Logisch programmeren 2012, Week 7

Michael Moortgat

1. Inhoud

- Meta-programmeren, interpreters
- `term_expansion/2`
- Zoeken met maximale diepte

2. Interpreters: meta-programmeren

De ingebouwde bewijsprocedure van Prolog heeft de volgende eigenschappen:

- keuze van literals: links-rechts
- regel selectie: van boven naar beneden
- zoeken: depth-first, backtracking

Soms wil je zelf de controle over het gedrag van de bewijsprocedure.

`Meta-programmeren`: je schrijft een Prolog programma dat een object-niveau programma als `data` gebruikt.

3. Voorbeeld: Prolog in Prolog

Het ingebouwde `clause/2` is een metapredicaat om een Prolog programma te inspecteren. Aanroep `clause(Head,Body)`, waarbij `Body` is `true` in het geval van een feit.

```
clause(append([], L, L),true)).
clause(append([H|T], L, [H|T1]),append(T, L, T1)).
```

Interpreter `prove/1` gebruikt programmaclauses als feiten gecodeerd met `clause/2`.

```
prove(true).
prove(Goal) :-
    clause(Goal,Body),
    prove(Body).
prove((Body1, Body2)) :-
    prove(Body1),
    prove(Body2).
```

4. Absorptie

`prove/2` houdt een bewijsboom bij voor zijn afleidingen.

```
prove(true, true).
prove(Goal, (Goal :- BodyProof)) :-
    clause(Goal,Body),
    prove(Body, BodyProof).
prove((Body1, Body2), (Body1Proof, Body2Proof)) :-
    prove(Body1, Body1Proof),
    prove(Body2, Body2Proof).
```

Sommige onderdelen van de bewijsprocedure zijn voor rekening van Prolog, andere kunnen expliciet gemanipuleerd worden.

- geabsorbeerd in Prolog: unificatie goal/head; regelvolgorde
- te manipuleren: literal selectie

5. Een interpreter voor DCG

We kunnen nu zelf een interpreter voor DCG regels schrijven.

- Regels zijn gecodeerd met `--->/2`
- Positie argumenten worden door de `parse/3` interpreter toegevoegd

```
parse(NT,L0,L) :-
   (NT ---> Body),
   parse(Body,L0,L).
parse((Body1,Body2),L0,L) :-
   parse(Body1,L0,L1),
   parse(Body2,L1,L).
parse([],L,L).
parse([Word|Rest],[Word|L0],L) :-
   parse(Rest,L0,L).
parse({Goals},L,L) :- call(Goals).
```

6. DCG in DCG

Nog beter: de `parse/2` interpreter als DCG!

- `-->/2` wordt door Prolog verwerkt
- `--->/2` is voor rekening van `parse/2`

```
parse(NT) -->
   {NT ---> Body},
   parse(Body).
parse((Body1,Body2)) -->
   parse(Body1),
   parse(Body2).
parse([]) --> [].
parse([Word|Rest]) -->
   [Word],
   parse(Rest).
parse({Goals}) --> {call(Goals)}.
```

7. term_expansion/2

Metaprogramma's zijn minder efficiënt dan de Prolog programma's gedefinieerd met `':-'`.

Het ingebouwde `term_expansion/2` transformeert een gegeven predicaat tot Prolog code in termen van `':-'`.

Hieronder als voorbeeld het omschrijven zoals dat voor de dcg notatie gebeurt. We gebruiken onze eigen lange pijl `'--->'/2`.

```
:-op(1200,xfx,'--->').

term_expansion((LHS--->RHS),(Head:-Body)):-
     dcg(LHS,Head,P,P1),
     dcg(RHS,Body,P,P1).
```

8. term_expansion/2, vervolg

```
dcg((A,B),(A1,B1),P,P1):-   % concatenatie
    !,
    dcg(A,A1,P,P0),
    dcg(B,B1,P0,P1).

dcg((A;B),(A1;B1),P,P1):-   % keuze
    !,
    dcg(A,A1,P,P1),
    dcg(B,B1,P,P1).

dcg(L,(P=P0),P,P1):-   % lijst
    is_list(L),
    !,
    list2dl(L,P0,P1).
```

9. term_expansion/2, vervolg

```
dcg(NT,H,P,P1):-   % predicaat uitbreiden
    NT =.. [F|Args],
    append(Args,[P,P1],Tail),
    H =.. [F|Tail].

% lijst omzetten in verschillijst

list2dl([])-->[].
list2dl([H|T])-->[H],list2dl(T).
```

10. Zoeken met een drempelwaarde

Prolog's ingebouwde depth-first strategie kan falen voor links-recursieve programma's. Voorbeeld: DCG met links-recursieve regels.

In de opdracht bouw je een meta-interpreter voor `iteratief verdiepend` zoeken (=`consecutively bounded` depth-first search).

- `bounded` search breekt het zoeken naar een afleiding af zodra een gegeven drempelwaarde is bereik
  - terminatie: gegarandeerd
  - incompleet: een oplossing kan voorbij de drempel liggen
- `consecutively bounded search` start met een minimale drempelwaarde k (voor axioma's); als er geen oplossingen meer zijn voor drempel k ga je over op zoeken met drempel $k + 1$
  - complete strategie: als er een oplossing is, wordt die gevonden
  - terminatie: als een probleem *geen* oplossing heeft, moet je nog steeds afbreken bij een voldoende hoge drempelwaarde

11. Zoeken met een drempelwaarde: voorbeeld

Definitie van `prove(Drempel,Goal,ProofTree)`: het bewijs mag niet dieper worden dan `Drempel`.

```
prove(_,true, true).
prove(N,Goal, (Goal :- BodyProof)) :-
    N>0,
    Goal\=true,
    clause(Goal,Body),
    N1 is N-1,
    prove(N1,Body, BodyProof).
prove(N,(Body1, Body2), (Body1Proof, Body2Proof)) :-
    prove(N,Body1, Body1Proof),
    prove(N,Body2, Body2Proof).
```

12. Iteratief verdiepend zoeken

```
id_prove(Goal,ProofTree) :-     % wrapper
    id_prove(1,Goal,ProofTree).

id_prove(N,Goal,ProofTree) :-
    prove(N,Goal,ProofTree).
id_prove(N,Goal,ProofTree) :-
    N1 is N+1,
    id_prove(N1,Goal,ProofTree).
```