

Anwendungen der Logik: Deklarative bzw. Logik-Programmierung in PROLOG

Übersicht

1. Einführung
2. Beweissuche und Logik-Programmierung
3. PROLOG-Grundlagen
4. Beispiel

PROGRAMMIER-PARADIGMEN

1. Einführung

Es gibt nicht nur Befehle...

- Imperatives Programmieren
 - JAVA, ASSEMBLER, C, PASCAL, ...

PROGRAMMIER-PARADIGMEN

Es gibt nicht nur Befehle...

- Objekt-orientiertes Programmieren
 - üblicherweise als Erweiterung imperativer Sprachen
 - SMALLTALK, C++, JAVA, ...
- Funktionales Programmieren
 - LISP, SCHEME, ML, HASKELL, ...
- *Logik*-Programmierung
 - basiert auf Fragment der Prädikatenlogik
 - PROLOG, ...
- ...

LOGIK-PROGAMMIERUNG

“Algorithm = Logic + Control”

(Kowalski 1979)

Logik-Programmierung

Kontrollfluss — ohne Befehle?

- Programm = Axiome
- Ausführung = Konstruktiver Beweis einer Ziel-Formel

LOGIK-PROGAMMIERUNG

“Algorithm = Logic + Control”

(Kowalski 1979)

Logik-Programmierung

Kontrollfluss — ohne Befehle?

- Programm = Axiome
- Ausführung = Konstruktiver Beweis einer Ziel-Formel

Also:

- Kontrollfluss durch Beweissuche
- Berechnung durch Unifikation

LOGIK-PROGRAMMIERUNG

Vorteile

- deklarativ: Programm sagt, *was* es macht, *nicht wie*
- abstrahiert von der *von Neumann-Architektur* — ähnlich wie funktionales Programmieren
- somit "*high level*" Programmiersprache
- Korrektheit bzw. Verifikation leicht im Vergleich zu imperativen Programmen

LOGIK-PROGRAMMIERUNG

Anwendungen

- hauptsächlich in der *KI*
- Expertensysteme
- *planning*
- *natural language processing*
- ...

BEWEISSUCHE

2. Beweissuche und Logik-Programmierung

- Gegeben Axiome Γ , finde Beweis für

$$\Gamma \vDash \phi.$$

- Besonders interessant für Formeln der Form

$$\Gamma \vDash \exists x. P(x)$$

da konstruktive Beweise ein konkretes x geben.

- Beispiel: "Es gibt eine Liste X , die eine *geordnete* Permutation von $[2,3,1]$ ist."

BEWEISSUCHE

Implementierung

- Natürliches Schließen: für Menschen
- Resolution: für Computer
- volle Prädikatenlogik als Programmiersprache: zu hohe Komplexität

HORN-LOGIK

Horn-Klausel

$$\phi \leftarrow \psi_1 \wedge \psi_2 \wedge \dots \wedge \psi_n$$

- deklarative Sicht: ϕ gilt, wenn ψ_1 und ψ_2 und $\dots \psi_n$ gelten
- Klausel: $\{\phi, \neg\psi_1, \dots, \neg\psi_n\}$
- prozedurale Sicht: um ϕ zu lösen (auszuführen), löse erst $\psi_1 \dots \psi_n$

HORN-LOGIK

Resolution von Horn-Klauseln

$\phi \leftarrow \psi_1 \wedge \psi_2$	$\{\phi, \neg\psi_1, \neg\psi_2\}$
$\psi_1 \leftarrow \psi_2 \wedge \psi_3$	$\{\psi_1, \neg\psi_2, \neg\psi_3\}$
ψ_2	$\{\psi_2\}$
ψ_3	$\{\psi_3\}$

Beweis von ϕ ?

HORN-LOGIK

Resolution von Horn-Klauseln

$$\begin{array}{ll} \phi \leftarrow \psi_1 \wedge \psi_2 & \{\phi, \neg\psi_1, \neg\psi_2\} \\ \psi_1 \leftarrow \psi_2 \wedge \psi_3 & \{\psi_1, \neg\psi_2, \neg\psi_3\} \\ & \psi_2 \quad \{\psi_2\} \\ & \psi_3 \quad \{\psi_3\} \end{array}$$

Beweis von ϕ :

$$\begin{array}{ll} \{\neg\phi\} & [\phi] \\ \{\neg\psi_1, \neg\psi_2\} & [\psi_1, \psi_2] \\ \{\neg\psi_3\} & [\psi_3] \\ \emptyset & \square \end{array}$$

HORN-LOGIK

Kontrollfluss induziert von einer Horn-Klausel

$$\phi \leftarrow \psi_1 \wedge \psi_2 \wedge \dots \wedge \psi_n$$

- entspricht etwa

```
static f(...)
{
    p1(...);
    p2(...);
    ...
    pn(...);
}
```

- plus *Unifikation* und "backtracking"

LOGIK-PROGRAMME

- Fakten: Axiome/Datenbank
 - ϕ
 - triviale Horn-Klausel $\phi \leftarrow \top$
 - Beispiele: $\text{male}(\text{Isaac})$, $\text{father}(\text{Abraham}, \text{Isaac})$, ...
- Regeln: "echte" Horn-Klauseln
 - $\phi \leftarrow \psi_1 \wedge \psi_2 \wedge \dots \wedge \psi_n$
 - Beispiel: $\text{son}(x, y) \leftarrow \text{father}(y, x) \wedge \text{male}(x)$

LOGIK-PROGRAMME

Variablen in Horn-Klauseln

- quantorenfrei
- (implizit) universell quantifiziert:

$$\phi(x_1, \dots, x_n) \leftarrow \psi_1(x_1, \dots, x_n) \wedge \dots \wedge \psi_n(x_1, \dots, x_n)$$

steht für

$$\forall x_1. \dots \forall x_n. (\phi(x_1, \dots, x_n) \leftarrow \psi_1(x_1, \dots, x_n) \wedge \dots \wedge \psi_n(x_1, \dots, x_n))$$

LOGIK-PROGRAMME

Variablen in Horn-Klauseln

- quantorenfrei
- (implizit) universell quantifiziert:

$$\phi(x_1, \dots, x_n) \leftarrow \psi_1(x_1, \dots, x_n) \wedge \dots \wedge \psi_n(x_1, \dots, x_n)$$

steht für

$$\forall x_1. \dots \forall x_n. (\phi(x_1, \dots, x_n) \leftarrow \psi_1(x_1, \dots, x_n) \wedge \dots \wedge \psi_n(x_1, \dots, x_n))$$

- betrachte: $\text{grandfather}(x) \leftarrow \text{father}(x, y) \wedge \text{parent}(y, z)$
- Variablen, die nur "rechts" vorkommen, haben existenzielle Bedeutung:

$$\forall x. (\text{grandfather}(x) \leftarrow \exists y. \exists z. \text{father}(x, y) \wedge \text{parent}(y, z))$$

PROLOG

3. PROLOG-Grundlagen

- Fakten:
 - `male(isaac).`
 - `father(abraham, isaac).`

PROLOG

4. PROLOG-Grundlagen

- Fakten:
 - `male(isaac).`
 - `father(abraham, isaac).`
- Anfragen:
 - `?- male(issac).`
 - `?- father(haran, X).`

PROLOG

5. PROLOG-Grundlagen

- Fakten:
 - `male(isaac).`
 - `father(abraham, isaac).`
- Anfragen:
 - `?- male(issac).`
 - `?- father(haran, X).`
- bedeuten:
 - $\top \leftarrow \text{father}(\text{haran}, X)$
- sind also existenziell quantifiziert

BIBLICAL FAMILY DATABASE

```
father(terach, abraham).      male(terach).
father(terach, nachor).      male(abraham).
father(terach, haran).       male(nachor).
father(abraham, isaac).      male(haran).
father(haran, lot).          male(isaac).
father(haran, milcah).       male(lot).
father(haran, yiscah).
mother(sarah, isaac).        female(sarah).
                              female(milcah).
                              female(yiscah).
```

PROLOG

- Regeln:

```
son(X, Y) :- father(Y, X), male(X).
son(X, Y) :- mother(Y, X), male(X).

daughter(X, Y) :- father(Y, X), female(X).
daughter(X, Y) :- mother(Y, X), female(X).

grandfather(X) :- father(X, Y), parent(Y,Z).

parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

PROLOG

Rekursion

```
ancestor(X, Y) :- parent(X, Y).  
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

LISTEN

- Beispiele
 - [1, 2, 3]
 - []

LISTEN

- Beispiele

- [1, 2, 3]

- []

- *head* und *tail*

- [h | t]

- [1,2,3] \approx [1 | [2,3]] \approx [1 | [2 | [3 | []]]]

LISTEN

Rekursion

```
member(X, [X|Xs]).
```

```
member(X, [_|Xs]) :- member(X, Xs).
```

```
append([], L, L).
```

```
append([X|Xs], L, [X|L2]) :- append(Xs, L, L2).
```

WOLF-ZIEGE-KOHL-PROBLEM

6. Beispiel

PROLOG ist ideal zum Lösen von "*planning*" Problemen.

Wolf-Ziege-Kohl-Problem

- Wie transportiert man einen Wolf, eine Ziege und einen Kohl über einen Fluss mit einem winzigen Boot?

WOLF-ZIEGE-KOHL-PROBLEM

6. Beispiel

PROLOG ist ideal zum Lösen von "*planning*" Problemen.

Wolf-Ziege-Kohl-Problem

- Wie transportiert man einen Wolf, eine Ziege und einen Kohl über einen Fluss mit einem winzigen Boot?
- Nur einer der drei passt gleichzeitig ins Boot.
- Sowohl Wolf und Ziege als auch Ziege und Kohl dürfen nie gleichzeitig alleine an einem Ufer sein.

WOLF-ZIEGE-KOHL-PROBLEM

Repräsentation

- Mögliche Ufer-Positionen sind `empty`, `w`, `wg`, `wc`, `wgc`, `g`, `gc` und `c`
- Prädikat `cross(with, from, to, new from bank, new to bank)`
- PROLOG-Code:

WOLF-ZIEGE-KOHL-PROBLEM

Repräsentation

- Mögliche Ufer-Positionen sind `empty`, `w`, `wg`, `wc`, `wgc`, `g`, `gc` und `c`
- Prädikat `cross(with, from, to, new from bank, new to bank)`
- PROLOG code:

```
cross(wolf, wg, c, g, wc).
cross(wolf, wc, g, c, wg).
cross(goat, wg, c, w, gc).
cross(goat, wgc, empty, wc, g).
cross(goat, g, wc, empty, wgc).
cross(goat, gc, w, c, wg).
cross(cabbage, wc, g, w, gc).
cross(cabbage, gc, w, g, wc).
cross(alone, empty, wgc, empty, wgc).
cross(alone, wc, g, wc, g).
cross(alone, g, wc, g, wc).
```

WOLF-ZIEGE-KOHL-PROBLEM

Repräsentation der Transporte

- Prädikat pos(boat position, left bank, right bank)
- Prädikat move(position, passenger, new position)

WOLF-ZIEGE-KOHL-PROBLEM

Repräsentation der Transporte

- Prädikat pos(boat position, left bank, right bank)
- Prädikat move(position, passenger, new position)
- PROLOG-Code:

```
move(pos(left, Left, Right), Passenger, pos(right, L1, R1)) :-  
    cross(Passenger, Left, Right, L1, R1).
```

```
move(pos(right, Left, Right), Passenger, pos(left, L1, R1)) :-  
    cross(Passenger, Right, Left, R1, L1).
```

WOLF-ZIEGE-KOHL-PROBLEM

Lösung — erster Versuch

- Prädikat `solve(position, moves)`, mit `moves` als Liste der Passagiere.
- in PROLOG:

WOLF-ZIEGE-KOHL-PROBLEM

Lösung — erster Versuch

- Prädikat `solve(position, moves)`, mit `moves` als Liste der Passagiere.
- in PROLOG:

```
solve(pos(right, empty, wgc), []).  
solve(Position, [Move | SubSolution]) :-  
    move(Position, Move, Pos2),  
    solve(Pos2, SubSolution).
```

WOLF-ZIEGE-KOHL-PROBLEM

Lösung mit "Gedächtnis"

- es gibt Lösungen ohne Zyklen

WOLF-ZIEGE-KOHL-PROBLEM

Lösung mit "Gedächtnis"

- es gibt Lösungen ohne Zyklen
- Prädikat `solve(position, cycle-free moves)`
- Prädikat `solve(position, history, moves)`
- Positionen in der "*history*" Liste sind verboten

WOLF-ZIEGE-KOHL-PROBLEM

Lösung mit "Gedächtnis"

- es gibt Lösungen ohne Zyklen
- Prädikat `solve(position, cycle-free moves)`
- Prädikat `solve(position, history, moves)`
- Positionen in der "history" Liste sind verboten
- in PROLOG:

```
solve(Start, Solution) :-
    solve(Start, [Start], Solution).

solve(pos(right, empty, wgc), History, []).
solve(Position, History, [Move | SubSolution]) :-
    move(Position, Move, Pos2),
    \+ member(Pos2, History),
    solve(Pos2, [Pos2 | History], SubSolution).
```

Literatur

- [1] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, 2nd Edition, Springer Verlag, 1984.
- [2] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, The MIT Press, 1986.