Olomoucký informatický korespondenční seminář – Olinx

Programovací jazyk PROLOG, 4. díl

Miroslav Kolařík

olinx.inf.upol.cz

Úvod

V posledním dílu našeho semináře o PROLOGu se nejprve budeme podrobněji věnovat tomu, jak probíhá výpočet v PROLOGu. Poté se budeme zabývat řezy. S využitím řezů si představíme některé základní třídící algoritmy. Závěrem se krátce zmíníme o negaci.

Výpočet v PROLOGu

Interpret PROLOGu se během výpočtu snaží splnit zadaný dotaz, přitom probírá jednotlivá pravidla v přesně daném pořadí. Vhodné pořadí pravidel může zvýšit efektivitu výpočtu. Abychom pochopili, jak PROLOG funguje, nastíníme si podstatné principy.[1]

Předpokládejme, že máme PROLOGovský program, který označíme písmenem `P`. Zhruba řečeno, položí-li uživatel dotaz `D` probíhá výpočet takto. Překladač PROLOGu přidá k `P` negaci dotazu `D` a snaží se z této množiny formulí odvodit (rezoluční metodou) spor.[2] Oznámí-li PROLOGovský překladač po zadání dotazu `D` na program `P` odpověď „true“, znamená to, že překladač odvodil z `P`, ¬`D` spor. Průběh výpočtu je přitom založen na rezolučním zásobníku – tak je hledání odpovědi implementováno.

Obecná rezoluční metoda byla navržena v 60. letech 20. století Robinsonem.[a] Základem je rezoluční odvozovací pravidlo, které ze dvou formulí odvodí jinou formuli, jejich tzv. rezolventu. Používá se přitom nejobecnější možná unifikace.[b] PROLOG používá speciální formu rezoluční metody, tzv. SLD-rezoluci, která velmi efektivně a deterministicky (jednoznačně) pracuje s dotazy, fakty a pravidly.

[a] John Alan Robinson žil v letech 1930 až 2016. Významně přispěl v oblasti automatizovaného uvažování.

[b] Unifikace je substituce, po jejíž aplikaci přejdou všechny formule ve stejnou formuli.

Překladač PROLOGu musí být deterministický. Musí jednoznačně vědět, co v dané situaci a v danou chvíli dělat.

- Pokud dostane dotaz, který sestává z více podčástí, pak je třeba zajistit deterministický výběr dílčích částí dotazu. Je-li například položen dotaz ve tvaru:

```
?- D1, D2, ..., Dn,
```

  vidí jej překladač jako uspořádaný zleva doprava. Při jeho zpracování bere vždy první poddotaz zleva.
- Při pohledu na program (na fakta a pravidla) je determinismus zajištěn uspořádáním shora dolů. Překladač tedy použije první možný fakt, který lze použít, respektive první možné pravidlo, které lze pro další průběh výpočtu použít.
- Je-li zadán dotaz s proměnnými, musí být odpovědi co možná nejobecnější. Aby se žádné řešení neztratilo používá překladač PROLOGu algoritmus na nalezení nejobecnějšího unifikátoru, který po konečně mnoha krocích buď vrátí nejobecnější unifikátor, nebo odpoví, že to nejde.

Pro zachycení všech možných větví výpočtu zavádíme tzv. SLD-stromy.[3] Překladač PROLOGu je prohledává podle algoritmu zásobníku. Prohledávání do hloubky (anglicky depth first search) je výpočetně efektivní (má lineární paměťovou složitost), avšak výpočet se může vydat po nekonečné větvi, přičemž existující řešení nemusí být nalezeno, i když se v SLD-stromu nachází.

Základní fáze výpočtu PROLOGu

1) Inicializace

- počáteční dotaz (též tzv. cíl) je kořen procházeného SLD-stromu,
- SLD-strom se prochází do hloubky (celý strom se neudržuje v paměti),
- podle směru pohybu ve stromu dělíme na přímý chod, respektive zpětný chod.

2) Přímý chod

- průchod SLD-stromem shora dolů (v programu zleva doprava),
- přímý chod představuje jeden úspěšný elementární krok odvození,
- speciální případ přímého chodu: sestup po nekonečné větvi (obecně nelze nijak detekovat).

3) Zpětný chod (navracení – backtracking)

- průchod SLD-stromem zdola nahoru (v programu zprava doleva),
- zpětný chod je návratem k předcházejícímu cíli,
- zpětný chod je vyvolán buď snahou dostat se ven z neuspívající větve (větev, ve které není řešení), nebo snahou najít alternativní řešení (řešení ve větvi je, ale chceme najít jiné).

Programy v PROLOGu je nutné psát s ohledem na to, že SLD-strom se prohledává do hloubky a střídají se předchozí dvě fáze. V opačném případě riskujeme uvíznutí výpočtu v nekonečné větvi.

Na následujících třech jednoduchých příkladech si demonstrujeme závislost výpočtu PROLOGu na pořadí pravidel a faktů. V příkladech budeme definovat binární relaci menší nebo rovno (zkráceně `mnr`) s využitím unární relace následník (zkráceně `nasl`). Například číslo dva lze chápat jako následníka následníka nuly a mělo by platit, že nula je menší

[1] Do podrobností a technických detailů se však pouštět nebudeme.

[2] Dá se dokázat, že `D` sémanticky vyplývá z `P`, právě když je z `P`, ¬`D` odvoditelný spor.

[3] My je zde nebudeme přesně definovat.

nebo rovna číslu dva. V našem zápisu bychom tento fakt vyjádřili jako `mnr(nula,nasl(nasl(nula)))`. A třeba to, že jedna je menší nebo rovna jedné, bychom zapsali jako fakt `mnr(nasl(nula),nasl(nula))`. Obecně, že číslo je menší nebo rovno samo sobě, můžeme s využitím proměnné `X` zapsat takto: `mnr(X,X)`. Tento fakt využíváme u všech tří verzí příkladu, kde na definici relace `mnr` poukazujeme na závislost výpočtu PROLOGu na pořadí pravidel a faktů.

Příklad 1

```
mnr(X,Y) :- mnr(nasl(X),Y).
mnr(X,X).
```

Pro dotaz `mnr(nula,nasl(nula))` začne PROLOG cyklit[4]:

```
mnr(nula,nasl(nula)),
mnr(nasl(nula),nasl(nula)),
mnr(nasl(nasl(nula)),nasl(nula)),
...
```

Příklad 2

```
mnr(X,X).
mnr(X,Y) :- mnr(nasl(X),Y).
```

Pro dotaz `mnr(nula,nasl(nula))` vrátí PROLOG odpověď `true`.

Příklad 3

```
mnr(X,X).
mnr(X,nasl(Y)) :- mnr(X,Y).
```

V tomto případě na pořadí pravidel nezáleží. Pro dotaz `mnr(nula,nasl(nula))` vrátí PROLOG v obou případech odpověď `true`.

Řezy

Viděli jsme, že programátor může ovlivňovat průběh PROLOGovského programu uspořádáním faktů a pravidel. Také záleží na pořadí relací v nich obsažených. Výpočet ovlivňuje i formulace dotazů. Dále se podíváme na další možnost, jak lze ovládat průběh výpočtu. Zaměříme se na tzv. „řez“, kterým lze zabraňovat backtrackingu.

Relace řezu je v PROLOGu reprezentována speciální vestavěnou relací „!“. Narazí-li PROLOG během výpočtu na řez, je okamžitě splněn a interpret pokračuje dalším cílem napravo od řezu. Řez ovšem zabraňuje navrácení k cílům nalevo od něj. Opětovné plnění řezu tedy skončí neúspěchem a neúspěchem končí i plnění všech cílů nalevo od řezu včetně hlavy pravidla. Tím dojde k odřezání jisté množiny výpočtů. Rozeznáváme dva typy řezů:

- zelený řez – řez, jehož aktivací neztratíme žádné řešení,
- červený řez – řez, který není zelený.

Zelený řez lze chápat jako nástroj sloužící k odstranění nedeterminismu programu. Při psaní pravidel v PROLOGu často dojdeme do situace, kdy se jednotlivá pravidla „vzájemně vylučují“. PROLOG se ovšem vždy pokouší o jejich splnění, i když se z hlediska nalezení řešení jedná o „mrtvé větve výpočtu“, kde žádné řešení nalezeno být nemůže. V takovém případě je vhodné využít řez, který překladači PROLOGu odřeže větve výpočtu, které nikam nevedou.

[4] Ve skutečnosti dojde k přetečení zásobníku kvůli nedostaku paměti, čímž se výpočet přeruší.

První příklad se zeleným řezem

Procedura (podprogram) pro nalezení většího ze dvou čísel může být naprogramována jako relace

```
maximum(X,Y,Max),
```

kde `Max=X`, jestliže `X` je větší nebo rovno `Y` a `Max=Y`, jestliže `X` je menší než `Y`. To odpovídá těmto dvěma PROLOGovským pravidlům:

```
maximum(X,Y,X) :- X>=Y.
maximum(X,Y,Y) :- X<Y.
```

Tato dvě pravidla se vzájemně vylučují. Uspěje-li první, druhé neuspěje. Pokud první neuspěje, musí uspět druhé. Je proto možná úspornější formulace: Jestliže `X`$\geq$`Y` pak `Max=X`, jinak `Max=Y`. To lze v PROLOGu s využitím (zeleného) řezu napsat takto:

```
maximum(X,Y,X) :- X>=Y,!.
maximum(X,Y,Y).
```

Poznamenejme, že použití téhle procedury je bezpečné, není-li konkretizována proměnná `Max` v relaci `maximum(X,Y,Max)`. Nesprávný výsledek obdržíme třeba při dotazu:

```
?- maximum(5,2,2).

true.
```

Tento problém vyřešíme následující reformulací relace `maximum`:

```
maximum(X,Y,Max) :-
  X>=Y,!,Max=X
  ;
  Max=Y.
```

První příklad s červeným řezem

Naprogramujeme proceduru `member1`, která určí pouze první výskyt daného prvku v seznamu (samozřejmě pokud se v něm vůbec vyskytuje). Stačí nám pozměnit binární relaci `member` z prvního dílu tohoto semináře a to jednoduše tak, že jakmile první výskyt prvku nastane, zabráníme backtrackingu, aby hledal další možnosti. Využijeme k tomu červený řez.

```
member1(X,[X|Ocas]) :- !.
member1(X,[Hlava|Ocas]) :- member1(X,Ocas).
```

Na následující dotaz nyní dostaneme jako odpověď jediné řešení:

```
?- member1(X,[a,b,c]).

X=a.
```

Jak setřídit seznam?

Nyní se budeme zabývat tříděním seznamů. Je jasné, že seznam může být seřazen, jestliže je dána nějaká relace uspořádání mezi položkami seznamu. My budeme dále předpo-

kládat, že takovou relaci máme k dispozici. Tuto relaci uspořádání pojmenujeme

```
vetsinez(X,Y)
```

s významem, že X je větší než Y, aniž bychom přesně specifikovali, co „větší než“ znamená. Budou-li položky v seznamu čísla, pak relaci vetsinez definujeme jako

```
vetsinez(X,Y) :- X>Y.
```

Pokud bychom chtěli řadit slova podle abecedy, použijeme lexikografické uspořádání definované jako:

```
vetsinez(X,Y) :- X@>Y.
```

Zavedeme nyní binární relaci

```
setrid(Seznam,Setridenyseznam),
```

kde Seznam je seznamem položek a Setridenyseznam je seznam se stejnými položkami uspořádanými vzestupně v souladu s relací vetsinez. Podíváme se na tři způsoby, jak definovat relaci setrid. Použijeme k tomu tři ideje pro třídění seznamu. První idea následuje.
K setřídění seznamu Seznam:

- Najdeme dva sousední prvky X a Y takové, že vetsinez(X,Y) a vyměníme je v seznamu Seznam, čímž obdržíme Seznam1. Pak setřídíme Seznam1.
- Jestliže se v seznamu Seznam nevyskytuje dvojice sousedních prvků X a Y taková, že vetsinez(X,Y), je Seznam setříděn.

Účelem výměny dvou prvků X a Y, které nejsou správně uspořádané je, že po výměně je nový seznam blíže setříděnému seznamu. Po dostatečném množství výměn dvojic prvků budou všechny prvky uspořádány. Tento třídící princip je znám pod názvem bublinkové třídění (anglicky bubble sort). Odpovídající PROLOGovskou proceduru budeme proto označovat bubblesort:

```
bubblesort(Seznam,Setridenyseznam) :-
  %Je treba vymena v Seznamu?
  vymena(Seznam,Seznam1),!,
  bubblesort(Seznam1,Setridenyseznam).

%jinak je seznam setriden
bubblesort(Setridenyseznam,Setridenyseznam).

%vymena prvnich dvou prvku
vymena([X,Y|Ocas],[Y,X|Ocas]) :-
  vetsinez(X,Y).

%vymena prvku v ocasu
vymena([Z|Ocas],[Z|Ocas1]) :-
  vymena(Ocas,Ocas1).
```

Jiným jednoduchým třídícím algoritmem je třídění vkládáním (anglicky insertion sort). Tento algoritmus je založen na následující myšlence.
K setřídění neprázdného seznamu S=[X|O]:

- Setřídíme ocas O seznamu S.
- Vložíme hlavu X seznamu S do setříděného ocasu O na takovou pozici, aby byl výsledný seznam setříděný. Výsledkem je celý setříděný seznam.

V PROLOGu tak dostáváme následující proceduru insertsort:

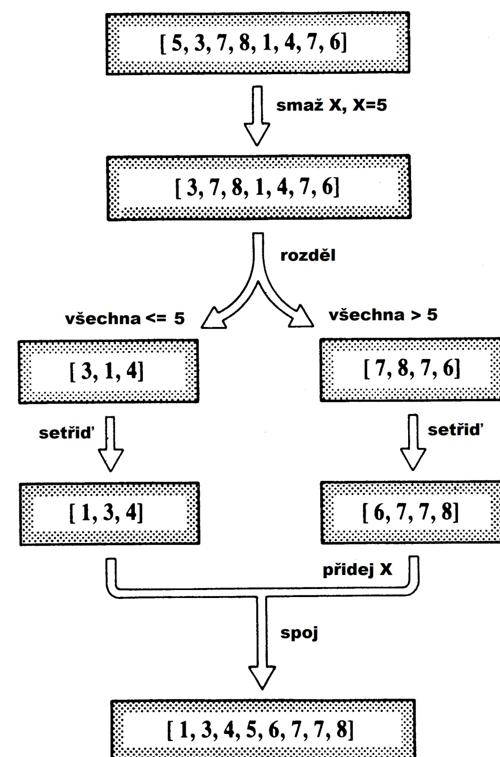
```
insertsort([],[]).

insertsort([X|Ocas],Setridenyseznam) :-
  %setrid ocas
  insertsort(Ocas,Setridenyocas),
  %Vloz X na spravne misto
  vloz(X,Setridenyocas,Setridenyseznam).

vloz(X,[Y|Setridenyseznam],[Y|Setridenyseznam1]) :-
  vetsinez(X,Y),!,
  vloz(X,Setridenyseznam,Setridenyseznam1).

vloz(X,Setridenyseznam,[X|Setridenyseznam]).
```

Třídící procedury bubblesort a insertsort jsou jednoduché, ale neefektivní. Z těchto dvou procedur je třídění vkládáním efektivnější. Průměrný čas, který insertsort potřebuje pro setřídění seznamu délky n roste úměrně k n^2. Pro delší seznamy je tak výhodnější použít rychlejší třídící algoritmus, například quicksort, který je založen na následující myšlence.



Obrázek 1: K principu třídění seznamu quicksortem.

K setřídění neprázdného seznamu S:

- Smažeme některý prvek X ze seznamu S a rozdělíme zbytek S na dva seznamy nazvané Male a Velke tak, aby všechny prvky v S větší než X patřily do seznamu Velke a všechny ostatní patřily do seznamu Male.
- Setřídíme Male, abychom obdrželi SetridenyMale.
- Setřídíme Velke, abychom obdrželi SetridenyVelke.
- Celý setříděný seznam vznikne spojením seznamu SetridenyMale se seznamem [X|SetridenyVelke].

Pokud je seznam prázdný, pak je výsledekem setřídění také prázdný seznam.[5] Prvek X, který v prvním

[5] Tento jednoduchý fakt slouží jako mezní podmínka rekurze.

kroku ze seznamu mažeme (tzv. pivot) lze brát například jako hlavu příslušného seznamu. Rozdělovací procedura tak může být realizována pomocí kvaternární relace `rozdel(X,S,Male,Velke)`.

Časová složitost quicksortu záleží zejména na volbě pivota. Pokud je seznam rozdělován na dva seznamy podobné délky, je časová složitost tohoto třídícího algoritmu řádově $n \log n$, kde n je délka seznamu, který má být setříděn. Naproti tomu, pokud pivot rozděluje seznamy tak, že jeden je mnohem delší než druhý je časová složitost řádově n^2. Naštěstí je známo, že v průměrném případě je časová složitost lineárně logaritmická.

O negaci

Zamysleme se, jak bychom mohli do PROLOGu formalizovat větu: „Marie má ráda všechna zvířata kromě hadů“. Je jednoduché převést první část tohoto tvrzení: Marie má ráda všechna `X`, když `X` je zvíře. V PROLOGu:

```
ma_rada(marie,X) :- zvire(X).
```

Máme ale vyloučit hady. To můžeme provést přeformulováním:

- Jestliže `X` je had, pak „Marie má ráda `X`“ není pravda, jinak, jestliže `X` je zvíře, pak „Marie má ráda `X`“.

To, že něco není pravda můžeme v PROLOGu vyjádřit pomocí speciální relace `fail`, která vždy neuspívá. Výše uvedená formulace lze do PROLOGu formalizovat takto:

```
ma_rada(marie,X) :-
  had(X),!,fail.

ma_rada(marie,X) :-
  zvire(X).
```

První pravidlo se týká hadů: jestliže je `X` had, pak řez zabrání backtrackingu (s výjimkou druhého pravidla); přičemž `fail` způsobuje selhání. Tato dvě pravidla mohou být kompaktněji zapsána takto:

```
ma_rada(marie,X) :-
  had(X),!,fail
  ;
  zvire(X).
```

Stejnou myšlenku můžeme použít pro definování binární relace `different(X,Y)`, která je pravdivá, jsou-li `X` a `Y` různé. Stačí nám formalizovat následující:

- Jestliže se `X` a `Y` shodují, pak `different(X,Y)` neuspěje, jinak `different(X,Y)` uspěje.

Opět použijeme kombinaci řezu a relace `fail`:

```
different(X,X) :- !,fail.
different(X,Y).
```

To můžeme přepsat do jednoho pravidla

```
different(X,Y) :-
  X=Y,!,fail
  ;
  true.
```

Speciální relace `true` vždy uspívá.

Tyto příklady ukazují, že by mohlo být užitečné mít k dispozici unární relaci `not` takovou, že

```
not(X)
```

je pravdivá, jestliže `X` není pravda. Nyní definujeme relaci `not`[6] následovně:

- Jestliže `X` uspěje, pak `not(X)` neuspěje, jinak `not(X)` uspěje.

V PROLOGu pak:

```
not(X) :-
  X,!,fail
  ;
  true.
```

Dva výše uvedené příklady můžeme s pomocí `not`[7] přepsat takto:

```
ma_rada(marie,X) :-
  zvire(X),
  not(had(X)).
```

```
different(X,Y) :-
  not(X=Y).
```

Tohle vypadá lépe než původní zápis, protože je to přirozenější a čitelnější.

Jako další příklad, kde použijeme relaci `not` uvedeme pozměněný první program pro řešení problému osmi dam z minulého dílu. Specifikujeme relaci `no_attack` mezi dámou a ostatními dámami. Tato relace může být formulována také jako negace relace `attack`. V PROLOGu pak:

```
solution([]).

%prvni dama na X/Y, ostatni damy na Others
solution([X/Y|Others]) :-
  solution(Others),
  member(Y,[1,2,3,4,5,6,7,8]),
  %prvni dama neohrozuje ostatni damy
  not(attacks(X/Y,Others)).

%dama na pozici X/Y ohrozuje nekterou z ostatnich dam
attacks(X/Y,Others) :-
  %dama na pozici X1/Y1 patri do seznamu Others
  member(X1/Y1,Others),
  (Y1=Y;
   Y1 is Y+X1-X;
   Y1 is Y-X1+X).
```

Program otestujeme dotazem:

```
?- solution([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```

Závěrem

V první části tohoto dílu jsme se blíže seznámili s průběhem výpočtu v PROLOGu. Poté jsme představili řezy, pomocí kterých jsme schopni vylepšit efektivitu programu.

[6] Negace, kterou zde definujeme nemá přestně stejný význam jako negace v matematiké logice.

[7] Již v prvním dílu jsme se s unární vestavěnou relací `not` setkali. Teď již víme, jakým způsobem ji lze zavést.

Můžeme totiž PROLOG donutit, aby nezkoušel jiné alternativy, protože nemají naději na úspěch (tedy ovlivnit výpočet PROLOGu, aby nechodil do větví, kde se nenaléza chtěné řešení). Pomocí řezů můžeme zavést užitečné podmínky, například (pro větvení programu) ve tvaru:

- *Jestliže* podmínka `P`, *pak* závěr `Q`,
  jinak závěr `R`.

Řezy tedy zvyšují expresivní sílu jazyka. Na druhou stranu použitím řezů může vzniknout nesoulad mezi tím, co se dá vyvodit z pohledu logického a z pohledu programátorského. Je proto žádoucí používat řezy opatrně a vždy s dobrým důvodem. Využití řezů jsme demonstrovali na vybraných třídících algoritmech.

Dále jsme využili řezy a vestavěnou relaci `fail` k definování unární relace `not`. Tato relace má význam negace, ale pouze v rámci daného programu, se kterým PROLOG pracuje. A to ve smyslu, že vše, co existuje, je součástí programu, nebo z něj vyplývá. Proto také `not` používáme opatrně a jen v odůvodněných případech.

V celém čtyřdílném semináři jsme se naučili základní principy a syntaxi programovacího jazyku PROLOG. Autor bude rád, když čtenáři budou nově nabyté vědomosti považovat za užitečné a někdy v budoucnu je využijí, případně si je rozšíří o oblasti, na které zde nezbyl čas.

Úkoly k odevzdávání

Následují úkoly, které se odevzdávají přes web semináře Olinx,

https://olinx.inf.upol.cz/

Úkoly stačí odevzdat ve formě jednoho jednoduchého textového souboru, ve kterém jsou napsány řešení ke všem úkolům. Pro tvorbu řešení se předpokládá používání aplikace SWI-PROLOG.

Úkol 1 *3 body*

Zjistěte a napište, co a k čemu jsou vestavěné ternární relace `bagof`, `setof` a `findall`.

Úkol 2 *8 bodů*

Naprogramujte v PROLOGu proceduru `slij` pro slití dvou setříděných seznamů do třetího seznamu tak, aby zůstalo zachováno uspořádání, tedy, aby i třetí seznam byl setříděný. Například:

```
?- slij([2,5,6,6,8],[1,3,5,9],S).

S=[1,2,3,5,5,6,6,8,9].
```

Úkol 3 *14 bodů*

Naprogramujte v PROLOGu rekurzivní třídící algoritmus quicksort.

Literatura

[1] I. Bratko, 2011. *PROLOG Programming for Artificial Intelligence (4th Edition)*. Pearson Education Canada. ISBN 9780321417466.