6.3 Erweiterungen von Prolog

Prolog ist prinzipiell eine berechnungsuniverselle Programmiersprache. Trotzdem sind verschiedene Erweiterungen möglich, die praktisch auch nützlich sind. Die wichtigsten Erweiterungen wollen wir im Folgenden kurz diskutieren.

6.3.1 Negation

Im bisherigen Sprachumfang konnten keine negativen Bedingungen in Regeln formuliert werden, obwohl dies manchmal nützlich wäre, wie das folgende Beispiel zeigt (logische Programme mit Negation werden manchmal auch als **normale Programme** bezeichnet).

Beispiel 6.8 (Negation)**.** Zwei Mengen `X` und `Y` sind verschieden:

```
verschieden(X,Y) :- element(Z,X), ¬element(Z,Y).
```

Damit stellt sich das Problem, wie man mit „¬“ beweistechnisch umgehen kann. Eine echte prädikatenlogische Negation ist aufwändig, da dann eine *lineare* Resolution nicht mehr möglich wäre. Daher wird die Negation in Prolog als **Negation als Fehlschlag** (**negation as failure**) interpretiert. Dies wird in Prolog als `\+` statt ¬ notiert, also

```
verschieden(X,Y) :- element(Z,X), \+ element(Z,Y).
```

Interpretiert wird „`\+` l“ als: Falls alle Beweise für das Literal l fehlgeschlagen, dann ist „`\+` l“ beweisbar.

Inferenzregel für “negation as failure”

$$\frac{P', \sigma \vdash l : [\,]}{cs, \sigma \vdash \backslash\!+ l : [\sigma]}$$

Hierbei ist P' eine Variante des Programms P mit jeweils neuen Variablen.

Wird die SLD-Resolution um diese Regel erweitert, spricht man auch von **SLDNF-Resolution**. Bedingungen für die Korrektheit der SLDNF-Resolution findet man z. B. in (Lloyd, 1987). Zusammengefasst ist die SLDNF-Resolution im Sinne der Logik korrekt unter folgenden Bedingungen:

1. Interpretiere Implikationen in Regeln als Äquivalenzen. Zum Beispiel wird das Programm

   ```
   p(a).
   p(b).
   ```

   als Formel $\forall x : p(x) \leftrightarrow (x = a \vee x = b)$ interpretiert, Man spricht dann auch von der „**Vervollständigung**“ (completion) des Programms.

2. Beweise „`\+` l“ nur, falls l variablenfrei ist (d.h. man muss eine flexible Selektionsregel anwenden).

Die Notwendigkeit der letzten Einschränkung soll durch ein kleines Beispiel gezeigt werden:

```
p(a).
q(b).
```

Die Anfrage sei „`?- \+ p(X), q(X).`“

- Falls „`\+ p(X)`“ direkt selektiert würde, dann wäre „`p(X)`“ beweisbar und damit „`\+ p(X)`“ nicht beweisbar, wodurch die Anfrage insgesamt nicht beweisbar wäre.
- Falls der Beweis von „`\+ p(X)`“ zunächst zurückgestellt wird: Beweise „`q(X)`“, was zu der Lösung `{X`$\mapsto$`b}` führt, und beweise dann die zunächst zurückgestellte Aussage „`\+ p(b)`“, was nun erfolgreich möglich ist. Damit erhalten wir als Gesamtlösung `{X`$\mapsto$`b}`.

6.3.2 Erweiterte Programme

In erweiterten Programmen dürfen Regeln der Form l `:-` b vorkommen, wobei b eine beliebige prädikatenlogische Formel (d.h. mit Disjunktionen, Negation, Quantoren) ist. Dies ist manchmal praktisch, erhöht aber nicht die Ausdrucksmächtigkeit, da erweiterte Programme in normale Programme transformierbar sind (Lloyd, 1987). Trotzdem werden erweiterte Programme z.B. für Datenbankanwendungen betrachtet.

6.3.3 Flexible Berechnungsregeln

Die Standardberechnungsregel von Prolog ist, dass alle Literale von links nach rechts bewiesen werden. Hierzu hat man auch verschiedene Modifikationen betrachtet:

Prolog mit Koroutinen: Verzögere den Beweis von Literalen, bis bestimmte Bedingungen erfüllt sind (z. B. das Literal ist variablenfrei). Dies ist nützlich zur korrekten Implementierung der SLDNF-Resolution, kann aber auch verwendet werden, um bestimmte Endlosableitungen in Programmen zu vermeiden.

Andorra-Prolog: Bei diesem Berechnungsprinzip werden zuerst „deterministische“ Literale bewiesen (bei denen höchstens eine Klausel passt). Dies führt häufig zu einer Verkleinerung des Suchraums. Kombiniert wird dieses Prinzip dann noch mit einer parallelen Auswertung.

Parallele logische Sprachen: Bei der Erweiterung um parallele Auswertung kann man folgende Arten unterscheiden:

UND-Parallelismus: Beweise mehrere Literale in einer Anfrage gleichzeitig (unabhängig oder abhängig bzgl. gleicher Variablen in verschiedenen Literalen).

ODER-Parallelismus: Probiere gleichzeitig verschiedene Alternativen (Regeln) aus (dies entspricht einer Breitensuche im SLD-Baum aller Ableitungen).

Committed Choice: Erweitere Klauselrümpfe um eine Bedingung ("guard"):

```
<head> :- <guard> | <body>
```

Falls mehrere Klauseln zu einem Literal passen: Wähle nicht-deterministisch („paralleles Raten") eine Klausel, bei der die Bedingung `<guard>` beweisbar ist, und ignoriere die anderen Alternativen. Dieses Prinzip ist aus logischer Sicht unvollständig, aber es ist gut geeignet zur deklarativen Programmierung nebenläufiger Systeme.

6.3.4 Constraints („Einschränkungen")

Der grundlegende Lösungsmechanismus in logischen Sprachen ist die Unifikation, was dem Lösen von Gleichungen zwischen Termen entspricht. Der Nachteil hiervon ist, dass Gleichungen rein syntaktisch gelöst werden, ohne dass ein „Ausrechnen" stattfindet:

```
?- X=3+2.
X=3+2
```

Wünschenswert wäre hier aber die Lösung `X=5`.

```
?- 5=X+2.
no
```

Wünschenswert wäre hier aber die Lösung `X=3`.

Um Letzteres zu erreichen, könnte man außer der Termgleichheit (Strukturgleichheit) auch die Gleichheit modulo interpretierten Funktionen und weiteren Prädikaten für spezielle Strukturen erlauben. Dies führt zu der Erweiterung der **Logikprogrammierung mit Constraints** (**Constraint Logic Programming, CLP**):

- Erlaube statt Termen auch Constraint-Strukturen, d. h. Datentypen mit einer speziellen festgelegten Bedeutung (im Gegensatz zu *freien* Termstrukturen).
- Ersetze die Unifikation durch spezielle Lösungsalgorithmen für diese Constraints.

Beispiel: CLP($\mathcal{R}$) bezeichnet die Erweiterung der Logikprogrammierung (LP) um Constraints (C) über reellen Zahlen ($\mathcal{R}$):

- Constraint-Struktur: reelle Zahlen (und Terme) und arithmetische Funktionen
- Constraints: Gleichungen/Ungleichungen zwischen arithmetischen Ausdrücken
- Lösungsalgorithmen: Gauß'sches Eliminationsverfahren (Gleichungen), Simplexmethode (Ungleichungen) und Unifikation (Terme)

Da die Implementierung dieser Lösungsalgorithmen (insbesondere in einer inkrementellen Variante) aufwändig ist, bietet nicht jedes Prolog-System diese Constraints an, aber diese stehen z.B. in SICStus-Prolog oder SWI-Prolog zur Verfügung:

```
?- use_module(library(clpr)).
?- { X=3+2 }.
X=5.0
?- { 5=3+X }.
X=2.0
```

Man beachte, dass arithmetische Constraints zur Unterscheidung der „normalen“ Unifikation in geschweifte Klammern geschrieben werden.

Als konkretes Beispiel betrachten wir die Berechnung von Hypotheken und deren Rückzahlung. Bei dieser Anwendung sind die folgenden Parameter relevant:

P: Hypothek-Gesamtbetrag (principal)

T: Laufzeit (in Monaten) (time)

IR: monatlicher Zinssatz (interest rate)

B: am Ende ausstehender Betrag (balance)

MP: monatliche Rückzahlung (monthly payment)

Ein Programm, das diese Parameter in die richtige Relation setzt, besteht aus zwei Regeln, wobei die erste Regel eine Laufzeit von maximal einem Monat beschreibt, und die zweite Regel eine Laufzeit von mehr als einem Monat rekursiv löst:

```
:- use_module(library(clpr)).

mortgage(P,T,IR,B,MP) :- { T>=0, T=<1, B = P * (1 + IR*T) - MP*T }.
mortgage(P,T,IR,B,MP) :- { T>1 },
                         mortgage(P * (1+IR) - MP, T-1, IR, B, MP).
```

Damit haben wir ein recht universell einsetzbares Programm, um verschiedene Hypothekenprobleme zu berechnen:

- Was muss monatlich zurückgezahlt werden, um eine Hypothek nach einer gegebenen Laufzeit komplett zu tilgen?

  ```
  ?- mortgage(100000,180,0.01,0,MP).
  MP=1200.68
  ```

- Falls ich einen festen Betrag monatlich zahlen kann, welche Zeit benötige ich, um eine Hypothek abzuzahlen?

```
?- mortgage(100000,T,0.01,0,1400).
T=125.901
```

- Was ist die Relation zwischen den Parametern `H, B, MR` bei einer festen Zeitdauer und Zinssatz?

```
?- mortgage(P,180,0.01,B,MP).
MP=0.012*P-0.002*B
```

Neben arithmetischen Strukturen sind auch weitere Constraint-Strukturen sinnvoll:

Boolsche Ausdrücke: Diese können beim Hardwareentwurf und der -verifikation eingesetzt werden.

Unendliche zyklische Bäume

Listen

Endliche Bereiche: Diese haben zahlreiche Anwendungen bei Planungsaufgaben und Optimierungsproblemen (Operations Research), z. B. bei der Containerverladung, Flottenplanung, Personalplanung, Produktionsplanung, etc. Dies ist daher eine der praktisch wichtigsten Erweiterungen von Prolog.

Das typische Vorgehen bei der Programmierung mit endlichen Bereichen ist wie folgt:

1. Definiere den endlichen(!) Wertebereich der Variablen („domain“).
2. Definiere die Constraints/Randbedingungen der Variablen.
3. Definiere ein Aufzählungsverfahren/Backtracking-Strategie für Variablen („labeling“).

Ohne dies weiter zu vertiefen, geben wir ein Beispiel hierfür an. Wir wollen das folgende kryptoarithmetische Puzzle lösen:

$$\begin{array}{r} \text{BASE} \\ +\ \text{BALL} \\ \hline \text{GAMES} \end{array}$$

Hierbei steht jeder Buchstabe für eine Ziffer, und unterschiedliche Buchstaben für unterschiedliche Ziffern. Mit CLP(FD) (Constraint Logic Programming over Finite Domains) kann man dieses Problem z.B. in SICStus-Prolog wie folgt lösen:

```
% Lade den Constraint-Löser für endliche Bereiche:
:- use_module(library(clpfd)).

crypt(Vs):-
   Vs = [B,A,S,E,L,G,M],  % Liste aller vorkommenden Buchstaben
   domain(Vs, 0, 9),      % Wertebereich festlegen
   B #> 0, G #> 0,        % Constraints festlegen
   all_different(Vs),
              1000*B+100*A+10*S+E
   +          1000*B+100*A+10*L+L
   #= 10000*G+1000*A+100*M+10*E+S,
   labeling([],Vs).       % Variablenwerte aufzählen

?- crypt([B,A,S,E,L,G,M]).
B = 7, A = 4, S = 8, E = 3, L = 5, G = 1, M = 9
```