Einführung in das Programmieren – Prolog
Sommersemester 2006

Teil 8: Fortgeschrittene Techniken

Version 1.0

Gliederung der LV

Teil 1: Ein motivierendes Beispiel

Teil 2: Einführung und Grundkonzepte
- Syntax, Regeln, Unifikation, Abarbeitung

Teil 3: Arithmetik

Teil 4: Rekursion und Listen

Teil 5: Programmfluß
- Negation, Cut

Teil 6: Verschiedenes
- Ein-/Ausgabe, Programmierstil

Teil 7: Wissensbasis
- Löschen und Hinzufügen von Klauseln

Teil 8: Fortgeschrittene Techniken
- Metainterpreter, iterative Deepening, PTTP, Differenzlisten, doppelt verkettete Listen

Differenzlisten

- Aus Laufzeiteffizienz grosser Unterschied ob Element `X` in einer Liste am Kopf oder am Ende einer Liste `L` eingefügt werden soll.
- Für Einfügen vorne reicht eine Unifizierungsoperation

```
L1 = [X|L]
```

- Andernfalls muß man sich bis ans Ende durchhangeln

```
append(L,[X],L1)
```

- Grund für unterschiedlichen Aufwand: Implementierungstechnische Details von Listen in Prolog

Offene Listen

Wie wäre es mit folgender Idee:

- repräsentiere Liste anders, als sogenannten ***offene Liste***
  - das heißt, die Liste wird nicht mit [] abgeschlossen, sondern mit einer Variablen
  - Beispiel: statt [1,2,3] jetzt [1,2,3|X], d.h. .(1,.(2,.(3,X)))
- Anhängen an Liste ist jetzt eine einzige Unifikation, nämlich das Ersetzen der Variable
  - Beispiel: L=[1,2,3|X], X = [4,5] ergibt [1,2,3,4,5]

Von offenen Listen zu Differenzlisten

Einziges Problem: Woher kennen wir die Variable X?

Lösung: wir speichern diese Variable irgendwo am Anfang der Liste

- Beispiel: [1,2,3] wird gespeichert als f([1,2,3|X], X)

Anhängen jetzt ganz einfach:

```
app(f(Liste,X), Liste2, Ergebnis):-
        X = Liste2,
        Ergebnis = Liste.
```

Differenzlisten

Obwohl der Funktor f eigentlich beliebig ist, wird er aus bestimmten Gründen gewählt: -

- Infixnotation, was das Lesen erleichtert
- Assoziation des Funktors '-' mit Differenz, daher der Begriff ***Differenzlisten***
  - Liste [1,2,3] repräsentiert als Differenzliste [1,2,3—X]-X

idiomatisch:

- Wenn man das X in der DL [1,2,3—X]-X durch irgendeine Liste , z.B. [4,5] ersetzt, erhält man [1,2,3,4,5]-[4,5], bei Ersetzung durch [a,b,c] ergibt sich [1,2,3,a,b,c]-[a,b,c]
- was man wiederum als Differenz lesen kann und eine schöne Erklärung für den Namen liefert

Anhängen an Differenzliste

Aufgabe: Hänge ein Element an die DL an:

```
app_dl(DL, Element, DLneu):-
        DL = L-X,
        X = [Element | Y],
        DLneu = L-Y.
```

Läßt sich übrigens auch kürzer schreiben:

```
app_dl(L-[Element | Y], Element, L-Y).
```

Verketten zweier Differenzlisten

```
append_dl(DL1, DL2, DLneu):-
        DL1 = L1-X1,    % Struktur der ersten DL
        DL2 = L2-X2,    % Struktur der zweiten DL
        X1 = DL2,       % Belegen des Endes von DL1, d.h. Anhängen
        DLneu = L1-X2.  % neue Liste zusammenbauen
```

Läßt sich kürzer schreiben:

```
append_dl(L-X, X-Y, L-Y).
```

Trsf. zwischen normalen Listen und Differenzlisten

Differenzliste in normale Liste:

```
dl2l(L-[ ], L).
```

Normale Liste in Differenzliste

```
l2dl(L, L1-X):-
            l2ol(L,X,L1).

l2ol([ ], X, X).

l2ol([E | R], X, [E | R1]):-
            l2ol(R, X, R1).
```

Metainterpreter

```
% L ist eingebaut
prove( L ) :-
        builtin(L),
        L.
% Beweis eines Goals durch ein Fakt
prove( L ) :-
        not(builtin(L)),
        clause( L, true).
% Beweis einer Konjunktion von Goals
prove( (C1,C2) ) :-
        prove(C1),
        prove(C2).
% Beweis eines Goals mithilfe einer Regel
prove( Head ) :-
        not(builtin(Head)),
        not( Head = (_, _)),
        clause( Head, Body ),
        Body \= true,
        prove(Body).
```

Metainterpreter

```
% L ist eingebaut
proof_tree( L, L) :-
            builtin(L),
            L.

% Beweis eines Fakts ist das Fakt selbst
proof_tree( L, L) :-
            not(builtin(L)),
            clause(L, true).

% Beweis einer Konjunktion von Goals ist die Konjunktion der Beweise für die Goals
proof_tree( (C1,C2), (ProofC1, ProofC2) ) :-
            proof_tree(C1, ProofC1),
            proof_tree(C2, ProofC2).

% Beweis eines Goals mit einer Regel ist der Beweis des Bodies
proof_tree( Head, (Head :- BodyProof)) :-
            not(builtin(Head)),
            not( Head = (_,_)),
            clause(Head, Body ),
            Body t̄rue,
            proof_tree(Body, BodyProof).
```

Iterated Deepening

Prolog implementiert Tiefensuche

- Bekommt man auch Breitensuche hin?

Nur mit großem Aufwand

Aber: Approximation der Breitensuche möglich

Zunächst: ***beschränkte Tiefensuche***

- Setze Tiefenschranke und lasse nur bis zu dieser Schranke rechnen

Idee:

- Erweitere jedes Prädikat um ein zusätzliches Argument, den Tiefenzähler. Jeder Aufruf erhöht diesen Zähler.
- Zu Beginn jedes Prädikats steht der Test, ob der Tiefenzähler noch kleiner als die Schranke ist.

```
p(...):-
        q1(...),
        q2(...),
        ...
```

wird zu

```
p(...) :- p(...,0).
p(..., I):-
        I < 42,
        succ(I1, I),
        q1(..., I1),
        q2(..., I1),
        ...
```

Iterated Deepening

Iterierte Tiefensuche

- Setze die Tiefenschranke auf einen Startwert
- Wenn innerhalb dieser Schranke keine Lösung gefunden wird, erhöhe Tiefenschranke und rechne erneut

Idee:

- Erweitere jedes Prädikat um zwei zusätzliches Argumente, den Tiefenzähler und die Schranke. Jeder Aufruf erhöht den Zähler.
- Wenn mit aktueller Schranke kein Erfolg, erhöhe sie.
- Zu Beginn jedes Prädikats steht der Test, ob der Tiefenzähler noch kleiner als die Schranke ist.

```
p(...):-
        q1(...),
        q2(...),
        ...
```

wird zu

```
p(...) :- p(...,0,0).
p(..., _, S):- succ(S,S1), p(..., 0, S1).
p(..., I, S):-
        I < S,
        succ(I1, I),
        q1(..., I1),
        q2(..., I1),
        ...
```

Rekursive Strukturen

Angenommen, wir wollen folgende Struktur repräsentieren:

$$\begin{array}{ccc} a & \longrightarrow & b \\ \uparrow & & \downarrow \\ d & \longleftarrow & c \end{array}$$

Standardansatz:

```
nf(a,b).
nf(b,c).
nf(c,d).
nf(d,a).
```

Wie wäre es mit folgender rückgekoppelter Struktur?

Term = nf(a, nf(b, nf(c, nf(d, Term))))