

Proseminar Programmiersprachen

Prolog

Johannes Weißl

15. November 2006

Inhaltsverzeichnis

1	Einleitung	3
2	Einführung	3
2.1	Geschichte	3
2.2	Logische Programmierung	4
3	Grundlegende Sprachelemente	5
3.1	Fakten	5
3.2	Anfragen	6
3.3	Variablen	6
3.3.1	Variablen in Anfragen	6
3.3.2	Variablen in Fakten	7
3.4	Verknüpfte Anfragen	8
3.5	Regeln	8
3.6	Syntax	9
4	Weiterführende Konzepte	10
4.1	Datenstrukturen	10
4.2	Rekursion	10
4.2.1	Allgemein	10
4.2.2	Listen	11
4.3	Negation	12
5	Ausführung	12
5.1	Unifikation	12
5.2	Abstrakter Interpreter	14
5.3	Backtracking	15
6	Programm	16
7	Schluss	18
7.1	Implementierungen	18
7.2	Fazit und Ausblick	18
8	Literatur	19

1 Einleitung

Im Rahmen des Proseminars *Programmiersprachen* möchte ich die Sprache *Prolog* vorstellen. Sie ist eine bekannte und einflussreiche Vertreterin der *Logischen Programmiersprachen*.

Dieser Text ist hauptsächlich in vier Schwerpunkte gegliedert: In „Grundlegende Sprachelemente“ wird Prolog Schritt für Schritt an Beispielen ähnlich wie in einem Tutorial eingeführt. Im nächsten Kapitel werden „Weiterführende Konzepte“ wie Rekursion vorgestellt. Der nächste Inhaltsabschnitt behandelt in Grundzügen die Funktionsweise und die Techniken eines Prolog-Systems, wie *Unifikation*. Anschließend wird noch ein kleines Beispielprogramm gezeigt, das besondere Spracheigenschaften von Prolog demonstrieren soll.

2 Einführung

2.1 Geschichte

Anfang der 70er Jahre arbeitete ein französisches Team um *Alain Colmerauer* und *Philippe Roussel* in Marseilles an einem System zur Verarbeitung von natürlicher Sprache. Man sollte mit dem Computer in französisch kommunizieren können, und dieser sollte Schlussfolgerungen aus den gemachten Äußerungen ziehen.

Das System bestand hauptsächlich aus zwei Teilen: Einem linguistischen Regelsystem (geschrieben in *Q-systems*), das die Sprache in logische Formeln umwandelte, und einem Theorembeweiser, der diese Formeln ableitete (geschrieben in *Algol-W*).

Während seiner Forschung an automatischen Theorembeweisern wurde *Jean Trudel*, ein Mitglied des Teams, auf Arbeiten des Amerikaners *Robert Kowalski* zu diesem Thema aufmerksam. Dieser wurde daraufhin mehrmals nach Frankreich eingeladen, und lieferte wertvolle Beiträge für die Arbeiten.

Um diese Zeit erfuhr das Team von der Existenz der logischen Programmiersprache *Planner*. Da sie *Lisp* für ihr Projekt nicht einsetzen wollten, und ihnen *Planner* zu wenig formalisiert war, beschlossen sie, ihr System in einer eigenen logischen Sprache zu formulieren. Die Namenswahl fiel auf *Prolog* (für: „**P**rogrammation en **L**ogique“). Im Herbst 1972 wurde das Mensch-Maschine-System mit einer ersten Prolog-Version fertiggestellt, und schließlich 1973 die Sprache weiter verbessert, die Syntax spezifiziert und vereinfacht. (nach: Colmerauer und Roussel [2])

Allerdings stand ein Großteil der amerikanischen KI-Forscher der neuen Sprache kritisch gegenüber, logische Sprachen galten seit dem Fehlschlag von *Planner* als langsam und schwer beherrschbar. Auch dass gegen Ende der 70er Jahre ein erster Compiler von *David H. D. Warren* entwickelt wurde, der – selbst größtenteils in Prolog geschrieben –

durchaus mit den besten Lisp-Systemen dieser Zeit mithalten konnte, verhalf der Sprache nicht zu viel mehr Aufmerksamkeit und Anerkennung.

Dies änderte sich 1981 mit Japans Ankündigung des „*Fifth Generation Computer Systems*“ Projekts (FGCS). Diese Initiative hatte zum Ziel, eine völlig neue und revolutionäre Sorte von Computersystemen zu entwickeln. Dabei wurde für das Betriebssystem als *Kernel Language* eine Variante von Prolog eingesetzt. Obwohl das Projekt schließlich Anfang der 90er Jahre scheiterte, führte es als Nebeneffekt zu einer Popularisierung von Prolog und Logischer Programmierung im Allgemeinen. (nach: Sterling und Shapiro [3])

Seitdem sind viele Bücher über Prolog erschienen, und die Sprache ist nicht mehr nur ein Werkzeug für Wissenschaftler. Heute wird sie unter anderem bei der Erstellung von Expertensystemen und im Bereich des Systemmanagements eingesetzt, und natürlich in der Computerlinguistik und bei automatischen Theorembeweisern. (nach: Wikipedia [4])

2.2 Logische Programmierung

Logische Programmiersprachen wie Prolog oder auch funktionale wie Lisp oder OCaml unterscheiden sich stark von *imperativen Sprachen* wie C, C++ oder Java. Ein imperatives Programm in eine Folge von Befehlen, welche bestimmen *wie* es sich verhält. Die Art der Befehle orientiert sich deshalb an der Zielarchitektur, heutzutage meist die *Von-Neumann-Architektur*, auch wenn viele solcher Sprachen abstrahierende Konzepte wie Strukturierung von Quelltext oder Objektorientierung anbieten.

Logische Programmierung dagegen ist *deklarativ*, es wird nicht beschrieben, *wie* etwas gelöst werden soll, sondern *was*. Die Sprachmittel richten sich nicht nach einer konkreten technischen Maschine, sondern einer abstrakten Theorie. Nach dem Konzept der Logischen Programmierung ist ein Programm eine Menge von Axiomen, also festgelegten wahren Sätzen. Als Ausführung wird der Beweis eines Zielausdrucks aus den Programmaxiomen verstanden.

Prolog ist eine Umsetzung dieses Konzepts in eine reale Programmiersprache. Dabei mussten natürlich viele Designentscheidungen in Hinblick auf praktische Durchführbarkeit und reellen Nutzen getroffen werden, so dass sie, wie fast jede Sprache, nur eine Annäherung an das zugrundeliegende Programmierparadigma ist.

(nach: Sterling und Shapiro [3])

3 Grundlegende Sprachelemente

3.1 Fakten

Ein Programm in Prolog ist eine Sammlung von *Fakten* (engl. *facts*) und *Regeln* (engl. *rules*). Fakten sind Aussagen, die eine Relation zwischen Objekten ausdrücken. Beispiele für Fakten in natürlicher Sprache sind zum Beispiel:

- Marge ist weiblich.
- Homer ist ein Elternteil von Lisa.

In Prolog werden diese Fakten als *Prädikate* wie in der Prädikatenlogik angegeben, mit abschließenden Punkt:

```
female(marge).  
parent(homer,lisa).
```

Dabei sind `female` und `parent` Prädikate, `marge`, `homer` und `lisa` sogenannte *Atome*. Prädikate müssen mit einem Kleinbuchstaben beginnen, Atome entweder mit einem Kleinbuchstaben oder in einfache Anführungszeichen gesetzt sein.

Allgemein hat ein Prädikat folgenden Aufbau:

$$f(t_1, t_2, \dots, t_n)$$

- f : *Funktor*
- n : *Stelligkeit*
- t_1, \dots, t_n : *Argumente*

Ein Prädikat wird eindeutig über seinen Namen **und** seine Stelligkeit identifiziert, es könnte also zwei `parent` Prädikate mit verschiedener Anzahl von Argumenten im selben Programm geben. Man schreibt deshalb oft z.B. `parent/2`, um sich auf das richtige zu beziehen.

Um weitere Sprachkonstrukte besser demonstrieren zu können, folgt nun ein kleines Beispielpogramm, das die Familienbeziehungen der Simpsons durch Fakten darstellt.

Beispielprogramm P_S : Die Simpsons

```
female(marge).           parent(grandpa,homer).
female(lisa).            parent(marge,bart).
female(maggie).         parent(marge,lisa).
male(grandpa).          parent(marge,maggie).
male(homer).            parent(homer,bart).
male(bart).             parent(homer,lisa).
                        parent(homer,maggie).
```

3.2 Anfragen

An ein Prolog-System können *Anfragen* gestellt werden, die dann in Hinblick auf das Programm beantwortet werden. In diesem einfachen Beispiel muss nur überprüft werden, ob die Fakten so im Programm vorkommen. Die Reihenfolge der Argumente ist dabei von Bedeutung.

Beispiel:

```
female(lisa)?           parent(bart,homer)?
yes                  no
female(bart)?           parent(homer,bart)?
no                   yes
female(edna)?
no
```

Das Fragezeichen am Ende steht dafür, dass die Anfrage dem Prolog-System übergeben wird. Bei vielen Implementierungen gibt man sie stattdessen ebenfalls mit abschließenden Punkt in die Eingabeaufforderung ein.

3.3 Variablen

3.3.1 Variablen in Anfragen

In Prolog gibt es auch *Variablen*, diese unterscheiden sich jedoch erheblich von Variablen in gewöhnlichen imperativen Sprachen. Sie sind keine „Behälter“ für Speicherobjekte, die im Programmverlauf geändert werden können, sondern Platzhalter für Werte.

In Anfragen gibt das Prolog-System alle Belegungen der Variable aus, die eingesetzt in den Ausdruck mit „*yes*“ beantwortet werden würden.

Beispiel:

parent(marge,X)?
X = bart ;
X = lisa ;
X = maggie

parent(Y,maggie)?
Y = homer ;
Y = marge

Zur Unterscheidung von Atomen beginnen Variablen mit einem Großbuchstaben oder einem Unterstrich.

Die „Belegung der Variable“ nennt man auch *Substitution*, das Einsetzen in einen Ausdruck *Instanzieren*. Formal:

- **Substitution:** Menge $\theta = \{x_1 \mapsto t_1, \dots, x_i \mapsto t_i\}$
Die x_i sind paarweise verschieden: $x_i \neq x_j$ für $i \neq j$, und x_i darf in t_i nicht vorkommen.
Anwenden auf einen Ausdruck t (Ersetzen aller Vorkommnisse): $t\theta$
- **Instanz:** Resultat einer Ersetzung
 s ist Instanz von t , wenn θ existiert, so dass: $s = t\theta$

Beispiel:

t	=	parent(Y,maggie)
θ	=	{Y \mapsto homer}
<hr/>		
$t\theta$	=	parent(homer,maggie)

3.3.2 Variablen in Fakten

Variablen können auch in Fakten verwendet werden. Sie werden dann als *Allgemeingültige Fakten* (engl. *Universal Facts*) bezeichnet.

Beispiel Programm:
hates(X,mr_burns).

Beispiel Anfragen:
hates(maggie,mr_burns)?
yes

hates(foobar,mr_burns)?
yes

Wie man im Beispiel sieht, wird „passt“ jedes beliebige Atom im ersten Argument, auch wenn es bisher noch nicht verwendet wurde.

3.4 Verknüpfte Anfragen

Anfragen können mit einem Komma zu einer neuen Anfrage verknüpft werden. Wie in der Aussagenlogik wird dies als *Konjunktion* bezeichnet.

Beispiel:

```
parent(homer,bart), female(bart)?
```

no

```
parent(homer,bart), parent(marge,lisa)?
```

yes

Die Anfrage wird nur mit „**yes**“ beantwortet, wenn alle Teilanfragen wahr sind.

Wenn eine Variable in mehreren Teilanfragen verwendet wird, steht sie überall für denselben Wert. Man spricht dann von *gemeinsam benutzten Variablen* (engl. *Shared Variables*). Beispiel:

```
parent(homer,X), female(X)?
```

X = lisa ;

X = maggie

Im Beispiel wird nach allen Kindern von Homer gefragt, die auch weiblich sind, also den Töchtern von Homer.

3.5 Regeln

Wie eingangs erwähnt, besteht ein Prolog-Programm aus Fakten und Regeln. Mit Regeln (auch Sätze oder engl. *clauses*) lassen sich Anfragen, die sich als sinnvoll erwiesen haben, zu neuen Prädikaten zusammenfassen. Diese können dann in Anfragen oder wieder in neuen Regeln verwendet werden.

Formal werden Regeln mit dem Konditionalpfeil aus der Aussagenlogik notiert:

$$A \leftarrow B_1, B_2, \dots, B_n.$$

Gelesen: *A* falls *B*₁ und *B*₂ und ... und *B*_{*n*}.

Dabei wird *A* als der Kopf und *B*₁, *B*₂, ..., *B*_{*n*} als der Rumpf des Satzes bezeichnet. Fakten stellen eigentlich nur einen Spezialfall der Sätze dar, nämlich wenn *n* = 0. Dann wird natürlich auch der Konditionalpfeil nicht mehr geschrieben.

In der Aussagenlogik werden solche Formeln als *Hornklauseln* bezeichnet.

Beispiel:

```
father(X,Y) ← parent(X,Y), male(X).
```

Hier wird ein neues Prädikat definiert, das die Relation „Vater von“ zwischen X und Y festlegt. Die Bedeutung ist dabei: X ist ein Vater von Y, falls es ein X und Y gibt, so dass X ein Elternteil von Y ist und X männlich ist.

Man kann die Regel jedoch auch prozedural verstehen: Um die Frage zu lösen, ob X ein Vater von Y ist, muss man erst `parent(X,Y)` und dann `male(X)` lösen.

Mit diesem Prädikat lassen sich neue Regeln definieren:

```
grandfather(X,Y) ← father(X,Z), parent(Z,Y).
```

Hier wird eine Variable Z im Rumpf verwendet, die im Kopf nicht vorkommt. Wie auch bei den anderen Variablen ist der Geltungsbereich die gesamte Regel.

Anmerkung: Da Pfeile beim Eintippen des Programms in den Computer schwer zu erzeugen sind, verwenden die Prolog-Systeme meist die Zeichenfolge „:-“. Es wird hier jedoch weiterhin der Pfeil verwendet, da dies die Lesbarkeit erhöht.

3.6 Syntax

Nach dieser eher informellen Einführung in Prolog folgt nun eine formale Beschreibung der Syntax ähnlich der EBNF. Diese stellt allerdings nur einen Grundstock dar, in den meisten Prolog-Versionen gibt es noch mehr *syntactic sugar*.

```
program      = {clause}*
clause       = goal [ ":-" {term}+ ] "."
goal         = atom | compound-term
term         = constant | variable | compound-term
compound-term = atom "(" {term}+ ")"
              | term atom term
variable     = ("_" | upper-case-letter) {character}*
constant     = atom | number
atom         = no-upper-case-letter {character}*
              | "'" {character | space}+ "'"
```

Eckige Klammern markieren optionale Elemente, geschweifte Klammern Wiederholungen (*: 0 bis n, +: 1 bis n) und der senkrechte Strich Alternativen.

Wie man bei `compound-term` sieht, gibt es die Möglichkeit, zweistellige Prädikate als Infix-Operator zu schreiben.

4 Weiterführende Konzepte

4.1 Datenstrukturen

Als Argumente von Prädikaten können nicht nur unstrukturierte Atome verwendet werden. Beispiel:

Ein Fakt zur Beschreibung eines Proseminares (Zeit und Ort):

```
proseminar('Programmiersprachen', wednesday, 14, 15, 01, 11, 018).
```

Es ist oft übersichtlicher, bestimmte Argumente zu einem neuen zusammenzufassen:

```
proseminar('Programmiersprachen', time(wednesday, 14, 15), room(01, 11, 018)).
```

Dies hat den Vorteil, dass Regeln, die z.B. die Zeit nicht verwenden, nichts über deren genaue Repräsentation wissen müssen:

```
same_room(proseminar(_, _, X), proseminar(_, _, X)).
```

Regeln, die diese Information brauchen, können ohne weiteres darauf zugreifen:

```
same_floor(proseminar(_, _, room(X, -, -)), proseminar(_, _, room(X, -, -))).
```

Der Unterstrich steht für eine unbenannte Variable, die nicht weiter verwendet wird.

4.2 Rekursion

4.2.1 Allgemein

Ein mächtiges Werkzeug in Prolog ist Rekursive Programmierung. Dabei wird der Kopf einer Regel im Rumpf wieder verwendet. Beispiel:

```
ancestor(X, Y) ← parent(X, Y).  
ancestor(X, Y) ← parent(X, Z), ancestor(Z, Y).
```

X ist ein Vorfahr von Y, wenn es ein Elternteil von Y ist oder ein Elternteil von einem Vorfahren von Y. Das Notieren von mehreren Regeln mit gleichem Kopf drückt praktisch ein „oder“ aus. Es wird die Regel angewendet, die passt.

Bei Rekursion ist darauf zu achten, dass die Regeln nicht zirkulär definiert werden, da es sonst im schlimmsten Fall zum Abbruch der Ausführung wegen einer Endlosschleife kommen kann.

4.2.2 Listen

Listen sind rekursive Datenstrukturen, bestehend aus einem Kopf (engl. *head*), der das eigentliche Element darstellt, und einem Restteil (engl. *tail*), welcher wieder eine Liste ist.

In Prolog ist die leere Liste das Atom „[]“, das Listenpaar wird durch einen Punkt als Funktor gebildet: `.(Head,Tail)`. Es gibt jedoch eine gleichbedeutende syntaktische Variation: `[Head|Tail]`

Die Definition der Listen in Prolog:

```
list([]).  
list([H|T]) ← list(T).
```

Eine Liste der Zahlen 1, 2 und 3 würde also so dargestellt werden:

```
[1|[2|[3|[]]]] oder .(1,.(2,.(3,[])))
```

Da dies sehr umständlich zu schreiben ist, gibt es eine syntaktische Kurzform: `[1,2,3]`. Inhaltlich sind jedoch alle drei Formen gleich.

Eine wichtige Operation für Listen ist das Aneinanderhängen. Dazu dient das folgende Prädikat `append/3`.

```
append([], XL, XL).  
append([Y|YL], XL, [Y|ZL]) ← append(YL,XL,ZL).
```

Dabei drückt das Prädikat folgende Relation zwischen den drei Argumenten aus: Die dritte Liste ist die erste und die zweite aneinandergehängt. Anders wie z.B. in funktionalen Sprachen kann das Prädikat deshalb auf viele Arten benutzt werden, je nachdem bei welchem Argument eine freie Variable steht:

```
prefix(XL,YL) ← append(XL, RL, YL).  
Die Liste XL ist ein Präfix der Liste YL.
```

```
member(X,YL) ← append(AL, [X|XL], YL).  
Das Element X ist in der Liste YL enthalten.
```

Diese Prädikate lassen sich wieder auf mehrere Arten verwenden, z.B. entweder zum Überprüfen, ob eine Liste Präfix der anderen ist, oder zum Erzeugen aller Präfixe einer Liste. Diese mehrfache Anwendbarkeit von Prädikaten ist typisch für Prolog.

4.3 Negation

In vielen Prolog-Systemen gibt es ein vordefiniertes Prädikat `not/1`. Beispiel:

```
mother(X,Y) ← parent(X,Y), not(male(X)).
```

Dies ist allerdings **keine** echte logische Negation. Stattdessen ist `not(X)` genau dann wahr, wenn die Anfrage `X` mit **no** beantwortet werden würde. Man nennt dieses Verhalten auch *Negation as failure*.

Die Verwendung von `not` ist oft nützlich, jedoch muss darauf geachtet werden, dass die Variablen im Argument schon an einen festen Wert gebunden ist, da das Prädikat sonst immer fehlschlägt.

5 Ausführung

Nachdem die grundlegenden Sprachelemente und einige weiterführenden Programmierkonzepte besprochen wurden, folgt nun eine Beschreibung des Ausführungsmodells.

Um eine Anfrage beantworten zu können, muss das Prolog-System eine Regel finden, die auf die Anfrage passt. Ist eine solche Regel gefunden, wird mit dem eventuell vorhandenen Rumpf weiter gemacht.

Ob eine Anfrage auf den Kopf einer Regel „passt“ kann man feststellen, indem man versucht, beide Ausdrücke zu einem Ausdruck zu vereinheitlichen. Dieses Verfahren wird *Unifikation* genannt, und spielt eine große Rolle nicht nur in der Logischen Programmierung, sondern auch zum Beispiel bei automatischen Beweisern.

5.1 Unifikation

Das *Unifikationsproblem* $\{s =? t\}$ ist die Frage, ob sich die Ausdrücke s und t durch eine geeignete Ersetzung der enthaltenen Variablen syntaktisch gleich machen lassen. Die (syntaktische) Unifikation ist eine Methode zum Finden einer Substitution θ , so dass $s\theta = t\theta$. Diese Substitution nennt man dann auch *Unifikator*.

Ein Beispiel:

```
s = father(homer,C)
```

```
t = father(X,Y)
```

```
Unifikator  $\theta = \{X \mapsto \text{homer}, Y \mapsto C\}$ 
```

```
 $s\theta = t\theta = \text{father}(\text{homer}, C)$ 
```

Oft gibt es viele Unifikatoren, die ein Unifikationsproblem lösen. Meist ist man aber nur an dem kleinsten gemeinsamen Unifikator interessiert, dem *MGU* (engl. Most General Unifier). Der Name rührt daher, dass alle anderen Unifikatoren nur Instanzen des MGU sind:

σ ist ein MGU, wenn für alle anderen Unifikatoren θ eine Substitution δ existiert, so dass $\theta = \sigma\delta$. Allerdings kann es bei dieser Definition vorkommen, dass diese Gleichheit syntaktisch nicht exakt stimmt, wie man auch im Beispiel sieht:

Unifikationsproblem: $\{\text{male}(X) =? Y\}$

Ein Unifikator: $\theta = \{Y \mapsto \text{male}(Z), X \mapsto Z\}$

Ein MGU: $\sigma = \{Y \mapsto \text{male}(X)\}$

Es gilt: $\theta = \sigma\{Z \mapsto X\}$, wenn man die überflüssige Ersetzung $X \mapsto X$ ignoriert.

Doch wie findet man den MGU allgemein? Dazu gibt es vier Umwandlungen, die man auf ein Unifikationsproblem anwenden kann:

Unifikationsproblem: $S = \{s_1 =? t_1, \dots, s_n =? t_n\}$

- **Löschen** von überflüssigen Gleichungen:

$$\{t =? t, \dots\} \mapsto \{\dots\}$$

- **Aufspalten** von Prädikaten, um die Unifizierbarkeit der Argumente zu prüfen:

$$\{f(t_1, \dots, t_n) =? f(s_1, \dots, s_n), \dots\} \mapsto \{t_1 =? s_1, \dots, t_n =? s_n, \dots\}$$

Dies darf natürlich nur bei gleichen Prädikaten, also bei gleichem Namen und gleicher Stelligkeit geschehen.

- **Anwenden** von Substitutionen:

$$\{x =? t, \dots\} \mapsto \{x =? t\} \cup \{\dots\}\{x \mapsto t\}$$

Wenn x eine Variable ist, die in t nicht vorkommt, dann ersetze alle Vorkommen von x in ganz S durch t .

- **Wenden** von Konstanten nach rechts:

$$\{t =? x, \dots\} \mapsto \{x =? t, \dots\}$$

Dadurch wird es dann z.B. möglich, die obere Regel anzuwenden.

Mit diesen vier Regeln muss S solange umgewandelt werden, bis es die Form

$$S = \{x_1 =? t_1, \dots, x_n =? t_n\}$$

besitzt, wobei die Variablen x_i paarweise verschieden und nicht in t_i enthalten sind. Man nennt das Unifikationsproblem dann *gelöst* (engl. *solved form*). Es kann sowohl gezeigt werden, dass die Reihenfolge der Schritte unwichtig ist, als auch dass das Verfahren einen MGU findet, wenn es einen gibt.

Die Unifikation von zwei Ausdrücken ist jedoch nicht immer möglich, etwa bei diesen Beispielen:

1. $\text{male}(A) =? \text{female}(B)$
2. $\text{natural_number}(X) =? \text{natural_number}(s(X))$

Um Fälle wie im ersten Beispiel erkennen zu können, gibt es den **Clash-Test**:
 $S = \{f(t_1, \dots, t_n) =? g(s_1, \dots, s_m), \dots\}$, Abbruch wenn $f \neq g$ oder $n \neq m$

Bei Fall 2 hilft der **Occurs-Check**:
 $\{x =? t, \dots\}$, Abbruch wenn die Variable $x \neq t$ und x in t vorkommt

Dieser Occurs-Check wird aber bei vielen Prolog-Systemen mitunter angeblich aus Effizienzgründen ausgelassen. Dann führen Unifikationsprobleme wie im 2. Beispiel zu einer Endlosschleife.

(Unifikationsverfahren nach Baader und Nipkow [1])

5.2 Abstrakter Interpreter

Nachdem die Unifikation erläutert wurde, kann ein abstrakter Interpreter angegeben werden, der ein logisches Programm P ausführt:

- Gegeben: Programm P und Anfrage $q (= q_1, \dots, q_n)$
- Gesucht: Aus P ableitbare Instanz von q
- Vorgehen:

Resolvent $R := \{q_1, \dots, q_n\}$, $q' := q$

Solange bis R leer oder Auswahl fehlschlägt:

Wähle einen Zielausdruck r aus R

Wähle eine Regel $a \leftarrow b_1, \dots, b_n$ aus P , so dass $a\sigma = r\sigma$ (σ : MGU)

Ersetze r durch b_1, \dots, b_n in R

Wende σ auf R und q' an

Wenn R leer, gib θ ($q' = q\theta$) aus, **andernfalls no**

Der Interpreter initialisiert eine Menge R , den *Resolventen*, mit allen Teilanfragen aus q . Dann wählt er immer wieder mittels Unifikation zu einem Zielausdruck aus R eine passende Regel aus dem Programm, ersetzt den Zielausdruck (engl: *goal*) durch den

Rumpf der Regel und wendet die durch die Unifikation ermittelte Ersetzung auf *ganz R* und *q'* an. Da der Rumpf im Falle von Fakten leer sein kann, wird auch *R* leer, wenn die Anfrage aus dem Programm ableitbar ist. Dann ist *q'* die gesuchte Instanz von *q*, und es wird die Ersetzung ausgegeben, die von *q* nach *q'* führt.

Falls einmal keine geeignete Regel gefunden werden kann, bricht der Interpreter ab: Die Anfrage wird mit *no* beantwortet.

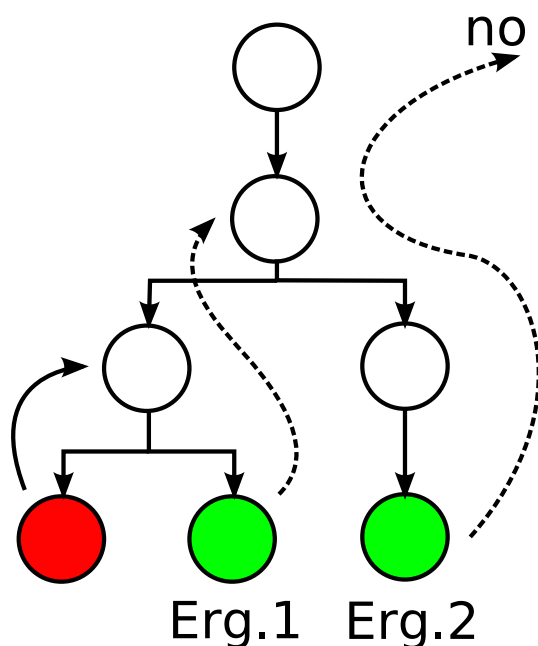
(nach: Sterling und Shapiro [3])

5.3 Backtracking

In dem abstrakten Interpreter kommt zweimal die Anweisung „Wähle“ vor. Erst soll ein Zielausdruck aus dem Resolventen gewählt werden. Dabei ist es vom logischen Standpunkt relativ unwichtig, in welcher Reihenfolge dies geschieht, da alle Anfragen mit „und“ verknüpft sind. In Prolog jedoch werden alle Teilanfragen von links nach rechts abgearbeitet. Dies ist zum Beispiel bei der Verwendung von nicht rein logischen Prädikaten wie Ein- und Ausgabe wichtig.

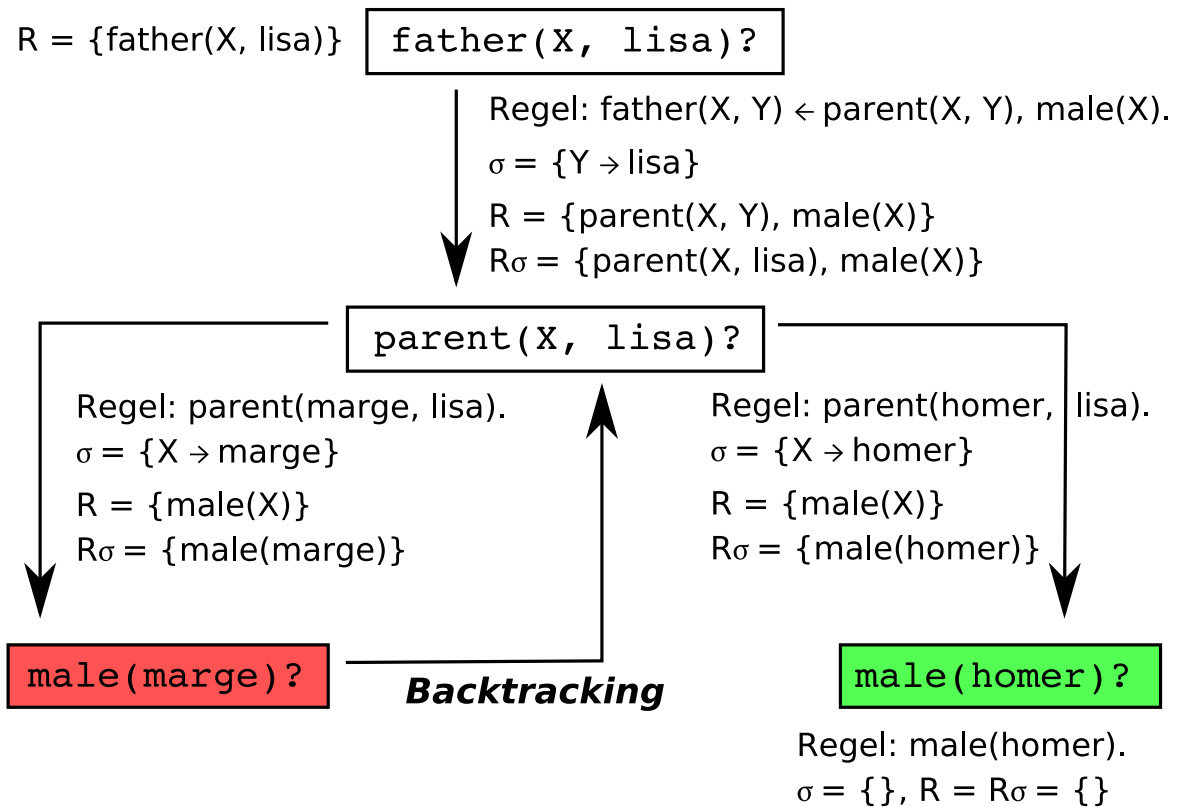
Die zweite Wahl ist ungleich schwieriger: Um aus allen syntaktisch passenden Regeln die richtige auswählen zu können, die am Ende zur Instanz führt, müsste er das Endergebnis schon kennen. Man bezeichnet dies auch als *Nichtdeterminismus*.

Es gibt viele Ansätze, um dieses Problem zu lösen. In Prolog verwendet man das Konzept der *Tiefensuche*. Die Sätze werden in der Reihenfolge abgearbeitet, in der sie in der Datei notiert sind. Bei einem Fehler, wenn also kein passender Satz gefunden werden kann, geht das System an die Stelle zurück, an der noch mindestens eine weitere Alternative vorhanden gewesen wäre, und probiert diese. Dieses Verhalten wird als *Backtracking* bezeichnet.



Die Grafik veranschaulicht das Backtracking. Der Weg durch das Programm wird als Baum dargestellt. Links unten stößt der Interpreter auf einen Fehler, deshalb geht er zurück und zur nächsten Alternative. Dies führt dann zu einem Ergebnis, also einer Instanz der Abfrage, die dann ausgegeben wird. Wenn man an weiteren Ergebnissen interessiert ist, kann der Interpreter das Backtracking auch fortsetzen, bis alle Ergebnisse ausgegeben wurden.

Hier eine Veranschaulichung zur Auswertung der Anfrage `father(X, lisa)`:



6 Programm

Als Beispielprogramm habe ich die Lösung eines logischen Rätsels gewählt, da hier besonders gut die deklarative und nichtdeterministische Programmierung sichtbar wird:

Franz und Karl sind Simpsons-Fans. Franz hat ein paar Sammelkarten und möchte Karl ärgern: „Wenn du es schaffst, 4 Karten nach folgenden Kriterien in eine Reihe zu legen, dann schenke ich dir die Karten“.

- Platz 4 soll ein Elternteil von Platz 3 sein
- Platz 2 und Platz 4 sollen das gleiche Geschlecht haben
- Platz 1 soll weiblich sein
- Hinter Lisa soll eine männliche Person sein, aber nicht Bart
- Auf Platz 2 soll nicht Grandpa sein

- Keine Karte soll doppelt vorkommen

Er hat aber nicht damit gerechnet, dass Karl als Informatikstudent die Sprache Prolog beherrscht, mit der er das Rätsel leicht lösen kann:

```

1 same_gender(X,Y) :- female(X), female(Y).
2 same_gender(X,Y) :- male(X), male(Y).
3
4 follow(A,B,[A,B|_]).
5 follow(A,B,[_|T]) :- follow(A,B,T).
6
7 unique([H|T]) :- not(member(H,T)), unique(T).
8 unique([]).
9
10 solve(X) :-
11     X = [A,B,C,D],
12     parent(D,C),
13     same_gender(D,B),
14     female(A),
15     follow(lisa,Z,X),
16     male(Z),
17     Z \= bart,
18     B \= grandpa,
19     unique(X).

```

Die Fakten über die Simpsons aus Programm P_S gehören natürlich ebenfalls in das Programm, falls das Rätsel gelöst werden soll.

Zur Erklärung:

Zuerst werden verschiedene Hilfsprädikate definiert, die alle ziemlich selbsterklärend sind. Das Prädikat `member(X,Y)` prüft, ob das Objekt `X` in der Liste `Y` vorkommt.

Das Prädikat `solve/1` schließlich löst das Rätsel, indem es durch das Aneinanderhängen der Bedingungen den Suchraum immer weiter eingrenzt. Durch Backtracking werden falsche Suchzweige verlassen, um zurückzukehren und neue Möglichkeiten zu testen. Dies bleibt jedoch hinter der deklarativen Programmierung verborgen.

Der Operator „`X = Y`“ bewirkt, dass das Prolog-System versucht, `X` und `Y` zu unifizieren. Die Negation davon, `X \= Y`, ist ähnlich wie `not` genau dann wahr, wenn die Unifikation fehlschlägt.

7 Schluss

7.1 Implementierungen

Es gibt viele freie und kommerzielle Implementierungen von Prolog. Hier folgt eine Liste von vier bekannten Open-Source Prolog-Systemen:

- **SWI-Prolog** (LGPL): sehr umfangreich, gut zum Lernen
<http://www.swi-prolog.org/>
- **YAP Prolog** (Artistic License): sehr schnell
<http://www.ncc.up.pt/~vsc/Yap/>
- **GNU Prolog** (GPL): nativer Compiler enthalten
<http://gnu-prolog.inria.fr/>
- **Ciao Prolog** (GPL/LGPL)
<http://www.clip.dia.fi.upm.es/Software/Ciao/>

7.2 Fazit und Ausblick

Wie man gesehen hat, ermöglicht Prolog eine – im Vergleich zu herkömmlichen Sprachen – neuartige Programmierung. Diese ist oft näher am Denkprozess des Menschen und erleichtert schnelle Entwicklung von Prototypen.

Im vollen Sprachumfang von Prolog sind noch viele weitere nützliche Funktionen enthalten, auf die in dieser Einführung nicht eingegangen werden konnte.

So gibt es auch einige Konstrukte, die die Performanz der Programme erhöhen, indem sie etwa Suchteilbäume abkürzen oder den Typ von Ausdrücken prüfen. Dadurch leidet jedoch oft die Klarheit und Deklarativität der verwendeten Sätze.

Auch Funktionen, um mit dem unterliegenden System zu interagieren, wie Ein- und Ausgabe, vertragen sich in Prolog eher mäßig mit dem Prinzip der logischen Programmierung.

8 Literatur

- [1] BAADER, FRANZ and TOBIAS NIPKOW: *Term Rewriting and All That*. Cambridge University Press, Cambridge, 1999.
- [2] COLMERAUER, ALAIN and PHILIPPE ROUSSEL: *The birth of prolog*. Draft of a paper in *History of Programming Languages*, edited by Thomas J. Bergin and Richard G. Gibson, ACM Press/Addison-Wesley, 1996, 1992. Available from: <http://www.lim.univ-mrs.fr/~colmer/ArchivesPublications/HistoireProlog/19november92.pdf>.
- [3] STERLING, LEON and EHUD SHAPIRO: *The Art of Prolog*. The MIT Press, Cambridge, London, second edition, 1994.
- [4] WIKIPEDIA. *Prolog (Programmiersprache)* — *Wikipedia, Die freie Enzyklopädie* [online]. 2006. Available from: http://de.wikipedia.org/w/index.php?title=Prolog_%28Programmiersprache%29&oldid=21280501.