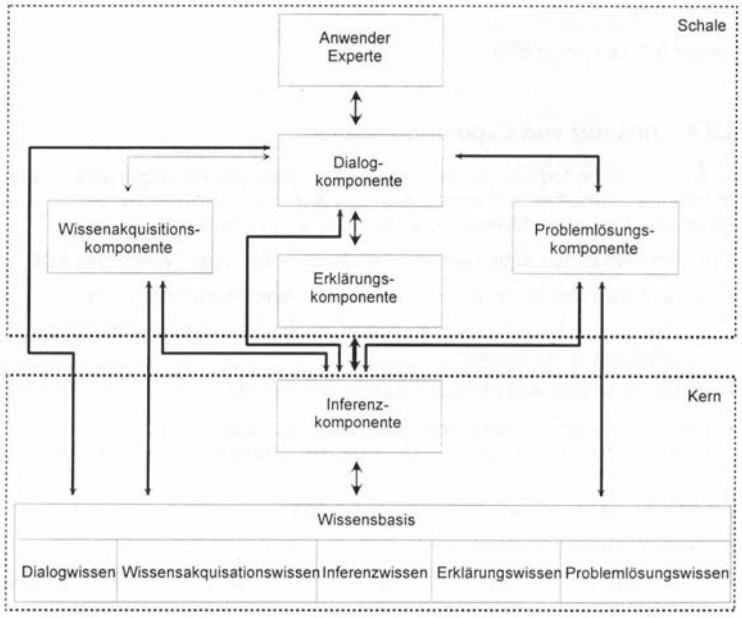
- **Eine weitere Anwendung von Prolog findet sich leicht auf dem Gebiet der Expertensysteme.**
- **Expertensysteme kann man folgendermaßen klassifizieren:**



Haun, M., 2000. *Wissensbasierte Systeme: Eine praxisorientierte Einführung*, Expert-Verlag.

Klassifikation der Expertensysteme (Haun 2000), Seite 126

Architektur von Expertensystemen



Architektur von Expertensystemen nach (Haun 2000)

Expertensysteme und Prolog

- **Diese lassen sich einfach in Prolog umsetzen. Das liegt daran, dass man in Prolog sehr leicht Meta-Interpreter für Prologcode schreiben kann, wie wir bereits gesehen haben**

- **Es geht bei Meta-Interpretern darum, dass man vor oder nach einer Goal-Ausführung etwas mit dem Goal anstellt. Dies passiert für Expertensysteme in Prolog folgendermaßen:**
  - Das vorhandene Expertenwissen wird in Form von Prolog-Fakten und vor allem -Regeln direkt in Prolog codiert. Dies ist dann der Expertensystemkern
  - Die Module der Schale sind in Form einer Meta-Interpreter für den Code des Kerns implementiert. Solch eine Schale kann man für andere Kerne wiederverwenden.

Beispiel eines Expertensystem in Prolog

- **Das Prologbeispielprogramm**

  `expert_system_shell.pl`

- **bildet zusammen mit den Beispiel-Kern**

  `expert_system.pl`

- **ein Expertensystem, das interaktiv Fragen an den Benutzer stellen kann und dabei auch erklären kann warum es diese Fragen stellt.**