Programování v „čistém“ Prologu

Petr Štěpánek

S využitím materiálu Krysztofa R. Apta

2006

Ukázali jsme, že logické programy mohou sloužit k výpočtům. Volně řečeno, logiské programz mohou sloužit jako programovací jazyk.

To je jen první krok na dlouhé cestě ke skutečnému programovacímu jazyku. Kromě mnoha jiných problémů je třeba řešit otázku efektivnosti a uživatelského prostředí takového jazyka.

Prolog je programovací jazyka založený na myšlence logického programování, který oba zmíněné problémy řeší přijatelným způsobem.

V této kapitole se budeme zabývat programováním v podmnožině Prologu, která odpovídá logickým programům. Tuto podmnožinu budeme nazývat „Čistý Prolog“.

Každý logický program, pokud se na něj díváme jako na posloupnost klauzulí, tedy ne jako na množinu klauzulí (což je v logickém programování možné,) je také programem v čistém Prologu. Naopak to však neplatí.

Všechny logické programy, pokud se na ně díváme jako na posloupnost klauzulí a ne jako na množinu klauzulí, je program v čistém Prologu.

Zatím jsme uvedli jenom tři takové programy, *SUMMER, SUM, PATH* je možné spustit v každém současném Prologovském systému.

Čistý Prolog – syntaktické konvence

• Každá klauzule programu a každý dotaz je zakončen tečkou ‘ . ‘

• Abychom zdůraznili, že nejde o plný Prolog, místo prologovské implikace ‘ :- ‘ píšeme jako dosud šipku ‘ ← ‘ a u jednotkových klauzulí ji vynecháváme.

• Jednotkové klauzule nazýváme ***fakta*** a klauzule, které mají neprázdné tělo nazýváme ***pravidla*** .

Definice predikátu.

Je-li P daný program a p predikátový symbol vyskytující se v P, definicí tohoto predikátu rozumíme množinu všech klauzulí programu P, v jejichž hlavách se vyskytuje predikátový symbol p.

Termové konvence

• znakové řetezce začínající malým pismenem
například *‘ f ‘ , ‘ g ‘ , ‘ suma ‘ , ‘ summer ’ ‘ happy’*

jsou vyhrazeny pro jména predikátů a funkcí. (Tyto nenumerické konstanty se v kontextu Prologu nazývají *atomy* .)

• znakové řetězce začínající velkým písmenem nebo podtržítkem ‘ _ ‘
například *‘ X ‘ , ‘ Xs ’, ‘ _1796 ‘*

označují proměnné.

• řádky komentářů začínají vždy symbolem *%* .

Ambivalentní syntax

• Ačkoliv v predikátové logice (mlčky) předpokládáme, že množiny funkčních a predikátových symbolů jsou disjunktní a disjunktní jsou i množiny symbolů různé četnosti,

v Prologu můžeme použít stejné jméno pro funkční nebo predikátový symbol a dokonce současně v několika četnostech.

• funkční nebo predikátový symbol ‘ *f* ‘ četnosti *n* deklarujeme krátce jako *‘ f / n ‘* .

Příklad.

V kontextu Prologu můžeme tentýž symbol *p* použít jako predikátový symbol *p / 2* a funkční symbol *p / 1* a *p / 2* .

Potom se můžeme setkat se syntakticky správným faktem

. $p(p(a, b) , [c, p(a)])$.

Při používání ambivalentní syntaxe je třeba změnit Martelliho a Montanariho unfikační algoritmus následovně:

V akci 2 musíme připustit, že na obou stranách rovnosti jsou stejné funkční symboly. Formulujeme novou verzi akce 2 :

Akce 2′

$f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m)$ pokud $f \neq g$ nebo $n \neq m$ stop – neúspěch

V dalším výkladu budeme (v souladu s praxí) funkce četnosti nula nazývat *konstanty* a funkčními symboly budeme rozumět jenom symboly kladné četnosti.

Anonymní proměnné

Prolog umožňuje používání takzvaných ***anonymních proměnných*** , které se označují podtržítkem " _ " .

Jde o proměnné na jejichž hodnotách nám nezáleží (máme zájem jen na tom, že taková věc existuje). Takové proměnné se v klauzuli nebo v dotaze vyskytují zpravidla jenom jednou.

Proto každý výskyt anonymní proměnné v klauzuli nebo dotazu se interpretuje jako ***jiná proměnná*** .

Moderní verze Prologu, například SICStus Prolog , při syntaktické analýze programu identifikují solitérní proměnné a doporučují zaměnit je anonymními proměnnými.

Do čistého Prologu zařadíme obě syntaktické možnosti : ***ambivalentní syntax*** i ***anonymní proměnné*** .

Výpočty

Výpočetní proces Prologu se řídí následujícími pravidly:

(i) Výběrové pravidlo je pevně stanoveno a vybírá z každého dotazu nejlevější atom.

Z praktických důvodů v takovém případě budeme mluvit o LD-rezoluci místo o SLD-rezoluci,

Podobně budeme mluvit o LD-derivacích, LD-stromech atd.

(ii) Klauzule použitelné k vybranému atomu dotazu se zkouší v pořadí, v jakém jsou uvedeny v programu,

program je tedy posloupnost klauzulí.

Silná věta o úplnosti SLD-rezoluce říká, že (až na přejmenování proměnných) lze všechny vypočtené odpovědi k danému dotazu najít v SLD-stromu daného programu.

Nicméně výpočty některých logických programů mohou být beznadějně neefektivní i v případě, že se omezíme jen na výběrové pravidlo nejlevějšího atomu.

To znamená, že prohledávání LD-stromu (stavového prostoru LD-rezolvent) se stává vitálním aspektem z hlediska efektivnosti výpočtu.

Pokud bychom prohledávali LD-strom do šířky, tedy po hladinách, máme zaručeno, že pokud existuje, vypočtenou odpovědní instanci (vypočtenou odpovědní substituci) jistě najdeme.

Je zřejmé, že takové prohledávání může být exponenciálně složité vzhledem k výšce stromu a totéž platí o paměťových nárocích na ukládání navštívených uzlů.

Prohledávání do hloubky

Terminologie. *Uspořádaný strom* je strom, je zakořeněný a pro každý jeho uzel platí, že jeho bezprostřední následnící jsou lineárně uspořádané.

Prohledávání do hloubky se používá většinou jen u zakořeněných konečně se větvících stromů. Navíc se předpokládá, že každý list je označen jako úspěšný (success) nebo neúspěšný (fail).

Prohledávání do hloubky začíná v koření stromu a je charakterizováno tím, že z následníků již navštíveného uzlu je navštíven dříve než jeho sourozenci napravo od něj. Při takovém prohlednávání je každá hrana stromu navšívena nejvýše jednou.

Navštívíme-li úspěšný list stromu, tento fakt je avizován.

Navštívíme-li list označený jako neúspěšný, je vyvolán proces *navracení* (*backtracking*) . To znamená návrat k rodičovskému uzlu a pokračování od jeho dalšího následníka napravo, pokud existuje. V opačném případě se navracíme k rodičovskému uzlu o hladinu výš.

Prohledávání pokračuje až do okamžiku kdy se navrátí až ke kořenu stromu a všechny jeho následníci již byly navštíveny.

Jestliže prohledávání do hloubky vstoupí do nekonečné větve stromu dříve než byl navštíven úspěšný list, výsledkem je divergence tohoto procesu.

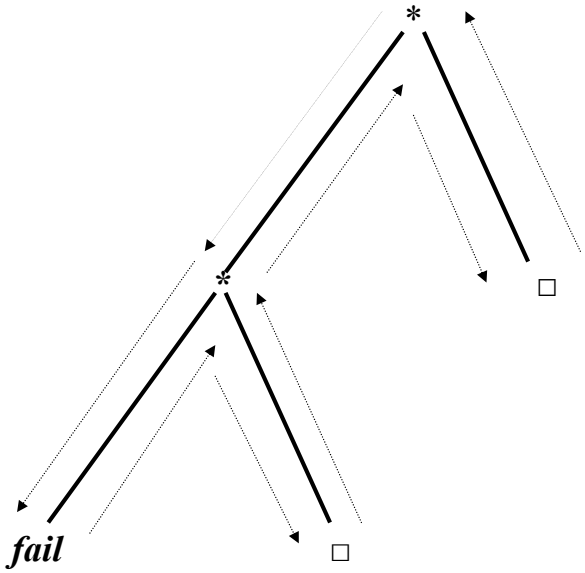
Prohledávání LD-stromů

V případě Prologu prohledávání do hloubky provádíme v LD-stromu odpovídajícímu danému programu a dotazu .

Je-li list LD-stromu označen jako úspěšný, tedy je-li ohodnocen prázdným dotazem, potom prohledávání končí a výstupem je odpovídající vypočtená odpovědní substituce.

Žádost o další řešení příkazem ‘ ; ‘ (středník) vede k obnovení prohledávání od posledního úspěšného uzlu dokud není nalezen nový úspěšný uzel. V případě, že daný LD-strom již neobsahuje žádný další úspěšný uzel, je neúspěch prohledávání avizován výstupem ‘ no ‘ .

Navracení v LD-stromu



Příklady.

a) Mějme následující program P_1 :

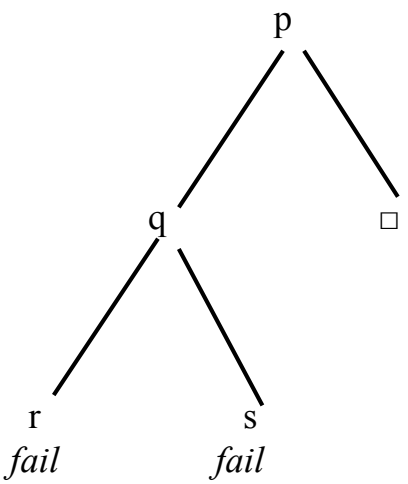
$p \leftarrow q$.

p .

$q \leftarrow r$.

$q \leftarrow s$.

a jeho LD-strom pro dotaz p .



.

LD-strom pro $P_1 \cup \{p\}$ je konečný se dvěma neúspěšnými a jedním úspěšným listem.

Konstrukce LD-stromů

nebo dokonce prologovských stromů je pouhá fikce. Vyhledávání odpovědí na dotazy v čistém Prologu je prohledávání do hloubky odpovídajícího LD-stromu. V Prologu jsou jistá omezení konstrukce prologovského stromu. Konstrukce prologovského stromu „*krok za krokem*“ je jen abstrakcí tohoto procesu. Pro naše účely však postačí.

Konstrukce prologovského stromu (*krok za krokem*) pro dotaz Q generuje posloupnost postupně vybíraných uzlů. Tyto uzly odpovídají odpovídají uzlům, které jsou postupně navštěvovány v průběhu prohledávání odpovídajícího LD-stromu s jediným rozdílem, navracení k rodičovskému uzlu je „neviditelné“.

Jak víme, pro každý dotaz Q a program P existuje právě jeden LD-strom pro $P \cup \{Q\}$. Je-li dán program P a dotaz Q , zavedeme následující terminologii:

• Říkáme, že *dotaz Q vždy zakončuje výpočet* (*universally terminates*) , je-li LD-strom pro $P \cup \{Q\}$ konečný. Například dotaz $path(X, c)$ generuje konečný (a uspěšný) LD-strom pro $PATH \cup \{path(X, c)\}$, který je zobrazen Obr. 1.

Podobně dotaz $path(c, X)$ vždy (univerzálně) zakončuje i když odpovídající strom je neúspěšný, (Obr. 1a).

• Q ***diverguje***, jestliže v LD-stromu existuje nekonečná větev vlevo od všech úspěšných uzlů. Například dotaz $path(X, Z)$ diverguje v pozměněném programu $PATH'$ (Obr. 2).

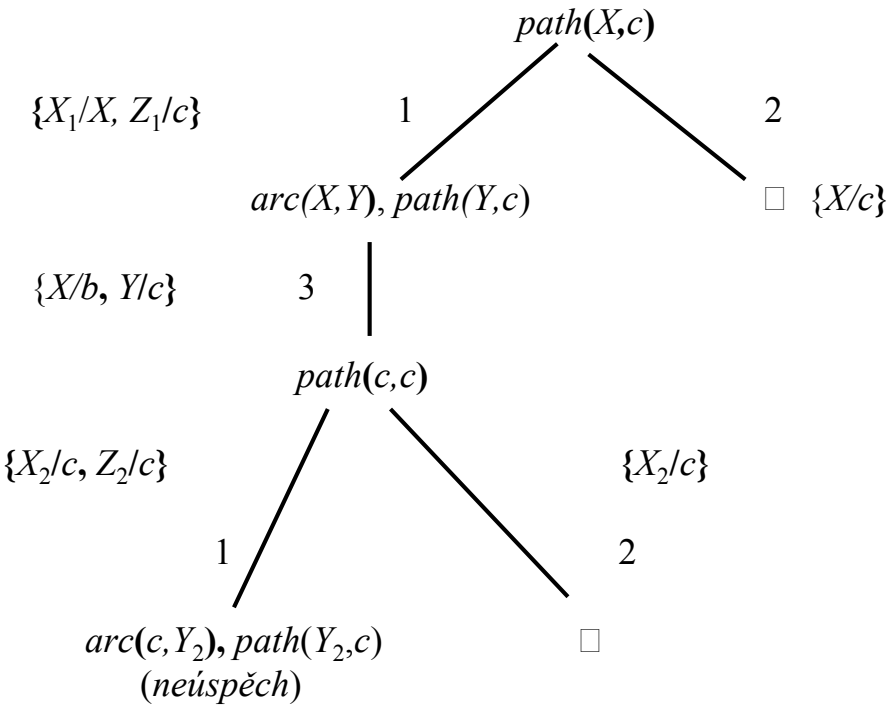
• Q ***potenciálně diverguje***, jestliže v LD-stromu pro $P \cup \{Q\}$ existuje úspěšný uzel, takový, že
- všechny větve nalevo od něj jsou konečné,
- napravo od něj existuje nekonečná větev (Obr. 3).

- *Q* ***generuje nekonečně mnoho odpovědí***, jestliže LD-strom pro $P \cup \{Q\}$ má nekonečně mnoho úspěšných uzlů a všechny nekonečné větve leží napravo od nich. (Obr. 4)
- *Q* ***selhává***, jestliže LD-strom pro $P \cup \{Q\}$ konečně selhává.

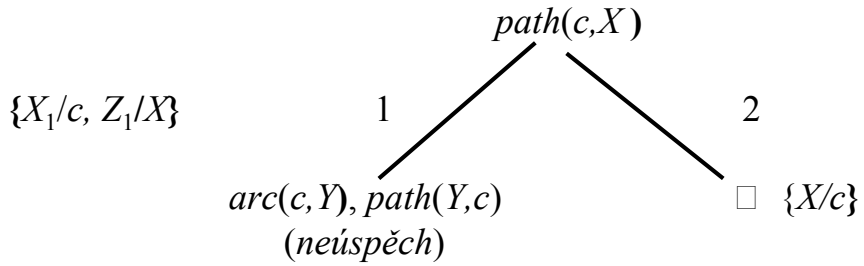
Příklad.

Uvažujme program *PATH*

1. *path(X, Z)* ← *arc*(*X, Y*), *path*(*Y, Z*).
2. *path*(*X, X*).
3. *arc*(*b, c*).



Obr. 1

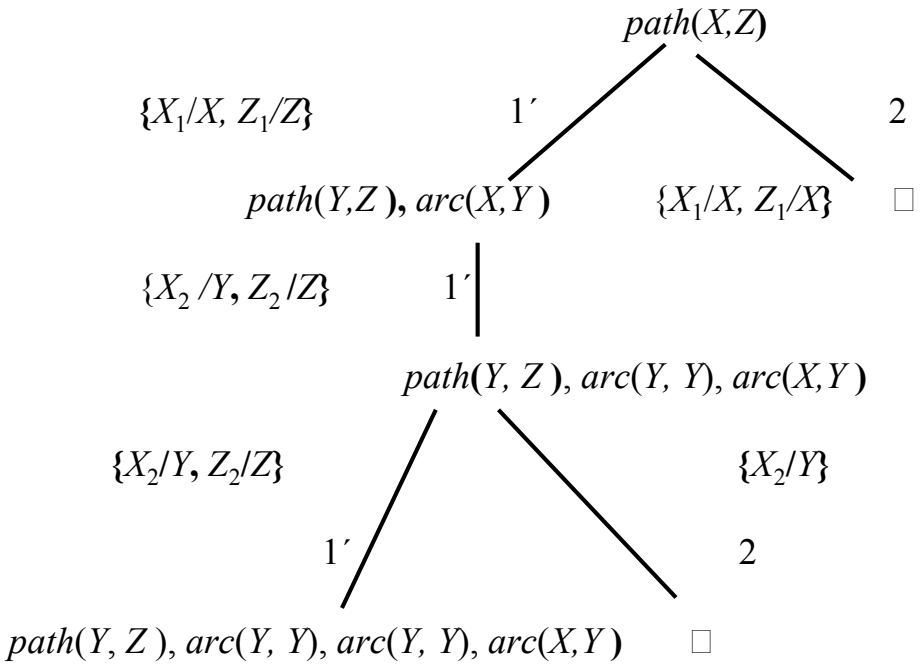


Obr. 1a

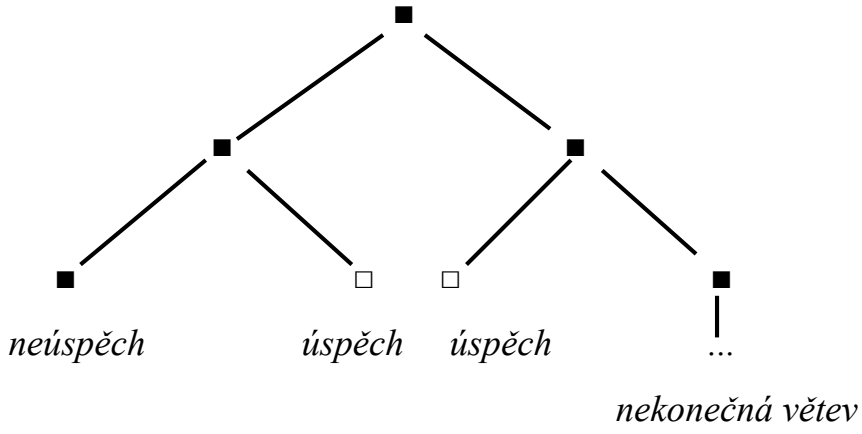
Změníme-li pořadí atomů v kaluzuli 1. dostaneme program PATH´

1. *path(X, Z)* ← *path*(*Y, Z*), *arc*(*X, Y*).
2. *path*(*X, X*).
3. *arc*(*b, c*).

V tomto programu dotaz *path*(*X, Z*) diverguje.

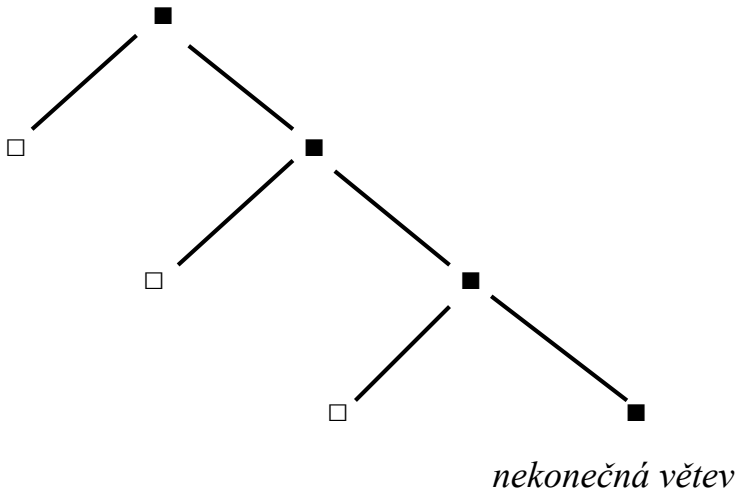


nekonečná větev Obr. 2



Obr. 3

Potenciální divergence



Dotaz produkující nekonečně mnoho odpovědí

Obr. 4