Logické programování

Petr Štěpánek

S využitím materialu Krysztofa R. Apta

2006

Prolog a Logické programování

Prolog vznikl jako programovací jazyk v Marseille 1970, jeho autorem je Alain Colmerauer, implementován byl jeho skupinou.

Logické programování jako obecnější interpretaci odvozovacího mechanismu Prologu v predikátové logice zavedl Robert Kowalski v roce 1974.

V obou případech se jedná o Deklarativní programování

Co můžeme čekat od Prologu? V příkladech.

Příklad 1. Databáze leteckých spojení.

```
direct(amsterdam, seattle).
direct(amsterdam, paramaribo).
direct(seattle, anchorage).
direct(anchorage,fairbanks).

connection(X,Y) :- direct(X,Y).
connection(X,Y) :- direct(X,Z), connection(Z,Y).
```

Dotazy: Je spojení z Amsterodamu do Fairbanks?

```
?- connection(amsterdam,fairbanks).
yes
```

Kam se dá letět ze Seattlu ?

```
?-  connection(seattle,X).
X = anchorage ;
X = fairbanks ;
no
```

Dá se někam letět z Fairbanks?

```
?- connection(fairbanks,X).
no
```

Poučení: - stejný program může odpovídat na různé dotazy

- program může sloužit jako databáze (zde dokonce v Datalogu)

Příklad 2. Hledání společných prvků dvou seznamů. (po jednom prvku)

```
member(X,[X|List]).
member(X,[Y|List] :– member(X,List).
member_both(X,L1,L2) :-
                    member(X,L1), member(X,L2).

?- member_both(X,[1,2,3],[2,3,4,5]).
X = 2 ;
x = 3 ;
no
```

Testování

```
?- member_both(2,[1,2,3],[2,3,4,5]).
yes
```

Nalezení vhodné instance

```
?-  member_both(2,[1,2,3],[X,3,4,5]).
X = 2
```

Příklad 3. Zvláštní čtverce.

Druhá mocnina čísla 45 je 2025. Když rozdělíme cifry na dvě stejné části, dostaneme 20 a 25, a platí 20 + 25 = 45.

Hledání podobných čísel v Prologu je pouhé rozepsání definice. K tomu potřebujeme dvě proměnné N, Z pro hledáné číslo a pro jeho druhou mocninu. Program by mohle vypadat třeba takhle:

```
sol(N,Z) :-
        between(10,99,N),
        Z is N*N,
        Z ≥ 1000,
        Z // 100 + Z  mod 100 =:= N.
```

Relace `between(X,Y,Z)` platí když číslo `Z` je mezi `X` a `Y`, tedy když $X \leq Z \leq Y$.

Řešení:

```
?- sol(N,Z).
N = 45,
Z = 2025;
N = 55,
Z = 3025;
N = 99,
Z = 9801;
no
```

Jiná využití programu:

```
?- sol(55,3025).
yes

?- sol(55,Z).
Z = 3025
yes

?- sol(44,Z).
no
```

Problémy s Prologem

- každý program (nejen v Prologu) může být použit špatným způsobem.

- aritmetika v Prologu je neohrabaná a může vést k mnoha problémům.

- některé programy v Prologu jsou beznadějně neefektivní.

a) zakončování výpočtů (smyčky):

K příkladu 1 stačí přidat fakt

```
direct(seattle,seattle).
```

přidáme-li ho na konec programu, dotaz

```
?- connection(seattle,X).
```

generuje opakovaně všechna řešení.

Naopak, přidáme-li tento fakt na začatek, dostaneme

```
?- connection(seattle,X).
X = seattle ;
X = anchorage;
X = seattle ;
X = anchorage;
```

atd.

Odpověď `X = fairbanks` nebude nikdy generována.

Vyměníme-li pořadí pravidel definujících relaci `connection` a fakt

```
direct(seattle,seattle).
```

přidáme na konec programu, opakovaně je generována jen jedna odpověď.

```
?- connection(seattle,X).
X = fairbanks ;
X = fairbanks ;
```

atd.

Uvedené příklady ukazují jak je snadné generovat nekončící výpočty.

b) Konflikt proměnných v literatuře známý pod anglickým názvem ‘Occur-check Problem’. Jestliže ve druhém příkladu testujeme zda proměnná X se vyskytuje v seznamu sestávajícímu ze dvou položek `f(X)` a `X` dotazem `?- member(X,[f(X),X])`, dostaneme nekončící výpočet, který generuje „odpověď “

```
X = f(f(f(f(f( ..
```

V Prologu proměnné nabývají hodnoty prostřednictvím unifikačního algoritmu. Ten je pro zrychlení výpočtu zjednodušen vynecháním tak-zvaného testu výskytu (konfliktu) proměnných. Unifikační algoritmus nezjistí, že proměnná `X` je podslovem výrazu `f(X)` a snaží se oba výrazy unifikovat dosazováním `f(X)` za `X`.

c) Aritmetika je teorie mimo logiku a k implementaci Prologu byla přidána velmi okleštěným způsobem. Ve třetím příkladu jsme záměrně vynechali definici relace `between`.

Kdybychom se i zde opírali o deklarativní charakter programů, očekávali bychom, že tato relace bude definována pravidlem

```
between(X,Y,Z) :- X ≤ Z , Z ≤ Y.
```

Ovšem relace ≤ použitá v definici je v aritmetice Prologu vyhodnocena jen v případě, že oba argumenty jsou přirozená čísla.

Proto v Prologu musíme tuto relaci definovat programem (předpokládáme že v okamžiku volání relace `X ≤ Y` jsou hodnoty obou proměnných přirozená čísla, a to je v daném příkladu splněno).

```
between(X,Y,Z):- X ≤ Y, Z is X.
between(X,Y,Z):- X < Y, X1 is X+1,between(X1,Y,Z).
```

V aritmetických výrazech nelze použít složitější aritmetické termy

```
?- between(X,X+10,Z).
{INSTANTIATION ERROR: in expression}
```

d) Neefektivnost. Pravidlo

```
sorted(List,Out) :-
        permutation(List,Out),ordered(Out).
```

Je jednoduchým programem, který má uspořádat seznam přirozených čísel.

Jeho neefektivnost je na první pohled zřejmá, je to program typu „generuj a testuj“. Přesto je velmi oblíbený jako standard při testování efektivnosti různých implementací.

Není obtížné najít další příklady neefektivních Prologovských programů. Stačí jen přepsat některé definice relací jako program.

Tyto příklady ukazují, že chceme-li psát efektivní programy v Prologu, nestačí jen „myslet deklarativně“. Je třeba něco vědět o procedurální interpretaci programů, tedy o tom, jakým způsobem Prolog vyhodnocuje dotazy.

Logické programování. Proč počítá Prolog ?

Výpočty Prologu jsou prováděny pomocí strojového dokazování ve fragmentu logiky prvního řádu. Logické programování je teorie, která se touto složkou podrobně zajímá.

Prolog ovšem obsahuje rysy, které přesahují rámec logiky prvního řádu, mají charakter operátorů, to znamená že jsou to funkce, které pracují s formulemi, například

neúspěch `fail` konec prohledávání stavového prostoru odvozených formulí.

řez `!` dovoluje definovat jistý druh negace

```
not(p(X)) :- p(X),! , fail.
not(p(X)).
```

volání `call(p(X))` procedury na výpočet dotazu na `p(X).`

seznam všech řešení `findall(...,List)`

Jde o operátory, které lze přidat přidat při implementaci (někdy velmi jednoduše). Navíc, je přidána Aritmetika, která není součástí logického odvozování. Těmito operátory se zatím nebudeme zabývat.

Logické programování je teorie, která podrobně zkoumá logické odvozování v Prologu. Je poněkud obecnější než samotný Prolog.

Značení a konvence.

Jazykobsahuje

- proměnné x, y, z, x_1, x_2 , x_3 , ...
- logické spojky $\leftarrow$ (implikace) a , (konjunkce)

- funkční symboly f, g, h, f_1, f_2, . . .
- predikátové symboly p, q, r, p_1, p_2, . . .

Syntax

a) termy (jako obvykle)

(i) každá proměnná je term.

(ii) je-li f n-ární funkční symbol a $t_1, t_2, \ldots, t_n$ jsou termy, potom $f(t_1, t_2, \ldots, t_n)$ je term.

Abychom odlišili termy od termů v jazyku Prologu,

- používáme jiné písmo,
- proměnné označujeme malými písmeny

b) formule

(i) je-li p n-ární predikátový symbol a $t_1, t_2, \ldots, t_n$ jsou termy, potom $p(t_1, t_2, \ldots, t_n)$ je atomická formule, říkáme jí atom.

(ii) jsou-li $H, B_1, B_2, \ldots, B_n$ atomy potom

$$H \leftarrow B_1, B_2, \ldots, B_n$$

je formule, říkáme jí definitní klauzule.

Atomu H říkáme *hlava* a n-tici atomů $B_1, B_2, \ldots, B_n$ říkáme *tělo* klauzule. Někdy píšeme krátce $H \leftarrow \mathbf{B}$ (v angličtíně je to dokonce mnemotechnické).

(iii) jsou-li $A_1, A_2, \ldots, A_n$ atomy, říkáme této n-tici dotaz. Někdy píšeme krátce **A** .

Z logického nadhledu.

Formule a termy, které jsme definovali, patří do fragmentu predikátové logiky prvního řádu, které se říká Hornova logika.

Výpočet logického programu provádí strojový dokazovač resoluční metodou. Resoluční metoda dokazování postupuje dokazováním sporu.

Z logiky připomeňme:

Je-li T množina formulí (předpokládáme, že je konečná) a S je formule (můžeme předpokládat, že je uzavřená), víme že platí

$T \vdash S$ právě když $T \cup \{\neg S\}$ je sporná množina

Spornou množinu formulí na pravé straně označíme T´. Můžeme předpokládat, že T´ sestává z jediné formule, sestrojíme-li konjunkci všech jejích formulí.

Můžeme předpokládat, že T´ je uzavřená formule (Věta o uzávěru) a že je v prenexním tvaru (Věta o prenexních tvarech).

Máme tedy formuli tvaru

$$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n B \qquad (1)$$

kde B je otevřená formule a Q_i pro i , $1 \leq i \leq n$ jsou kvantifikátory.

Existenční kvantifikátory v prefixu můžeme eliminovat pomocí Skolemovských funkcí, například

Formuli $\forall x \exists y\, A(x,y)$ můžeme nahradit formulí $\forall x\, A(x,h(x))$, pokud rozšíříme jazyk o nový funkční symbol h a přidáme axiom A(x,h(x)).

Připomeňme Skolemovu větu, každou teorii lze konzervativně rozšířit do otevřené teorie T´´, (kterou lze v našem případě představit jako jedinou formuli).

Formuli T´´ lze uzavřít a opět převést do tvaru podobného formuli (1)

$$\forall x_1 \forall x_2 \ldots \forall x_m B' \qquad (2)$$

Teď zbývá převést otevřenou formuli B´ do konjunktivního normálního tvaru.

Pro úplnost definujeme literály a klauzule , literál je atomická formule (atom) nebo negace atomu. Klauzule je disjunkce literálů.

Protože platí $\vdash \forall x(C \,\&\, D) \leftrightarrow (\forall x\, C \,\&\, \forall x\, D)$

můžeme universální kvantifikátory z prefixu formule (2) distribuovat k jednotlivým klauzulím. Nakonec dostaneme konjunkci klausulí tvaru

$$(\forall \ldots)(L_1 \vee L_2 \vee \ldots \vee L_k)$$

Protože konjunkce je asociativní, a tedy na uzávorkování nezáleží, můžeme konjunkci klausulí chápat jako množinu klauzulí. Prefix universálních kvantifikátorů se obvykle nepíše.

Implicitní universální uzávěr dává klausuli řadu vlastností:

• platí-li klausule, potom platí každá její instance,

• přejmenováním proměnných (některých nebo všech) vznikne varianta dané klausule, která je s ní ekvivalentní,

• použijeme-li ve dvou různých klausulích stejnou proměnnou, tyto proměnné kromě jména spolu nemají nic společného, každou z nich lze přejmenovat. Při výpočtu mohou nabývat různých hodnot.

Hornova logika

V logickém programování používáme jen klausule speciálního typu (Hornovy klauzule). Jsou to klausule, které mají nejvýš jeden positivní literál. Mohou mit dva tvary: jsou-li $H, B_1, B_2, \ldots, B_n$ atomy , potom klausule

$$c \equiv H \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_n$$

je ekvivalentní implikaci

$$\forall x_1 \forall x_2 \ldots \forall x_k \; (B_1 \;\&\; B_2 \;\&\; \ldots \;\&\; B_n \rightarrow H) \qquad (3)$$

kde $x_1, x_2, \ldots, x_k$ jsou všechny proměnné v klausuli c . Takové klausuli říkáme definitní. A stejně nazýváme implikaci v závorce formule (3).

Podobně klausule

$$c' \equiv \neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n$$

je ekvivalentní formuli

$$\neg \exists x_1 \exists x_2 \ldots \exists x_k \, (A_1 \,\&\, A_2 \,\ldots\, A_n) \qquad (4)$$

Formuli (4) chápeme jako negaci dokazované formule

$$\exists x_1 \exists x_2 \ldots \exists x_k \, (A_1 \,\&\, A_2 \,\ldots\, A_n) \qquad (5)$$

která je dotazem, zda existuje instance kvantifikovaných proměnných, která by splňovala všechny podmínky v konjunkci formule (5).

V logickém programování používáme čárku místo symbolu & pro konjunkci a implikační šipku $\leftarrow$ místo :- .

Definice

- *Dotaz* je konečná posloupnost atomů ,
- *Klausule* je výraz tvaru $H \leftarrow \mathbf{B}$ kde H je atom a $\mathbf{B}$ je dotaz , atom H nazýváme hlavou a výraz $\mathbf{B}$ tělem klauzule ,
- *Program* je konečná množina klauzulí.

V zápise vynecháváme kvantifikátorové prefixy a obrácený směr implikační šipky je motivován tím, že je-li například $H \equiv p(t)$, klauzuli $H \leftarrow \mathbf{B}$ chápeme jako součást definice predikátu p .

Při analýze důkazů budeme ještě používat výrazy tvaru

$$\mathbf{A} \leftarrow \mathbf{B}$$

kde **A** i **B** jsou dotazy. Takové výrazy budeme nazývat *rezultanty* .

Značení

Atomy označujeme symboly A, B, C, H ...

Dotazy značíme Q, **A**, **B, C,** ...

Klausule c, d, ...

Rezultanty R ...

Programy P ...

Speciálně prázdný dotaz označujeme symbolem [] a je-li **B** prázdný dotaz, místo $H \leftarrow$ **B** píšeme $H \leftarrow$ tomuto výrazu říkáme jednotková klauzule.

V našem případě, je-li dán program P a dotaz Q rezoluční výpočet začíná dotazem $Q \equiv A_1, A_2, \ldots, A_n$ s vybraným atomem A_i pro nějaké i, $1 \leq i \leq n$. Je-li v programu klauzule $H \leftarrow B_1, \ldots, B_k$ taková, že její hlava se shoduje s vybraným atomem, odvodíme nový dotaz (rezolventu) Q_1 tvaru

$$A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1} \ldots, A_n \qquad (6)$$

V obecném případě, hledáme unifikační substituci θ takovou, že po její aplikaci na oba atomy dojde ke shodě, tedy $H\theta \equiv A_i\theta$. V takovém případě rezolventa (6) vypadá takto

$$(A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1} \ldots, A_n)\theta \qquad (7)$$

kde do každého atomu dotazu (6) dosadíme za jeho proměnné podle substituce

$$\theta = \{x_1/t_1, \ldots, x_k/t_k\}$$

kde (t_i jsou termy).

Úspěšný výpočet pokračuje stejnými kroky až do odvození prázdného dotazu [] . Přitom sestrojíme složenou substituci

$$\theta_1 \theta_2 \ldots \theta_m \qquad (8)$$

která dává odpověď, jaké termy máme dosadit za proměnné dotazu Q , aby všechny jeho atomy byly splněny.

Přirozeně, ne každý dotaz a každý program vedou k úspěšnému výpočtu. Ten může skončit neúspěchem.

Při volnosti výkladu, kterou jsme si dovolili, při úspěšném výpočtu můžeme prázdný dotaz považovat za prázdnou konjunkci a ztotožnit ji s Booleovskou konstantou *true* (všechny její atomy jsou splněny).

Nebo můžeme prázdný dotaz považovat za prázdnou disjukci a ztotožnit ji s hodnotou *false* (žádný její atom není splněn). A považovat výpočet za důkaz sporu. Substituci dostaneme (8) v obou případech.

Tento exkurs do predikátové logiky jsme si dovolili především proto, abychom ukázali, že logické programování má solidní základ v logice a také proto, abychom ukázali, co nás čeká.

Pro začátek je nejdůležitější si uvědomit, že v logickém programování nabývají proměnné svých hodnot speciálními substitucemi a úspěšný výpočet vydává složenou substituci, která obsahuje řešení problému formulovaného v dotazu Q .

Rezoluční metodu vytvořil J. A. Robinson v roce 1962 pro množiny obecných klauzulí. Vyřešil problém, jaké substituce je třeba používat a vytvořil algoritmus, jak je sestrojit.

A. Horn je v rezoluci nevinně, z jeho výsledků dosažených již v roce 1951 mimo jiné vyplývá, že je-li každá klausule v programu bezesporná, potom je bezesporný celý program. Spor může být do úlohy vnesen jen dotazem.