Logické programy
Procedurální interpretace

Petr Štěpánek

S využitím materialu Krysztofa R. Apta

2006

Logické programy počítají pomocí varianty resoluční metody, zvané SLD-resoluce. Unifikace tvoří základní ingredienci této resoluční metody, a proto jsme s unifikací výklad začali a podrobně jsme se s jejími vlastnostmi seznámili.

Nyní se můžeme - náležitě vyzbrojeni - věnovat hlavnímu tématu této přednášky: logickým programům.

Procedurální interpretace vysvětluje *jak* logické programy počítají. Detailní znalost této interpretace je nutná k pochopení základů logického programování a k výkladu výpočetního mechanismu Prologu.

I když se Prolog liší od Logického programování, pro hlubší pochopení základů Prologu je výhodné, zavedeme-li nejprve výpočetní mechanismus logických programů a pak jen vysvětlíme rozdíly.

Tento způsob výkladu má i své výhody

• vysvětluje určité volby při návrhu systému (například volba způsobu prohledávání prostoru resolvent a vynechání testu výskytu proměnných (occur-check) v unifikačním algoritmu) a

• osvětluje nebezpečí, která z přijatých voleb vyplývají (možnost divergentních výpočtů nebo ´occur-check problem´).

Nejprve rozšíříme jazyk termů, se kterým jsme dosud pracovali, do jazyka programů.

Pak budeme moci definovat programy a dotazy (queries) a zavést SLD-derivace, které představují výpočty odpovědí na dotazy položené programu.

Dotazy a programy

Do jazyka termů nejprve přidáme

- *predikátové symboly* označováné *p, q, r, ... ,*
- *obrácenou implikaci* $\leftarrow$ a *čárku* , označující konjunkci.

Počet predikátových symbolů stejně jako funkčních se v jednotlivých jazycích může lišit. Každý predikátový symbol má svou četnost (*aritu*) tedy počet argumentů. Připouštíme i četnost 0 a takový predikátový symbol nazýváme *výrokovou konstantou*.

Definice. (Atomy, dotazy, klauzule a programy)

- je-li p n-ární predikátový symbol a $t_1, t_2, \ldots, t_n$ jsou termy, výraz $p\,(t_1, t_2, \ldots, t_n)$ nazýváme *atomem* (je to atomická formule).

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

V matematické logice se místo $H \leftarrow \mathbf{B}$ píše $\mathbf{B} \rightarrow H$.

Použití obrácené šipky má kromě zdůraznění jiného formalismu také stravitelné vysvětlení : je-li $H \equiv p\,(t_1, t_2\,, \dots, t_n)$, klauzule $H \leftarrow \mathbf{B}$ se považuje jako součást deklarace predikátu p .

Pro další analýzu SLD-derivací se zavádí pojem *rezultanty* ve tvaru $\mathbf{A} \leftarrow \mathbf{B}$, kde **A** a **B** jsou dotazy.

Značení.

- atomy označujeme $A, B, C, H, \ldots,$
- dotazy Q , **A, B, C,** ... ,
- klauzule $c, d, \ldots,$
- rezultanty $R, R', R_1, \ldots,$
- programy $P, P', P_1, \ldots$.

Prázdný dotaz se označuje □ a je-li **B** prázdný dotaz, místo

$$H \leftarrow \mathbf{B} \quad \text{píšeme krátce} \quad H \leftarrow$$

a výraz nazýváme *jednotková klauzule*.

Používání velkých písmen zde není na závadu, v Prologu se proměnné označují zpravidla velkými písmeny z konce abecedy.

Zato politováníhodným rozdílem mezi terminologií Logických programů a Prologu je používání slova ´atom´ . V logickém programování se používá ve významu ´atomické formule´ a v Prologu znamená nenumerickou konstantu.

V našem výkladu budeme důsledně slovo ´atom´ používat jen ve významu ´atomické formule´.

Pro čtenáře obeznámené s predikátovou logikou prvního řádu bude užitečné, připomeneme-li to, co jsme o vztahu jazyka logických programů a jazyka predikátové logiky řekli hned na začátku.

Dotaz $Q \equiv A_1, A_2, \ldots, A_n$ interpretujeme jako formuli

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

kde $x_1, x_2, \ldots, x_k$ jsou všechny proměnné, které se vyskytují v atomech $A_1, A_2, \ldots, A_n$.

Z výpočtového hlediska dotaz Q interpretujeme jako požadavek nalézt hodnoty proměnných $x_1, x_2, \ldots, x_k$ tak, aby konjunkce $A_1 \,\&\, A_2 \,\&\, \ldots \,\&\, A_n$ byla pravdivá.

Prázdný dotaz $\square$ chápeme jako prázdnou konjunkci, která je vždy pravdivá, tomuto symbolu přiřazujeme hodnotu *true* .

Podobně klauzuli $c \equiv H \leftarrow B_1, B_2, \ldots, B_n$ interpretujeme jako implikaci

$$\forall x_1 \forall x_2 \ldots \forall x_k \, ((B_1 \,\&\, B_2 \,\&\, \ldots \,\&\, B_n) \rightarrow H)$$

kde $x_1, x_2, \ldots, x_k$ jsou proměnné vyskytující se v klausuli c.

Z výpočtového hlediska klauzuli c interpretujeme jako tvrzení " k důkazu H je třeba dokázat $B_1, B_2, \ldots, B_n$ ". Přitom záleží na pořadí v jakém mají být dokázány atomy B_i .

Ukázali jsme, že všechny proměnné v dotazech, klauzulích i programech jsou vázané.

Můžeme je tedy přejmenovávat a substituovat za ně podle potřeby.

Jazyk programu.

Kdykoli uvažujme nějaký program P předpokládáme, že jeho termy a atomy jsou napsány v nějakém jazyce logiky prvního řádu.

Nabízí se přirozená první volba "jazyk definovaný programem P", který obsahuje jen ty funkční a predikátové symboly, které se vyskytují v programu P. Je to minimální jazyk, ve kterém lze program, jeho výpočty a další vlastnosti analyzovat.

Na druhé straně nic nebrání tomu, abychom za jazyk programu P vzali jakékoli rozšíření jazyka definovaného programem. V takovém rozšíření se mohou vyskytovat nové funkční a predikátové symboly, které mohou být použity v dotazech kladených programu.

V dalším výkladu uvidíme, že tato volba je mnohem přirozenější, budeme-li studovat vztah logikya programů v Prologu.

SLD - derivace.

Intuitivně, program P je množina klauzulí a dotaz Q je požadavek nalézt instanci dotazu $Q\theta$, která by vyplývala z P.

Úspěšný výpočet vydá takovou substituci θ a můžeme jej chápat jako důkaz instance $Q\theta$ z množiny klauzulí P .

Logické programy počítají kombinací dvou mechanismů : nahrazení a unifikace. Abychom porozumněli lépe detailům tohoto výpočetního procesu, budeme se nejprve soustředit na nahrazení bez přítomnosti proměnných.

Předpokládejme na chvíli, že ani program ani dotaz neobsahují proměnné.

Uvažujme nějaký program P a neprázdný dotaz $Q \equiv \mathbf{A},B,\mathbf{C}$ a klauzuli $B \leftarrow \mathbf{B}$ z programu P. Dotaz $\mathbf{A},\mathbf{B},\mathbf{C}$ je výsledkem nahrazení naznačeného výskytu atomu B v $\mathbf{A}, B, \mathbf{C}$ tělem $\mathbf{B}$ použité klauzule. Dotaz $\mathbf{A},\mathbf{B},\mathbf{C}$ nazveme *rezolventou* $\mathbf{A},B,\mathbf{C}$ a $B \leftarrow \mathbf{B}$.

B se nazývá *vybraný atom* z $\mathbf{A},B,\mathbf{C}$. Píšeme $\mathbf{A},B,\mathbf{C} \Longrightarrow \mathbf{A},\mathbf{B},\mathbf{C}$.

Iterováním tohoto procesu nahrazování získáme posloupnost rezolvent, která se nazývá *derivace*. Derivace mohou být konečné nebo nekonečné.

Je-li poslední dotaz prázdný, mluvíme o *úspěšné derivaci* počátečního dotazu Q. Můžeme také říci, že jsme dokázali dotaz Q z programu P.

Je-li poslední dotaz v derivaci neprázdný a pro vybraný atom B není v programu P klausule s hlavou B mluvíme o *neúspěšné derivaci*.

Příklad. Uvažujme program *SUMMER* , který sestává z klauzulí

happy $\leftarrow$ *summer, warm.*
warm $\leftarrow$ *sunny.*
warm $\leftarrow$ *summer.*
summer $\leftarrow$.

Zde *summer*, *warm*, *sunny* a *happy* jsou predikátové symboly četnosti 0 , tedy výrokové symboly (výrokové konstanty).

Potom posloupnost

happy ==> $\underline{summer}$ *, warm* ==> *warm* ==> *summer* ==> $\square$

je úspěšná derivace dotazu *happy* a

happy ==> *summer*, $\underline{warm}$ ==> *summer*, $\underline{sunny}$

je neúspěšná derivace dotazu *happy* .

Je-li možnost výběru, vybraný atom v rezolventách je podtržen.

Předchozí příklad ukazuje, že logické programy mohou být použity k důkazům, ale umí také počítat.

Když rozšíříme předchozí úvahu o možnost pracovat s programy a dotazy, které mohou obsahovat proměnné, můžeme proces výpočtu vysvětlit.

Zde budeme používat oba základní kroky rezoluční metody, tedy nahrazení i unifikaci.

Nejprve je nutné rozšířit unifikaci z termů i na atomy. Toho dosáhneme okamžitě, budeme-li predikátové symboly chápat jako funkční symboly. Syntaktický tvar atomů a termů je pak stejný a k unifikaci můžeme použít kterýkoli unifikační algoritmus.

Nyní můžeme zavést obecný pojem rezolventy, kterému se říká SLD-rezolventa .

Definice. (SLD-rezolventa)

Mějme neprázdný dotaz $Q \equiv \mathbf{A}, B, \mathbf{C}$ a klazuli c. Nechť $H \leftarrow \mathbf{B}$ je varianta klauzule c, která je disjunktní v proměnných s dotazem Q. Předpokládejme, že atomy B a H lze unifikovat. Nechť θ je nějaká mgu B a H. B nazýváme vybraný atom dotazu Q.

Pak píšeme

$$\mathbf{A}, \mathrm{B}, \mathbf{C} \;=\!\theta/c\!\Rightarrow\; (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$$

a říkáme tomu *SLD-derivační krok*. $H \leftarrow \mathbf{B}$ se nazývá *vstupní klauzule*.

Pokud na klauzuli c nezáleží, můžeme ji vynechat a psát

$$\mathbf{A}, B, \mathbf{C} \;=\!\theta\!\Rightarrow\; (\mathbf{A}, \mathbf{B}, \mathbf{C})\theta$$

Povšimněme si, že SLD-resolventa byla sestrojena pomocí specifické varianty klauzule c místo klauzule c samé. Díky tomu definice SLD-rezolventy nezávisí na náhodné volbě proměnných v klauzuli c.

Následující příklad osvětluje tento bod definice.

Příklad. Mějme dotaz $Q \equiv p(x)$ a klauzuli $c \equiv p(f(y)) \leftarrow$. Potom je prázdný dotaz $\square$ rezolventou Q a c stejně jako v případě, kdy $c' \equiv p(f(x)) \leftarrow$. I když v tomto případě nelze atomy $p(x)$ a $p(f(x))$ unifikovat.

SLD-derivační krok můžeme také vyjádřit jako odvozovací pravidlo

$$\frac{\mathbf{A}, B, \mathbf{C} \qquad H \leftarrow \mathbf{B}}{(\mathbf{A}, \mathbf{B}, \mathbf{C})\theta}$$

.

kde $\mathbf{A}, B, \mathbf{C}$ a $H \leftarrow \mathbf{B}$ jsou disjunktní v proměnných, θ je mgu B a H a $H \leftarrow \mathbf{B}$ je variantou klauzule c.

Tato definice rezolventy je zobecněním výrokového případu, kde byla použita jen operace nahrazení.

Iterováním SLD-derivačních kroků dostaneme SLD-derivaci . Abychom dostali nejobecnější odpovědi na počáteční dotaz, je třeba dát jistá syntaktická omezení na použité mgu a vstupní klauzule.

Definice. (SLD-derivace)

Maximální posloupnost

.
$$Q_0 \overset{\theta_1 / c_1}{\Longrightarrow} Q_1 \ldots Q_n \overset{\theta_{n+1} / c_{n+1}}{\Longrightarrow} Q_{n+1} \ldots \qquad (1)$$

SLD-derivačních kroků nazveme *SLD - derivací* $P \cup \{Q_0\}$ jestliže

- $Q_0, \ldots, Q_{n+1}$, ... jsou dotazy buď prázdné nebo s vybraným atomem
- $\theta_1, \ldots, \theta_{n+1}$, ... jsou substituce
- $c_1, \ldots, c_{n+1}$, ... jsou klauzule z programu P

a pro každý rezoluční krok platí následující podmínky:

• Standardizace proměnných úkrokem stranou (apart): použitá vstupní klauzule je disjunktní v proměnných s počátečním dotazem Q_0 a se všemi substitucemi a vstupními klauzulemi použitými na každém z dosavadních kroků. Formálně:

$$Var(c_i') \cap \left[Var(Q_0) \cup \bigcup_{j=1}^{i-1} (Var(\theta_j) \cup Var(c_j') \right] = 0$$

pro $i \geq 1$, kde c_i' je vstupní klauzule použitá v rezolučním kroku

. $Q_{i-1} = \theta_i / c_i \Rightarrow Q_i$

Intuitivně je zřejmé, že stačí, aby v každém kroku SLD-derivace byly všechny proměnné nové vstupní kluzule "nové". Pokud budeme takové výpočty předvádět ručně, budeme nové proměnné indexovat pořadím kroku.

Je-li program P známý z kontextu, mluvíme o *SLD*-derivaci $\{Q_0\}$, pokud nám nezáleží na klauzulích $c_1 , \ldots , c_{n+1} , \ldots$ z daného programu můžeme odkaz na ně také vynechat.

SLD-derivaci (1) potom můžeme zapsat

$$Q_0 = \theta_1 \Rightarrow Q_! \ldots Q_n = \theta_n \Rightarrow Q_{n+1} \ldots$$

Je zřejmé, že derivace, které jsme použili v příkladu programu *SUMMER ,* bez proměnných jsou speciálním případem SLD-derivací. Používá se v nich jenom nahrazení ne však unifikace.

SLD je zkratka z anglického názvu *Selection rule driven Linear resolution for Definite clauses.*

Definice. (atom a k němu použitelné klauzule)

(i) Říkáme, že klauzule *je použitelná* k nějakému atomu, jestliže nějaká varianta hlavy této klauzule se unfikuje s daným atomem.

(ii) Délka konečné SLD-derivace je je počet jejích derivačních kroků.

Speciálně, SLD-derivace délky 0 sestává z jediného dotazu Q, který je buďto prázdný nebo žádná klauzule z daného programu není použitená k vybranému atomu dotazu Q.

Nekonečným SLD-derivacím délku nepřiřazujeme.

Konečné SLD-derivace mají pro nás zvláštní význam, i když při podrobné analýze chování programu je nutné analyzovat i derivace nekonečné.

Definice. (Vypočtená odpovědní substituce a vypočtená instance)

Uvažujme program P a konečnou SLD-derivaci

$$\xi \equiv Q_0 = \theta_1 \Rightarrow Q_1 \ldots Q_{n-1} = \theta_n \Rightarrow Q_n$$

dotazu $Q \equiv Q_0$. (přesněji $P \cup \{Q\}$)

(i) Říkáme, že ξ je úspěšná SLD-derivace jestliže $Q_n = \square$.

Omezení $(\theta_1\theta_2 \ldots \theta_n) \,|Var(Q)$ složené substituce $\theta = \theta_1\theta_2 \ldots \theta_n$ na proměnné dotazu Q, říkáme *vypočtená odpovědní substituce* dotazu Q a jeho instanci $Q\theta$ říkáme *vypočtená instance.*

(ii) SLD-derivace ξ je *neúspěšná* jestliže Q_n je neprázdný dotaz a žádná klauzule programu P není použitelná k vybranému atomu z Q.

Poznámka. Vypočtená odpovědní substituce θ vznikne složením nejobecnějších unifikací θ_i v pořadí v jakém byly použity v úspěšné SLD-derivaci a omezením složené substituce jen na proměnné dotazu Q .

Substituci θ chápeme jako výsledek výpočtu dotazu Q a vypočtenou odpovědní instanci $Q\theta$ chápeme jako tvrzení dokázané derivací ξ .

Vypočtená odpovědní substituce dává hodnoty proměnným dotazu Q pro které je Q pravdivým tvrzením. Podrobněji se budeme otázkám pravdivosti zabývat v souvislosti se sémantikou logických programů.

Povšimněme si, že definice neúspěšné derivace předpokládá, že výběr (nějakého výskytu) atomu v dotazu je prvním krokem ve výpočtu rezolventy. Totéž platí i pro úspěšné derivace pokud mají nenulovou délku.

Příklad. (Počítání s numerály)

Předpokládejme, že jazyk obsahuje konstantu 0 ("nula") a unární funkční symbol s ("funkce následníka").
Termům $0 , s(0), s(s(0)) , \ldots , s^n(0) , \ldots$ budeme říkat *numerály* a budeme je chápat jako representanty přirozených čísel

$$0, 1, 2, \ldots , n , \ldots$$

.
Následující program *SUMA* počítá součet dvou numerálů:

(1) $suma(x,0, x) \leftarrow .$

(2) $suma(x,s(y), s(z)) \leftarrow suma(x,y,z) .$

V SLD-derivacích budeme vstupní klauzule v i-tém kroku generovat z klauzulí programu tím, že proměnné již použité ve výpočtu budeme indexovat číslem i . Tím splníme podmínku standardizace proměnných.

Nejprve zkusíme spočítat kolik je dvě a dvě. Získáme úspěšnou derivaci dotazu $suma(s(s(0)), s(s(0)), z)$:

$suma(s(s(0)), s(s(0)), z) = \theta_1/2 \Rightarrow suma(s(s(0)), s(0), z_1) = \theta_2/2 \Rightarrow$
$suma(s(s(0)), 0, z_2) = \theta_3/1 \Rightarrow \square$

kde

$\theta_1 = \{x/s(s(0)), y/s(0), z/s(z_1)\}$
$\theta_2 = \{x_2/s(s(0)), y_2/0, z_1/s(z_2)\}$
$\theta_3 = \{x_3/s(s(0)), z_2/s(s(0))\}$

Tomu odpovídá vypočtená odpovědní substituce

$$\theta_1\theta_2\theta_3\,|\{z\} = \{z/s(z_1)\}\,\{z_1/s(z_2)\}\,\{z_2/s(s(0))\}\,|\{z\} = \{z/s(s(s(s(0))))\}$$

Můžeme volně říci, že jsme nalezli hodnotu proměnné z pro kterou dotaz $suma(s(s(0)), s(s(0)), z)$ platí. Jmenovitě $z = s^4(0)$. Přitom SLD-derivace pro z generovala postupně hodnoty $s(z_1), s^2(z_2)$ a $s^4(0)$.

Povšimněme si, že v tomto případě tvar dotazu v každém kroku určuje jenom jednu použitelnou klauzuli.

Položíme-li dotaz *suma*(x,y,z) a v každém kroku použijeme klauzu-li (2) dostaneme nekonečnou SLD-derivaci

$$suma(x,y,z) = \theta_1/2\Rightarrow suma(x_1,y_!,z_!) = \theta_2/2\Rightarrow suma(x_2,y_2,z_2) \ \ldots$$

kde

$$\theta_1 = \{x/x_1, y/s(y_!), z/s(z_!)\}$$

$$\theta_2 = \{x_1/x_2, y_!/s(y_2), z_!/s(z_2)\}$$

...

Při řešení dotazu *suma*(x,y,z) jsou v každém kroku použitelné obě klauzule z programu *SUMA*. Kdybychom ve třetím rezolučním kroku

Při řešení dotazu *suma*(x,y,z) jsou v každém kroku použitelné obě klauzule z programu *SUMA*. Kdybychom ve třetím rezolučním kroku použili klauzuli (1) místo klauzule (2) , dostaneme úspěšnou SLD-derivaci a substituci

.
$$\theta_3 = \{x_2/x_3,\ y_2/0,\ z_2/x_3\}$$

Složením všech tří substitucí dostaneme odpovědní substituci

.
$$\theta = \theta_1\theta_2\theta_3 \,|\{x,y,z\} = \{x/x_3,\ y/\mathrm{s}(\mathrm{s}(0))\ ,\ z_3/s(s(x_3)))\}$$

a vypočtenou odpovědní instanci *suma*(x_3, , $s^2(0)$, $s^2(x_3)$) , kterou můžeme volně interpretovat jako $x + s^2(0) = s^2(x))$.

Srovnání s některými styly programování.

I. Imperativní programování. (“destruktivní přiřazení”.)

V imperativních jazycích proměnné nabývají svých hodnot přiřazovacími příkazy tvaru $x := t$ (1)

Nabízí se možnost ztotožnit provedení přiřazovacího příkazu (1) se substitucí $\{x/t\}$. Zdánlivě je to ve shodě s axiomem

$$\{\varphi\{x/t\}\ x := t\ \{\varphi\}$$

Hoarovy logiky imperativních programů, který popisuje efekt přiřazovacího příkazu (1) v termínech tzv. “před-podmínky” a “po-důsledku”.

Tímto způsobem bychom mohli adekvátně vyjádřit efekt některých posloupností přiřazovacích příkazů.

Například posloupnost příkazů $x := 3$; y := z; by odpovídala složení substitucí $\{x/3\}\{y/z\}$, tedy substituci $\{x/3\ ,\ y/z\}$.

Tento postup však selhává při popisu jiných posloupností, jako posloupnost $x := 3; x := x + 1;$. Po provedení prvního příkazu v této rovině popisu vyjádřeného substitucí $\{x/3\}$ proměnná x „zmizí“ a není možné žádnou substitucí její hodnotu zvýšit o 1 .

Jinými slovy, první přiřazovací příkaz je *destruktivní* a jeho efekt nelze adekvátně popsat pomocí substituce.

Jak je tento problém řešen v logickém programování ? Hodnota proměnné x je vyčíslena prostřednictvím posloupnosti určitých substitucí

.

$$\theta_1 , \theta_2 , \ldots , \theta_n ,$$

které pro $1 \leq i \leq n$ postupně generují mezivýsledky $x\theta_1\theta_2 \ldots \theta_i$ hodnot proměnné x.

Posloupnost postupně generovaných hodnot $x\theta_1\theta_2 \ldots \theta_i$ chápeme jako "monotonní" posloupnost (vzhledem k počtu instancí) termů.

Výsledek je potom instance všech postupně generovaných hodnot.

Můžeme říci, že hodnoty postupně generované částečné hodnoty všech proměnných ("sklad") tvoří vzhledem k relaci "býti obecnější než" monotonní posloupnost substitucí.

Dobře to ilustrují úspěšné výpočty programu *SUMA* .

Naproti tomu hodnoty proměnných při výpočtu imperativních programů nemusí vykazovat žádnou pravidelnost.

Tím se výpočty logických programů podstatně liší od výpočtů imperativních programů. V logickém programování nenajdeme (mimo jiné) obdoby destruktivního přiřazení nebo zvýšení hodnoty proměnné.

II. Funkcionální programování - jiný styl deklarativního programování.

Funkcionální obdobu logického programu *SUMA* budeme modelovat jako systém přepisující termy. Tato metoda patří k základům většiny funkcionálních jazyků.

Funkcionální obdobou programu *SUMA* bude následující program $SUMA1$. Tento program používá stejnou množinu termů, ale predikát $suma(x, y, z)$ je nahrazen funkcí $suma1(x, y)$ dvou argumentů.

Program $SUMA1$ modeluje výpočet logického programu *SUMA* pomocí dvou přepisovacích pravidel:

1. $suma1(x,0) \rightarrow x$
2. $suma1(x, s(y)) \rightarrow s(suma1(x,y))$

Následující výpočet programu $SUMA1$ se v terminologii termy přepisujících systémů nazývá *redukční posloupnost* . Pro vstupní term $suma1(s(s(0)), s(s(0)))$ dává v jistém smyslu také monotonní posloupnost a “stejný” výsledek $s(s(s(s(0))))$.

$$suma1(s(s(0)), s(s(0))) \rightarrow s(suma1(s(s(0)), s(0)) \rightarrow$$
$$\rightarrow \mathrm{s(s(suma1(}s(s(0)), 0))) \rightarrow s(s(s(s(0))))$$

.

Na rozdíl od logického programu $SUMA$ funkcionální program $SUMA1$ nemá schopnost “invertibility”.

Pro odečítání je třeba napsat jiný funkcionální program. $SUMA1$ “neumí” modelovat výpočet logického programu $SUMA$ s dotazem $suma(z, s(0),s(s(s(0)))$ nebo složitěji $suma(s(x), y,s(s(0)))$.

Tento rozdíl se považuje za bezvýznamný.