Vision des logischen Programmierens

1. Prolog: "*programming in logic*".

2. Beschreibe gegebenes Problem durch Formeln.

   Prolog: Beschreibung durch

   (a) Fakten,

   (b) Regeln und

3. Benutze Automatischen Beweiser (Inferenz-Maschine) zur Beantwortung einer Anfrage.

   Kein Algorithmus erforderlich!

4. Das funktioniert nur in einfachen Fällen.

   Im allgemeinen muss man den Suchalgorithmus des Beweisers verstehen, um komplexe Probleme lösen zu können.

5. Trotzdem vorteilhaft:

   Programme sind Formeln, haben deshalb *deklarative Semantik* und sind daher (nach Gewöhnungsphase) leicht zu verstehen.

Prolog — Syntax

1. Variablen bestehen aus
   (a) Buchstaben
   (b) Ziffern
   (c) "_"

   Erster Buchstabe: Großbuchstabe oder "_".

   Beispiele:

   "X", "_", "A_Long_Variable_Name", "ABC", "_Y".

2. Funktions–Zeichen bestehen aus
   (a) Buchstaben
   (b) Ziffern
   (c) "_"

   Erster Buchstabe: Kleinbuchstabe

   Beispiele:

   "abc2", "a_XY", "etc", "x", "q", "q2", "+".

   Zusätzlich: Operator-Symbole

   "+", "-", "*", "/", ".", ...

   Außerdem: Zahlen

   1, -32, 0.5, 1.4e-6

Syntax

3. Prädikats–Zeichen bestehen aus

   Buchstaben, Ziffern, "_"

   Erster Buchstabe: Kleinbuchstabe

   Beispiele:

   "`abc2`", "`a_XY`", "`etc`", "`x`", "`q`", "`q2`".

   Außerdem: Operator-Symbole

   "`=`", "`\=`", "`<`", "`>`", "`=<`", "`>=`", ...

   Kontext: Unterscheidung

   Prädikats–Zeichen — Funktions–Zeichen

4. Fakten

   $p(s_1, \cdots, s_m)$.

   p: Prädikats-Zeichen, s_i: Terme

5. Klauseln (Regeln)

   $p(s_1, \cdots, s_m)$ `:-` $p_1(s_1^1, \cdots, s_{m(1)}^1)$`,` $\cdots$`,` $p_n(s_1^n, \cdots, s_{m(n)}^n)$.

   Interpretation

   $p_1(s_1^{(1)}, \cdots, s_{m(1)}^{(1)}) \wedge \cdots \wedge p_n(s_1^{(n)}, \cdots, s_{m(n)}^{(n)}) \rightarrow p(s_1, \cdots, s_m)$.

   Ersetzung:

   (a) "$\leftarrow$" durch "`:-`"

   (b) "$\wedge$" durch "`,`"

Programm: Beispiel

```
gallier(asterix).
gallier(obelix).

stark(X) :- gallier(X).

maechtig(X) :- stark(X).
maechtig(X) :- kaiser(X).

kaiser(caesar).
roemer(caesar).

spinnt(X) :- roemer(X).
```

Interpretation:

1. Asterix ist ein Gallier.
2. Obelix ist ein Gallier.
3. Gallier sind stark.
4. Wer stark ist, ist mächtig.
5. Wer Kaiser ist, ist mächtig.
6. Cäsar ist ein Kaiser.
7. Cäsar ist ein Römer.
8. Die Römer spinnen.

Knobeleien mit Prolog

1. Drei Freunde belegen den ersten, zweiten und dritten Platz bei einem Programmier-Wettbewerb.

2. Jeder der drei hat genau einen Vornamen, genau ein Auto und hat sein Programm in genau einer Programmier-Sprache geschrieben.

3. Michael programmiert in *Setl* und war besser als der Audi-Fahrer.

4. Julia, die einen Ford Mustang fährt, war besser als der Java-Programmierer.

5. Das Prolog-Programm war am besten.

6. Wer fährt Toyota?

7. In welcher Sprache programmiert Thomas?

Darstellung von Personen durch Prädikat `person/3`

`person`(*Name*, *Car*, *Language*)

1. *Name* $\in$ {`julia`, `thomas`, `michael`},

2. *Car* $\in$ {`ford`, `toyota`, `audi`},

3. *Language* $\in$ {`java`, `prolog`, `setl`}.

Vision des logischen Programmierens

1. Beschreibe gegebenes Problem durch Formeln.

   Prolog: Beschreibung durch

   (a) Fakten,

   (b) Regeln und

2. Benutze Automatischen Beweiser (Inferenz-Maschine) zur Beantwortung einer Anfrage.

   Kein Algorithmus erforderlich!

   Leider ist das nur eine Vision.

Vorheriges Beispiel:

Gibt es einen Mächtigen, der spinnt?

Anfrage an Prolog:

```
$ pl
Welcome to SWI-Prolog (Version 5.0.8)

1 ?- consult(gallier).
% gallier compiled 0.00 sec, 1,712 bytes

Yes
2 ?- maechtig(X), spinnt(X).

X = caesar
```

Listen-Notation in Prolog

Spezielle Funktions-Zeichen:

1. `.` : *Term* $\times$ *Term* $\rightarrow$ *Term*
2. `[]` : *Term*

Interpretation:

1. $l =$ `[]`: l ist leere Liste
2. $l = .(x, r)$:
   (a) x erstes Element von l
   (b) r restliche Elemente von l

Kurzschreibweisen:

1. `[a,b,c] := .(a, .(b, .(c, [])))`
2. `[` x `|` r `] := .(`x`,` r`)`

Beispiel:

```
concat( [], L, L ).
concat( [ X | L1 ], L2, [ X | L3 ] ) :-
     concat( L1, L2, L3 ).
```

Prolog – Was passiert

Prolog = Implementierung von $\rightsquigarrow_{\mathcal{P}}$:

Gegeben:

1. Prolog-Programm $\mathcal{P}$
2. Ziel G

Vorgehen: $\langle G, [] \rangle \rightsquigarrow_{\mathcal{P}}^{*} \langle \top, \tau \rangle$

τ ist *berechnete Antwort*.

Problem: $\rightsquigarrow_{\mathcal{P}}$ ist nicht-deterministisch:

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

q1(a).
q2(b).
```

Betrachte Ziel `p(X)`:

$\langle \texttt{p(X)}, [] \rangle \rightsquigarrow_{\mathcal{P}} \langle \texttt{q1(X)}, [] \rangle \rightsquigarrow \langle \top, [\texttt{X} \mapsto \texttt{a}] \rangle$

$\langle \texttt{p(X)}, [] \rangle \rightsquigarrow_{\mathcal{P}} \langle \texttt{q2(X)}, [] \rangle \rightsquigarrow \langle \top, [\texttt{X} \mapsto \texttt{b}] \rangle$

Die Substitutionen $[\texttt{X} \mapsto \texttt{a}]$ und $[\texttt{X} \mapsto \texttt{b}]$ sind beide richtig.

Frage: Welche liefert Prolog?

Antwort: Prolog wählt Klauseln in der Reihenfolge aus, in der sie im Programm stehen!

Scheitern von Zielen

Definition:
Ziel G_1 *scheitert unmittelbar* (bezüglich Programm $\mathcal{P}$) g.d.w.

$$\neg\exists G_2 : G_1 \rightsquigarrow_{\mathcal{P}} G_2$$

Beispiel: Gegebenes Programm $\mathcal{P}$:

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

Ziel: $G =$ `q(c).`

Problem: Unendliche Reduktionen

Beispiel: Gegebenes Programm `loop`:

```
loop :- loop.
loop.
```

Es gilt

$$\langle \texttt{loop}, [] \rangle \rightsquigarrow_{\mathcal{P}} \langle \texttt{loop}, [] \rangle$$

```
8 ?- consult(loop).
% loop compiled 0.00 sec, 652 bytes

Yes
9 ?- loop.
ERROR: Out of local stack
```

Prolog-Algorithmus

Gegeben:

1. Ziel G
2. Prolog-Programm $\mathcal{P}$

Gesucht: Substitution τ mit

$$\mathcal{P} \models G\tau$$

1. Setze $GL := [\langle G, [] \rangle]$.
2. Falls $GL == []$ ist, ist Ziel G *endgültig gescheitert*.
3. Es sei $GL = [Z|Zs]$
   Fallunterscheidung:
   (a) $Z = \langle \top, \tau \rangle$.
   Gebe Ergebnis τ aus.
   (b) Seien
   $$Z \rightsquigarrow_{\mathcal{P}} Z_i \text{ für } i = 1, \cdots, n$$
   alle mögliche Reduktionen von Z. Dann setze
   $$GL := [Z_1, \cdots, Z_n|Zs].$$
   Gehe zu Schritt 2.

Probleme des Prolog-Algorithmus

I. Nicht vollständig wegen unendlicher Reduktionen!
Beispiel:

```
loop :- loop.
loop.
```

Es gilt $\mathcal{P} \models$ `loop`,
aber Prolog-Algorithmus terminiert nicht!

II. Unifikation falsch: 1. Martelli-Montanari-Regel:

1. Falls $y \in \mathcal{V}$ und $\boxed{x \notin FV(t)}$ so gilt:

$$\langle E \cup \{y \doteq t\}, \sigma\rangle \rightsquigarrow \langle E[y \mapsto t], \sigma[y \mapsto t]\rangle.$$

Occur-Check: $x \notin FV(t)$ fehlt in Prolog.

Beispiel: Programm `occur`:

```
equal(Y,Y).
```

```
1 ?- consult(occur).
% occur compiled 0.00 sec, 504 bytes

Yes
2 ?- equal(X, s(X)).

X = s(s(s(s(s(s(s(s(s(s(...))))))))))
```