UNIVERZITA KOMENSKÉHO V BRATISLAVE
Fakulta matematiky, fyziky a informatiky

Magické transformácie pre DATALÓG$^{\neg}$

Diplomová práca

Študijný odbor: Informatika

Vedúci diplomovej práce:
RNDr. Ján Šturc, CSc.

Diplomant:
Štefan Zajíček

Bratislava 2008

Čestné prehlásenie

Vyhlasujem, že som diplomovú prácu vypracoval samostatne s použitím uvedenej odbornej literatúry.

Bratislava 06. 05. 2008 .....

Vlastnoručný podpis

Poďakovanie

Touto cestou ďakujem vedúcemu svojej diplomovej práce RNDr. Jánovi Šturcovi, CSc. za cenné odborné rady a pripomienky pri jej vypracovávaní, Dr. Tomášovi Plachetkovi za pomoc pri ladení programu a podnetné pripomienky k implementácii a svojej rodine a priateľom za podporu.

Názov práce: Magické transformácie pre DATALÓG$^{\neg}$

Pracovisko: Katedra informatiky, FMFI UK v Bratislave
Autor: Štefan Zajíček
Vedúci DP: RNDr. Ján Šturc, CSc.
Dátum: 06. 05. 2008
Kľúčové slová: Datalóg, negácia, evaluácia, magické transformácie, vyhodnotenie, zdola-nahor, implementácia.

Abstrakt: Datalóg je logický dotazovací jazyk pre deduktívne databázy. Disponuje výrazovými prostriedkami, ktoré umožňujú jednoduchým a intuitívnym spôsobom písať dotazy, ktoré vo viacerých komerčne používaných databázových jazykoch (napr. v niektorých implementáciách SQL), nie sú dovolené. Príkladom môže byť použitie rekurzie.

V tejto práci popíšeme návrh a implementáciu aplikácie, slúžiacej na vyhodnocovanie datalógových programov obohatených o negáciu, pre ktorú budeme uvažovať well-founded sémantiku. Použijeme metódy zdola-nahor, optimalizované magickými transformáciami. Magická transformácia je optimalizačná technika, ktorá simuluje výpočet zhora-nadol pri výpočte zdola-nahor a tým výrazne zmenšuje priestor potrebný pre vyhodnocovanie.

Obsah

Úvod

- Myšlienka použiť jazyk logiky pri implementácii databázového systému, tzv. *deduktívnej databázy* nie je nová. Má niekoľko výhod. [7]

  - V logických (deklaratívnych) termoch je možné presne a výstižne vyjadriť viaceré databázové koncepty, akými sú: dotazy, pohľady, integritné ohraničenia, ako aj samotné dáta. To umožňuje vytvoriť jednotné rozhranie pre používateľov.
  - Deduktívne databázy majú väčšiu výrazovú silu ako väčšina relačných databáz. V relačných termoch možno prirodzeným spôsobom reprezentovať relácie, ktoré nie sú v *prvej normálnej forme*[2], a umožňujú tiež definíciu rekurzívnych pohľadov.
  - Pre aplikácie, ktoré nie je vhodné implementovať v logickom prostredí, nie je prístup do deduktívnej databázy z jazykov ako napr. C, C++ o nič komplikovanejší, ako prístup do relačných databáz.

  Datalóg je dotazovací jazyk pre deduktívne databázy. Syntakticky je podmnožinou Prologu. V základnej verzii neumožňuje použitie negácie, agregácie a funkčných symbolov. Je však snaha tieto obmedzenia efektívnym spôsobom prekonať. Na rozdiel od Prologu, datalóg rozlišuje medzi programom a dátami. Použité predikáty sa teda delia na *intenzionálne* (odvodené) a *extenzionálne* (reprezentujúce relácie externej databázy).

  Pri výpočte datalógového programu obvykle uvažujeme metódu *zdola-nahor*, ktorá je vhodnou alternatívou k prehľadávaniu do hĺbky, implementovanému v Prologu.

  Metóda *zdola-nahor* je efektívna hlavne pre programy, ktoré majú pomerne viac faktov, ako pravidiel. [7] Jej výhodou je napríklad schopnosť vyhodnotiť programy bez nutnosti odstraňovať ľavú rekurziu, ktorá napr. v Prologu spôsobuje zacyklenie výpočtu.

Nevýhodou je, že za istých okolností môže vyžadovať omnoho väčší priestor pre výpočet, ako metódy *zhora-nadol* (podrobnejšie v časti 1.2). Túto neefektivitu odstraňujú magické transformácie (podrobnejšie v časti 1.2.3).

Ak povolíme v datalógu negáciu, nemôžeme pri výpočte priamočiaro použiť magickú transformáciu, nakoľko môže zmeniť význam programu. Tomuto je možné predchádzať rôznymi obmedzeniami kladenými na vyhodnocované programy, alebo voľbou vhodnej metódy na ich vyhodnotenie. Podrobnejšie v časti 1.2.5.

Ďalším možným rozšírením je dovoliť, aby relácie mohli obsahovať aj n-tice s premennými, prípadne aj s neuzavretými termami.

- Pri štúdiu databáz sa často stretávame s potrebou vyhodnocovať programy písané v datalógu. Tu môže byť užitočná aplikácia, ktorá by to realizovala. Pri komplikovanejších programoch je to priam nevyhnutnosť. Takáto aplikácia by mala spĺňať nasledovné požiadavky:
  - Musí byť jednoduchá a ľahko použiteľná.
  - Mala by implementovať čo najviac syntaktických rozšírení datalógu.
  - Bolo by vhodné, aby dokázala zobrazovať priebeh výpočtu. A to nielen kvôli demonštrácii princípu techniky výpočtu, ale aj ako pomoc pri ladení programu.
  - Výhodou by bolo, keby umožňovala zdielať dáta s bežnou relačnou (SQL) databázou. Tým by sa zjednodušilo porovnávanie dotazov písaných v rôznych jazykoch bez nutnosti konvertovať dáta.

  Existujúcim systémom (diskutovaným v časti 3.1) chýba jedna, alebo viacero horeuvedených vlastností. Niektoré poskytujú komplexnú funkcionalitu (XSB, Glue-Nail), avšak na úkor jednoduchosti a transparentnosti. Autorovi sa nepodarilo nájsť v popise funkcionality žiadneho z nich možnosť priamočiareho zobrazenia priebehu výpočtu.

- Cieľom tejto práce je navrhnúť a implementovať aplikáciu, ktorá by slúžila ako datalógové rozhranie pre SQL databázu. Dôraz budeme klásť na jednoduchosť implementácie a použiteľnosť. Pri písaní dotazov užívateľ nebude musieť explicitne rozlíšiť medzi extenzionálnymi a intenzionálnymi predikátmi.

  Budeme uvažovať rozšírenie syntaxe o negáciu a v reláciách povolíme aj premenné. Výpočet bude prebiehať *zdola-nahor* a implementujeme

aj optimalizáciu pomocou magických transformácií pri zachovaní well-founded sémantiky. Priebeh výpočtu budeme zobrazovať na výstupe.

Kapitola 1

Teoretické základy

1.1 Základné pojmy

1.1.1 Pojmy relačnej algebry

Definície v tejto časti sú prevzaté z [4] a [5].

Definícia 1 (Projekcia)**.** Nech relácia R je typu $X \cup Y$.
Potom $\Pi_X(R) = \{X : (\exists Y) r(X, Y)\}$

Definícia 2 (Kartézsky súčin)**.** Nech relácia R_1 je typu X a R_2 je typu Y.
Potom $R_1 \times R_2 = \{[X, Y] : r_1(X) \wedge r_2(Y)\}$

Definícia 3 (Join)**.** Nech relácia R_1 je typu XZ a R_2 je typu YZ, kde Z sú spoločné atribúty.
Potom $R_1 \bowtie R_2 = \{[X, Y, Z] : r_1(X, Z) \wedge r_2(Y, Z)\}$

Definícia 4 (Anti-join)**.** Nech R a S su relácie. Potom anti-join R a S (značíme $R \triangleleft\!\triangleright S$) bude relácia:

$R \triangleleft\!\triangleright S = R \setminus \Pi_R(R \bowtie S)$

1.1.2 Pojmy logického programovania

Nasledujúce definície v tejto časti sú prevzaté z [5] a [7].

Definícia 5 (Term)**.**

1. Konštanty a premenné sú termy.
2. Nech $t_0, \ldots, t_{n-1}$ sú termy a f je funkčný symbol, potom aj $f(t_0, \ldots, t_{n-1})$ je term.

Definícia 6 (Atóm)**.** Ak p je n-árny predikát a $t_0, \ldots, t_{n-1}$ sú termy, tak $p(t_0, \ldots, t_{n-1})$ je *atóm*.

Definícia 7 (Literál)**.** Ak q je atóm, tak q aj $\neg q$ sú *literály* - q je pozitívny literál a $\neg q$ je negatívny literál.

Definícia 8 (Pravidlo)**.** Ak q je atóm, a $p_0, \ldots, p_n$ sú literály, tak
$q \leftarrow p_0, \ldots, p_n$
je pravidlo.

Definícia 9. Termy, literály a pravidlá sú *uzavreté*, ak neobsahujú žiadne premenné.

Definícia 10. *Program* je množina pravidiel.

Definícia 11. *Herbrandovo univerzum* HU_L jazyka L je množina všetkých uzavretých termov, ktoré možno zostrojiť pomocou konštánt a funkčných symbolov z L. (pridáme konštantu, ak L žiadne neobsahuje). *Herbrandova báza* HB_L jazyka L je množina všetkých uzavretých atómov, ktoré možno zostrojiť pomocou predikátov z L s uzavretými termami z HU_L. Pod označením Herbrandova báza a Herbrandovo univerzum programu P (HU_P a HB_P) budeme rozumieť Herbrandovu bázu a Herbrandovo univerzum jazyka zloženého z konštánt, funkčných symbolov a predikátov nachádzajúcich sa v P.

Definícia 12 (Substitúcia)**.** Substitúciou nazývame funkciu $\sigma : V \to T$, kde V je množina premenných a T je množina termov.
Substitúcie zapisujeme: $\sigma = [\{x_i \mapsto t_i\}_{i=1}^{n}]$
Aplikácia substitúcie: Substitúciu σ na term t aplikujeme tak, že výsledok je term $s = t\sigma$, ktorý vznikne nahradením premenných vyskytujúcich sa v t aj σ príslušnými termami. Nahradenie sa vykoná naraz pre všetky premenné.

Definícia 13. Hovoríme, že substitúcia σ je zložením substitúcií ρ a τ, píšeme $\sigma = \rho \circ \tau$, ak pre každý term t platí $t\sigma = (t\tau)\rho$.

Definícia 14. Hovoríme, že substitúcia σ je aspoň tak všeobecná ako substitúcia τ, píšeme $\sigma \sqsupseteq \tau$, ak existuje substitúcia ρ taká, že $\sigma = \rho \circ \tau$.

Definícia 15 (Unifikácia)**.** Unifikácou nazveme riešenie úlohy: Daná je dvojica termov t a s. Nájdite najvšeobecnejšiu substitúciu σ takú, že $t\sigma = s\sigma$. Substitúciu σ nazveme najvšeobecnejším unifikátorom.

Definícia 16 (Slabá unifikácia)**.** Slabou unifikácou nazveme riešenie úlohy: Daná je dvojica termov t a s. Nájdite dvojicu najvšeobecnejších substitúcií ι, σ takých, že $t\iota\sigma = s\sigma$.

Existuje viacero algoritmov, ktoré hľadajú najvšeobecnejší unifikátor. Algoritmus 1 zobrazuje herbrandovu metódu. Metóda predpokladá, že je daná množina E rovníc medzi termami. Substiúcia σ je na začiatku prázdna.

1.1.3 Datalóg

Datalóg je dotazovací jazyk pre deduktívne databázy využívajúci ideu logického programovania.

Relácie v datalógu sú reprezentované *predikátmi*. Každý predikát má pevne daný počet *argumentov* a predikát spolu s argumentami tvorí *atóm*. Syntax atómu je rovnaká, ako volanie funkcie v bežných programovacích jazykoch (C, pascal). Napríklad $p(X_1, \ldots, X_n)$ je atóm skladajúci sa z predikátu p a argumentov $X_1, \ldots, X_n$.

while $E \neq \emptyset$ **do**
  vyber rovnicu $e \in E$;
  $E \leftarrow E - \{e\}$;
  if e je tvaru $x = t$ *or* e je tvaru $t = x$ **then**
    if x sa nachádza v t **then**
      return(riešenie neexistuje);
    end
    else
      $s \leftarrow S \circ [x \mapsto t]$;
      $E \leftarrow E[x \mapsto t]$;
    end
  end
  else if e je tvaru $f(s_0, ..., s_{n-1}) = f(t_0, ..., t_{n-1})$ **then**
    $E \leftarrow E \cup \{s_0 = t_0, \ldots, s_{n-1} = t_{n-1}\}$;
  end
  else
    return(riešenie neexistuje);
end

Algoritmus 1: Herbrandova metóda

$anc(X, Y) \leftarrow par(X, Y).$

$anc(X, Y) \leftarrow par(X, Z), anc(Z, Y).$

$nocyc(X, Y) \leftarrow anc(X, Y), \neg anc(Y, X).$

Obr. 1.1: Stratifikovaný program

Predikát je v podstate meno funkcie, ktorá vracia booleovskú hodnotu. Ak r je relácia, s n atribútmi v nejakom pevne danom poradí, tak môžeme r použiť aj ako meno predikátu, ktorý tejto relácii zodpovedá. Atóm $r(a_1, \ldots, a_n)$ má hodnotu *true*, ak n-tica $a_1, \ldots, a_n$ patrí do r. V opačnom prípade má hodnotu *false* [2].

Datalógový program tvorí množina *pravidiel* (*Hornových klauzúl*).
Pravidlá sú tvaru:
$p(\tilde{x}) \leftarrow q_1(\tilde{y}_1), \ldots, q_n(\tilde{y}_n).$

Vektory $\tilde{x}, \tilde{y}_1, \ldots, \tilde{y}_n$ tvoria *argumenty* jednotlivých literálov. Sú tvorené premennými a konštantami, teda sú to *termy* bez funkčných symbolov.

Atóm $p(\tilde{x})$ sa nazýva *hlava* pravidla. Výraz $q_1(\tilde{y}_1), \ldots, q_n(\tilde{y}_n)$ nazývame *telo* pravidla. Jednotlivé literály tvoriace telo nazývame *podciele*.

1.1.4 Sémantika

Pre programy bez negácie obvykle uvažujeme sémantiku najmenšieho pevného bodu [6]. Ak povolíme negáciu, stále môžme tieto metódy používať pre niektoré triedy programov. Takými sú napríklad stratifikované programy [7]. Intuitívne: Sú to programy, ktoré neobsahujú cyklus s negáciou.

Príklad 1. *Na obrázku 1.1 je príklad stratifikovaného programu. Intuitívne, tento program je stratifikovaný, pretože definícia predikátu* nocyc *závisí (negatívne) na definícii* anc*, ale definícia* anc *nezávisí na definícii* nocyc.*[?]*

Nestratifikované programy však nemusia mať najmenší pevný bod a ich výpočet sa môže zacykliť.

Príklad 2. *Programy na obrázku 1.2 nie sú stratifikované. Predikáty* p *a* q *tvoria cyklus s negáciou. Podobne aj predikát* win *[7]*

(a) $p(a) \leftarrow q(a), \neg r(a).$

$q(a) \leftarrow \neg p(a).$

$r(a).$

$? - q(a).$

(b) $win(X) \leftarrow move(X, Y), \neg win(Y).$

Obr. 1.2: Nestratifikované programy

Je teda zrejmé, že sémantika najmenšieho pevného bodu vo všeobecnosti nie je postačujúca pre programy s negáciou. Inuitívny význam programu na obrázku 1.2(b) je taký, že X je vyhrané postavenie, práve vtedy, keď nejaký ťah z X vedie k nevyhranému postaveniu. Ak *move* reprezentuje acyklický graf, tak každé postavenie je buď vyhrané alebo nevyhrané. Inými slovami: má dvojhodnotový intuitívny model. Napríklad, ak relácia pre *move* je $\{move(a, b),\ move(b, c),\ move(c, d),\ move(d, e)\}$ tak intuitívny model bude $\{\neg win(e),\ win(d),\ \neg win(c),\ win(b),\ \neg win(a)\}$.

Viaceré návrhy sémantík programov s negáciou sa zhodujú s týmto intuitívnym modelom. Najzaužívanejšie sú sémantika stabílnych modelov [9] a well-founded sémantika [11]. Akonáhle však *move* obsahuje cyklus, tieto dva prístupy sa prestávajú zhodovať vo význame programu. Napríklad pridanie ťahu $move(c, a)$ ponechá program bez stabílneho modelu. Avšak, keďže well-founded sémantika je (jednoznačná) trojhodnotová sémantika, stále dokáže priradiť pravdivostné hodnoty tým postaveniam, ktoré nie sú nijakým spôsobom ovplyvnené cyklom a ostatné označí ako nedefinované. S pridanou hranou *move(c,a)* budú vo well-founded modeli $win(a)$, $win(b)$ a $win(c)$ nedefinované, ale $\neg win(e)$ a $win(d)$ budú stále platiť. Well-founded sémantika sa teda javí ako vhodná voľba sémantiky pre negáciu v deduktívnych databázach. [7]

$p(X, Y) \leftarrow e(X, Y).$

$p(X, Y) \leftarrow e(X, Z), p(Z, Y).$

Dotaz je

(a) *?- p(X, Y).*

(b) *?- p(v1, v2).*

Obr. 1.3: Výpočet ciest v orientovanom grafe

1.2 Spôsoby vyhodnocovania programov v datalógu

V [3] je popísaných viacero algoritmov pre výpočet programov zhora-nadol, aj zdola-nahor. V tejto časti porovnáme ich výhody a nevýhody.

Uvažujme program pre výpočet ciest v grafe na obr. 1.3. V prvom prípade si obidve metódy poradia viac-menej rovnako dobre. Vyrátajú celý tranzitívny uzáver grafu. V druhom prípade je však metóda zdola-nahor veľmi neefektívna. Napriek tomu, že by stačilo uvažovať len cesty, ktoré začínajú vo *v1* a končia vo *v2*, opäť vyráta celý tranzitívny uzáver. Teda zbytočne spracováva množstvo faktov, ktoré neskôr aj tak „zahodí".

Metóda zhora-nadol spracováva len tie fakty, ktoré naozaj aj využije. Pri druhom dotaze sa naviaže konštanta *v1* na premennú *X* a *v2* na *Y*. Pri rekurzívnom volaní využitím druhého pravidla sa zavolajú podciele *e(v1,Z)*, *p(Z,v2)*.

Teda pre každú hranu vychádzajúcu z *v1* sa zisťuje, či z jej druhého konca nevedie cesta do *v2*. Je teda zrejmé, že pri takomto vnáraní sa, nebudeme nikdy uvažovať napr. hrany nachádzajúce sa v inom komponente grafu ako *v1* a *v2*.

Uvažujme teraz program obsahujúci ľavú rekurziu alebo cyklus. Metóda zdola-nahor nemá problémy, ale metóda zhora-nadol sa môže zacykliť. Jedna z metód, ktorá spája výhody metód zdola-nahor aj zhora-nadol sú magické transformácie.

1.2.1 Naívna evaluácia

Naívnou evaluáciou vypočítavame najmenší pevný bod datalógového programu. (Táto časť je prevzatá z [5]).

Term-matching

Daný je term t a vzor term s. Úlohou je nájsť substitúciu ρ takú, že $t\rho = s$. Riešenie: Začneme prázdnou všade nedefinovanou substitúciou ρ a aplikujeme rekurzívny algoritmus 2. Ak procedúra vráti *true*, v ρ bude hľadaná substitúcia. Ak vráti *false*, riešenie neexistuje.

Konverzia argumentov na premenné

Nech P je relácia pre podcieľ $p(t_1, \ldots, t_k)$. Nech $X_1, \ldots, X_n$ sú premenné vyskytujúce sa v tomto podcieli. Definujeme reláciu Q so schémou $X_1, \ldots, X_n$ tak, ako ukazuje algoritmus 3.

Po transformácii podcieľov operáciou *atov* môžeme výrazy vyhodnotiť operáciami relačnej algebry. Výsledok musíme konvertovať späť na argumenty.

Konverzia premenných na argumenty

Nech $R(X_1, \ldots, X_n)$ je výsledok výpočtu výrazu E pre telo bezpečného pravidla. Nech $s(t_1, \ldots, t_m)$ je predikát pre hlavu pravidla obsahujúci premenné $X_1, \ldots, X_n$. Definujeme reláciu S pre hlavu pravidla tak, ako zobrazuje algoritmus 4.

Výpočet joinov a antijoinov pre n-tice s funkčnými symbolmi - uzavreté termy

Pôvodne sa n-tice spájali na základe rovnosti spoločných atribútov. Teraz sa budú spájať na základe unifikovateľnosti spoločných atribútov.

Dané sú dve n-tice:

$b =< s_1, s_2, \ldots, s_k, t_{k+1}, t_{k+2}, \ldots, t_n >$
$b =< u_1, u_2, \ldots, u_{n-k}, v_{n-k+1}, v_{n-k+2}, \ldots, v_m >,$

```
procedure match(u,v):boolean
begin
    if u is variable then
        if ρ(u) is undefined then
            ρ(u) ← v;
            return(true);
        end
        else if ρ(u) = v then
            return(true);
        end
        else
            return(false)
        end
    end
    else if u = f(u_0, ..., u_{k-1}) and v = f(v_0, ..., v_{k-1}) then
        i ← 0;
        while i < k and match(u_i, v_i) do
            i ← i + 1;
        end
        if i = k then
            return(true);
        end
        else
            return(false);
        end
    end
    else
        return(false)
    end
end
```

Algoritmus 2: match(u,v)

$Q \leftarrow \emptyset$;

for *each tuple* $< s_1, \ldots, s_k > \in P$ **do**

  if $match(p(t_1, \ldots, t_k), p(s_1, \ldots, s_k))$ **then**

    $Q \leftarrow Q \cup < \rho(X_1), \ldots, \rho(X_n) >$;

  end

end

Algoritmus 3: Konverzia argumentov na premenné: $Q = atov(p, P)$

$S \leftarrow \emptyset$;

for *each tuple* μ **do**

  $\sigma \leftarrow \emptyset$;

  for $i \leftarrow 1$ *to* n **do**

    $\sigma \leftarrow \sigma \cup [X_i \mapsto \mu[X_i]]$;

    $S \leftarrow S \cup < t_1, \ldots, t_m > \sigma$;

  end

end

Algoritmus 4: Konverzia premenných na argumenty $S = vtoa(s, R)$

treba vypočítať join d a antijoin e n-tíc b a c podľa „$t = u$“. N-tice obsahujú len uzavreté termy. Výpočet zobrazuje algoritmus 5

if $< t_{k+1}, t_{k+2}, \ldots, t_n > \neq < u_1, u_2, \ldots, u_{n-k} >$ **then**
| $d \leftarrow \emptyset$

else
| $d \leftarrow \{< s_1, s_2, \ldots, s_k, t_{k+1}, t_{k+2}, \ldots, t_n >\}$

end

if $< t_{k+1}, t_{k+2}, \ldots, t_n > = < u_1, u_2, \ldots, u_{n-k} >$ **then**
| $e \leftarrow \emptyset$

else
| $e \leftarrow \{< s_1, s_2, \ldots, s_k, t_{k+1}, t_{k+2}, \ldots, t_n >\}$

end

Algoritmus 5: Výpočet joinu a antijoinu

Poznámka 1.2.1. *Dve n-tice sa rovnajú, ak sa rovnajú všetky ich zložky. Dva uzavreté termy sa rovnajú, ak sú identické.*

Výpočet joinov a antijoinov pre n-tice s funkčnými symbolmi a premennými

Predpokladajme, že n-tice b a c obsahujú termy s premennými. Ďalej predpokladáme, že mená premenných sú lokálne v n-ticiach. T.j. premenná X v rámci jednej n-tice znamená to isté. Medzi n-ticami na mene premennej nezáleží. Vo výpočte sa rovnosť rieši slabou unifikáciou.

Idea výpočtu:

1. Vypočítame dve najvšeobecnejšie substitúcie σ_1 a σ_2, pre ktoré platí: $< t_{k+1}, t_{k+2}, \ldots, t_n > \sigma_1 = < u_1, u_2, \ldots, u_{n-k} > \sigma_2$
2. Vypočítame $b' = b\sigma_1$ a $c' = c\sigma_2$
3. Vypočítame join d ako klasický join b' a c'.
4. Antijoin $e = b - b\sigma_1$.

Vo všeobecnosti sa antijoin nedá vyjadriť lepším spôsobom.

Výpočet pravidla

Nech $r{:}p \leftarrow q_1, \ldots, q_k$ je bezpečné pravidlo a nech $R_1, \ldots, R_k$ sú relácie pre podciele $q_1, \ldots, q_k$, potom reláciu R_0 pre hlavu p vypočítame nasledovne:

1. Pre každý podcieľ q_i vypočítame reláciu $Q_i = atov(q_i, R_i)$.
2. Vypočítame $P = Q_1 \bowtie \ldots \bowtie Q_k$ reláciu pre telo.
3. Výsledok $R_0 = vtoa(p, P)$.

Výpočet programu bez negácie

- Predošlý algoritmus rieši jediné pravidlo. Ak máme viac pravidiel s tou istou hlavou, vypočítame každé zvlášť a výsledok zjednotíme.
- Ak graf závislosti predikátov je acyklický, jeho topologickým utriedením dostaneme poradie výpočtu pravidiel (podľa predikátu v hlave).
- Ak graf závislosti predikátov nie je acyklický, rozdelíme predikáty na intenzionálne a extenzionálne (dané). Všetkým intenzionálnym predikátom priradíme prázdnu množinu a iterujeme podľa Tarského vety o pevnom bode [1]. (T.j. v každej iterácii vypočítame všetky nové intenzionálne predikáty za pomoci starých).

Výpočet programu s negáciou

- Všetko robíme rovnako ako bez negácie, jedine v kroku 2 pre výpočet pravidla v prípade nenegovaného predikátu generujeme *join* a v prípade negovaného predikátu *antijoin*.
- Poradie negovaných a nenegovaných predikátov je dôležité, antijoin sa nesmie použiť, pokiaľ v pozitívnej časti nie sú inštanciované spoločné premenné.

1.2.2 Zdvojený program

Definícia 17 (Zdvojený program)**.** Pre daný program P je zdvojený program *D(P)* definovaný nasledovne:

Najskôr nahradíme každý idb atóm *p(s̃)*, ktorý sa nachádza v pôvodnom programe v negatívnom literále atómom *p'(s̃)*, pričom *p'* sa nenachádza v *P*.

$p(X) \leftarrow t(X,Y,Z), \neg p(Y), \neg p(Z).$

$p(b) \leftarrow \neg r(a).$

$t(a,a,b).$

$t(a,b,a).$

Obr. 1.4: Program P

Potom zdvojíme celý program a v kópii všade zameníme idb atómy *p(š̃)* za *p'(š̃)* a naopak.

Well-founded model programu P môzme vyrátať použitím dvoch polovíc zdvojeného programu. Prvá polovica (pravidlá definujúce nečiarkované predikáty) vyráta pravdivé fakty, zatial čo druhá polovica (pravidla definujúce čiarkované predikáty) vyráta komplement množiny nepravdivých faktov. Každá polovica programu je pozitívna, ak považujeme negované predikáty za pevne dané. Môžeme teda vyrátať pevný bod každej polovice programu pomocou ľubovolnej techniky zdola-nahor pre programy bez negácie.

Príklad 3. *Uvažujme program* P, *na obrázkoch 1.4 a 1.5, ktorý nie je modulárne stratifikovaný, ale má dvojhodnotový well-founded model.*

Dvojhodnotový well-founded model je vypočítaný pomocou nasledovnej procedúry. Proceúra inkrementálne vypočíta pravdivé *fakty pomocou nečiarkovaných predikátov a* nie nepravdivé *(alebo možno pravdivé) fakty pomocou čiarkovaných predikátov nasledovne.*

1. *Predpokladajúc, že všetky čiarkované fakty sú pravdivé v nečiarkovanom programe, najskôr vypočítame model nečiarkovaných predikátov ako* $T_1 = \{t(a,a,b), t(a,b,a)\}$.

2. *Keď použijeme toto ako bázu pre definíciu* $p(), r(), t()$ *v čiarkovanom programe, môžme vypočítať model čiarkovaných predikátov ako*

$p(X) \leftarrow t(X,Y,Z), \neg p'(Y), \neg p'(Z).$

$p(b) \leftarrow \neg r'(a).$

$t(a,a,b).$

$t(a,b,a).$

$p'(X) \leftarrow t'(X,Y,Z), \neg p(Y), \neg p(Z).$

$p'(b) \leftarrow \neg r(a).$

$t'(a,a,b).$

$t'(a,b,a).$

Obr. 1.5: Zdvojený program *D(P)*

$U_2 = \{t'(a,a,b), t'(a,b,a), p'(a), p'(b)\}.$

3. *Ak opakovane vypočítame model nečiarkovaných predikátov, dostaneme*

   $T_3 = \{t(a,a,b), t(a,b,a), p(b)\}.$

4. *Ak potom vypočítame model čiarkovaných predikátov, dostaneme*

   $U_3 = \{t'(a,a,b), t'(a,b,a), p'(b)\}$, *čím sme dosiahli pevný bod.*

Korektnosť metódy vyplýva z pozorovania, že vypočítavame *pravdivé* (nečiarkované) fakty pomocou už vypočítaných *nepravdivých* faktov (komplementu čiarkovaných predikátov). Čiže naše odvodenia sú korektné. Podobne, keďže vypočítavame *nie nepravdivé* (čiarkované) fakty úplne, čiže vypočítame každý fakt ktorý môže byť pravdivý (nemôže byť nepravdivý) pre dané pravdivé nečiarkované fakty, už vieme, že fakty v komplemente tejto množiny sú nepravdivé. Táto metóda je implementáciou Van Gelderovej sémantiky alternujúceho pevného bodu [10].

Našim cielom pri výpočte well-founded modelu programu je vypočítať dve množiny T a F, ktoré mu zodpovedajú. Zdvojený program $D(P)$ vždy vyrátava negatívnu informáciu komplementárne, to znamená, že vypočíta komplement nepravdivých faktov. Čiže nikdy nebudeme musieť explicitne použiť množinu nepravdivých faktov F a môžeme manipulovať s negatívnymi faktami pomocou ich komplementu $HB_P - F$, kde HB_P je Herbrandova báza P. Algoritmus techniky zdvojeného programu inkrementálne počíta pravdivé fakty, striedavo s výpočtom komplementu nepravdivých, v postupnosti T_1, $U_2 = (HB_P - F_2)$, T_3, $U_4 = (HB_P - F_4)$,...

Nech E sú extenzionálne (EDB) predikáty a I sú intenzionálne (IDB) predikáty. Ak máme danú trojhodnotovú interpretáciu $E^+ \cup \neg \cdot (HB_E - E^-)$ EDB predikátov, vypočítame trojhodnotovú interpretáciu $I^+ \cup \neg \cdot (HB_I - I^-)$ IDB predikátov p_j, $1 \leq j \leq m$ definovaných týmto programom použitím algoritmu 6.

$Rules^0 \leftarrow$ pravidlá v P bez IDB literálov v tele;

$Rules^- \leftarrow$ pravidlá v P s iba negatívnymi literálmi v tele;

$Rules^+ \leftarrow$ pravidlá v P s iba pozitívnymi literálmi v tele ;

$Rules^\pm \leftarrow$ pravidlá v $P - Rules^- - Rules^+ - Rules^0$;

$E^+, HB_E - E^- \leftarrow$ *pravdivé* a *nepravdivé* n-tice v čiastočnom

well-founded modeli EDB predikátov ;

%% Vypočítaj $I^+ = T_1$;

$I^+ \leftarrow bottom\_up(Rules^0, Rules^+, \emptyset, \_, E^+, E^-)$;

${I_0}^- \leftarrow bottom\_up(Rules^0, Rules^+, \emptyset, \_, E^-, E^+)$;

repeat

%% Vypočítaj $I^- = HB_I - F_{i+1}$;

$I^- \leftarrow bottom\_up(Rules^- \cup Rules^+ \cup Rules^\pm, Rules^+ \cup$

$Rules^\pm, I^+ \cup {I_0}^-, I^+, E^-, E^+)$;

${I_0}^+ \leftarrow I^+$;

%% Vypočítaj $I^+ = T_{i+2}$;

$I^+ \leftarrow$

$bottom\_up(Rules^- \cup Rules^\pm, Rules^+ \cup Rules^\pm, {I_0}^+, I^-, E^+, E^-)$;

until $I^+ = {I_0}^+$;

Algoritmus 6: Výpočet well-founded modelu

Intuitívne, volanie procedúry $bottom\_up(Rules^i, Rules^r, I^p, I^n, E^p, E^n)$ je priamočiary zdola-nahor výpočet najmenšieho pevného bodu programu daného $Rules^i \cup Rules^r$, pričom význam negatívnych literálov v pravidlách je daný $E^n \cup I^n$, a $E^p \cup I^p$ sú brané ako (vopred vypočítané) fakty.

1.2.3 Magická transformácia

Magické transformácie imitujú výpočet *zhora-nadol* pomocou výpočtu *zdola-nahor*. Hlavnou výhodou je, že umožňujú špecializovať výpočet *zdola-nahor* vzhľadom na dotaz a tým zvýšiť efektívnosť vyhodnocovania dotazov.

Magická transformácia je optimalizačná technika, ktorá transformuje datalógový program na iný datalógový program s nasledujúcimi vlastnosťami [5]:

1. Počíta tú istú odpoveď ako pôvodný datalógový program.

2. Keď ho vypočítavame seminaívnou (naívnou) evaluáciou, do výpočtu vchádza tá istá množina faktov (n-tíc) ako pri výpočte zhora nadol (SLD-rezolúciou alebo RGT, či QRGT [5]).

3. Selekcie (väzby argumentov) sa posúvajú od cieľov k podcieľom. Z hlavy pravidla k jednotlivým podcieľom v tele.

Nasledujúce definície sú prevzaté z [7].

Definícia 18 (Stratégia výberu podcieľa)**.** Nech R je pravidlo a p_h je hlavový literál ohraničený na množinu viazaných argumentov. Nech $Lits(R)$ je množina literálov tela pravidla R. *Stratégiou výberu podcieľa* (budeme označovať *sip stratégia*, alebo tiež *sips*) pre pravidlo R je graf s označenými hranami, spĺňajúci nasledovné podmienky:

1. Každý vrchol je buď podmnožina, alebo člen $Lits(R) \cup \{p_h\}$.

2. Každá šípka (orientovaná hrana) je tvaru $N \rightarrow_\chi q$, kde N je podmnožina $Lits(R) \cup \{p_h\}$, q je člen $Lits(R)$ a χ je množina premenných, takých, že každá premenná v χ sa nachádza v q a v niektorom argumente nejakého prvku N.

3. Existuje čiastočné usporiadanie literálov v $Lits(R) \cup \{p_h\}$ také, že

   (a) p_h je prvý.

(b) Pre každú šípku, všetky literály na jej chvoste predchádzajú literál na jej hlave.

(c) Literály, ktoré sa nenachádzajú v sips nasledujú všetky ostatné.

Sips pre program sa skladá zo sips pre každé pravidlo programu.

Definícia 19 (Program s ozdobami)**.** Nech P je program, S je ľubovoľná sip stratégia pre P a Q je dotaz pre P. Program s ozdobami $AP = Adorn(P, S, Q)$ získame nasledovne.

1. Pre každý odvodený predikát p, pre každé pravidlo, ktoré má p v hlave a každú ozdobu a pre p, zostrojíme novú ozdobenú verziu pravidla. Predikát p v hlave nahradíme ozdobeným predikátom p^a. Z množiny S zvolíme takú sip, ktorá je kompatibilná s p^a a nahradíme každý odvodený predikát v tele pravidla ozdobenou verziou, ktorú získame nasledovne. Nahradíme $p_i()$ s ${p_i}^{a_i}$, pričom pozíciu argumentu v a_i označíme ako *viazanú*, ak

   - Všetky premenné, ak sú nejaké prítomné, na danej pozícii sa nachádzajú v označení sip šípky vstupujúcej do literálu.

   Argumenty literálu v novom pravidle ostanú nezmenené.
   Nahradili sme teda pôvodné predikáty a pravidlá množinou ozdobených predikátov a pravidiel.

2. Nahradíme dotaz jeho ozdobenou verziou. Ak predikát dotazu je q, samotný dotaz určuje väzby pre q a teda môžme nahradiť q zodpovedajúcou verziou s ozdobou.

3. Na záver eliminujeme ozdobené predikáty a pravidlá, ktoré ich definujú, ak nie sú dosiahnuteľné z dotazu v grafe volania predikátov.

Intuitívne, ozdobený literál p^a zodpovedá výpočtu predikátu p pričom má niektoré argumenty viazané a ostatné voľné, podla toho, ako určuje ozdoba.

Definícia 20 (Magická transformácia)**.** Nech P je program, S je ľubovoľná sip stratégia pre P a Q je dotaz pre P. Výsledkom *magickej transformácie* je program $MP = Magic(P, S, Q)$, definovaný ako zjednotenie dvoch programov MM a PP, zostrojených nasledovne. Na začiatku sú MM a PP prázdne.

1. Najskôr zostrojíme ozdobenú verziu AP programu P.

2. Zostrojíme nový predikát $magic\_p$ pre každý ozdobený predikát p v AP, pričom arita $magic\_p$ je počet viazaných argumentov v p.

3. Pre každé pravidlo v AP pridáme *upravenú verziu* pravidla do PP. Ak hlava pravidla je $p(\tilde{s})$, modifikovanú verziu tohto pravidla získame pridaním literálu $magic\_p(\tilde{s}^b)$ do tela, kde $\tilde{s}^b$ je množina pozícií viazaných argumentov $p(\tilde{s})$.

4. Pre každé pravidlo R v AP s hlavou $p(\tilde{s})$ a pre každý literál $q(\tilde{t})$ alebo $\neg q(\tilde{t})$ v jeho tele pridáme *magické pravidlo* do MM. Hlava pravidla bude $magic\_q(\tilde{t}^b)$. Telo bude obsahovať všetky literály tela R, ktoré sú na chvoste šípky v S s hlavou šípky $q(\tilde{t})$, okrem prípadu, keď chvost šípky obsahuje špeciálny literál p_h. Vtom prípade ho nahradíme literálom $magic\_p(\tilde{s}^b)$.

5. Zostrojíme *počiatočný fakt* (seed) $magic\_q(\tilde{c}^b)$ z dotazu Q a pridáme ho do MM.

1.2.4 Zovšeobecnená magická transformácia

Podrobný popis zovšeobecnenej magickej transformácie možno nájsť v [3]. Jej hlavnou myšlienkou je ponechať voľné premenné v hlavách magických predikátov. Tým môžeme zabrániť strate niektorých informácií užitočných pri

ich výpočte. Napr. predikat $p(X, X, 1)$ nesie okrem informácie, že tretí argument je viazaný aj informáciu, že prvé dva argumenty sa rovnajú, aj keď nepoznáme ich hodnoty. V takto upravenom programe sa však môzu vyskytovať premenné, ktoré nebudú bezpečné. Vyhodnotenie takéhoto programu vyžaduje zmenu naívnej evaluácie, aby vedela pracovať aj s neuzavretými termami.

My budeme uvažovať trochu odlišný spôsob konštrukcie zovšeobecnenej magickej transformácie, ako tá, ktorá je popísaná v [3]. V rámci konzultácie RNDr. Ján Šturc, CSc. navrhol zovšeobecnenie definície 20 tak, že pri konštrukcii magických predikátov budeme ignorovať ozdoby a každý magický predikát bude mať rovnaké argumenty, ako predikát, ku ktorému prislúcha. Počiatočný fakt bude obsahovať všetky argumenty dotazu.

1.2.5 Magické transformácie a negácia

V časti 1.1.4 sme popísali dôvody pre voľbu well-founded sémantiky pre negáciu v deduktívnych databázach. Magické transformácie popísané v častiach 1.2.3 a 1.2.4 boli však navrhnuté len pre čistý datalóg (bez negácií, funkčných a agregačných symbolov). Nasledujúci príklad demonštruje hlavný problém pri výpočte well-founded modelu programu po magickej transformácii.

Príklad 4. *Uvažujme program na obrázku 1.6 (a). Jeho (dvojhodnotový) well-founded model je* $\{r(a), \neg p(a)\}$. *Ak zvolíme úplnú* zľava-doprava *sips, vzhľadom na dotaz* $p(a)$, *dostaneme magický program, ktorý je na obrázku 1.6 (b). Well-founded model tohto programu je* $\{magic\_p(a), magic\_q(a)\}$, *ktorý sa nezhoduje s pôvodným programom v odpovedi na dotaz* $p(a)$.

Ako už vieme, niektoré tiedy programov s negáciou môžme vyhodnocovať obyčajným iterovaním najmenšieho pevného bodu. Ak by sme chceli zefektívniť výpočet týchto programov pomocou magických transformácií, znovu narazíme problém, ktorý demonštruje ďalší príklad.

Príklad 5. *Program na obrázku 1.7 (a) počíta v orientovanom grafe reprezentovanom reláciou edge také dvojice uzlov* X *a* Y, *pre ktoré platí, že existuje cesta z* X *do* Y, *ale cesta z* Y *do* X *neexistuje. Tento program síce je stratifikovaný, ale keď aplikujeme magickú transformáciu s úplnou* zľava-doprava *sips pre dotaz* $nocyc(a, e)$, *dostávame program 1.7 (b), ktorý už stratifikovaný*

(a) $p(a) \leftarrow q(a), \neg r(a).$
$q(a) \leftarrow \neg q(a).$
$r(a).$

(b) $p(a) \leftarrow magic\_p(a), q(a), \neg r(a).$
$r(a) \leftarrow magic\_r(a).$
$q(a) \leftarrow magic\_q(a), \neg q(a).$
$magic\_q(a) \leftarrow magic\_p(a).$
$magic\_r(a) \leftarrow magic\_p(a), q(a).$
$magic\_q(a) \leftarrow magic\_q(a).$ magic_p(a).

Obr. 1.6: Program s negáciou

nie je. Nech relácia edge $= \{(a,b),(b,c),(c,d),(d,e),(e,a)\}$. *Ak vyhodnotíme pôvodný program naívnou evaluáciou, odpoveďou je prázdna relácia, pretože* X *a* Y *sú súčasťou cyklu v grafe. Keď však rovnakou metódou skúsime vyhodnotiť program po magickej transformácii, výpočet sa zacyklí a nedá žiadnu odpoveď. Stále však môžme použiť lubovoľnú techniku na výpočet well-founded modelu a oba programy dajú rovnakú správnu odpoveď.*

Uvedené dva príklady naznačujú nasledovné.

1. Nestratifikované programy nemusia po magickej transformácii zachovávať well-founded model.
2. Stratifikované programy síce well-founded model zachovávajú, ale nezachovávajú si stratifikovanosť. Nemôžeme ich viac vyhodnocovať iterovaním najmenšieho pevného bodu.

Veta 1.2.2. *Nech* P *je stratifikovaný program a* Q *je dotaz. Nech* S *je ľubovoľná sip stratégia. Potom well-founded modely* Magic(P,S,Q) *a* P *sa zhodujú*

(a) $path(X,Y) \leftarrow edge(X,Y).$

$path(X,Y) \leftarrow edge(X,Z), path(Z,Y).$

$nocyc(X,Y) \leftarrow \neg path(Y,X), path(X,Y).$

$? - nocyc(a,e).$

(b) $nocyc(X,Y) \leftarrow magic\_nocyc(X,Y), \neg path(Y,X), path(X,Y).$

$path(X,Y) \leftarrow magic\_path(X,Y), edge(X,Y).$

$path(X,Y) \leftarrow magic\_path(X,Y), edge(X,Z), path(Z,Y).$

$magic\_path(Y,X) \leftarrow magic\_nocyc(X,Y).$

$magic\_path(X,Y) \leftarrow magic\_nocyc(X,Y), \neg path(Y,X).$

$magic\_path(Z,Y) \leftarrow magic\_path(X,Y), edge(X,Z).$

$magic\_nocyc(a,e).$

$? - nocyc(a,e).$

Obr. 1.7: Magická transformácia stratifikovaného programu

v odpovedi na Q.

Dôkaz. Pozri dôkaz *Theorem 4* v [7]. □

Obidva problémy si získali nemalú pozornosť. Medzi techniky riešiace problém vyhodnocovania nestratifikovaného magického programu získaného transformáciou stratifikovaného programu patria: značenie predikátov, interpreter magických množín, usporadúvanie pravidiel a slabá stratifikácia. Dôsledkom vety 1.2.2 je fakt, že ľubovolná procedúra na výpočet well-founded modelu dokáže tento problém prekonať. Avšak za cenu nižšieho výkonu, keďže vo všeobecnosti pri výpočte well-founded modelu musíme explicitne vyrátať okrem pravdivých faktov aj množinu nepravdivých (alebo jej komplement).

Pri snahe riešiť problém zachovávania well-founded modelu sa naskytá otázka, aká je najvačšia trieda programov, ktoré po magickej transformácii zachovávajú well-founded model. Ako ukazuje nasledujúci príklad, trieda stratifikovaných programov nepokrýva všetky programy s touto vlastnosťou.

Príklad 6. *Znovu uvažujme program na obrázku 1.6. Ak zmeníme poradie predikátov v tele prvého pravidla tak, aby* $\neg r(a)$ *bolo pred* $q(a)$*, well-founded model magického programu sa bude zhodovať s pôvodným v odpovedi na dotaz.*

Medzi triedy programov, ktoré rozširujú triedu stratifikovaných programov patria napr. *modulárne stratifikované programy* [8] a programy s *well-founded sips* [7]. Vychádzajú z faktu, že základným problémom s aplikáciou magických transformácií na nestratifikované programy je vznik magických predikátov trojhodnodnotového charakteru. Popisujú prípady, keď je zaručené, že magické predikáty budú dvojhodnotové.

V [7] je predstavený nový typ transformácie, založenej na magickej transformácii a technike zdvojeného programu. Táto transformácia zachováva well-founded model vzhľadom na dotaz, bez ohľadu na voľbu sips či triedu programov, ktoré su transformované.

Definícia 21 (Well-founded magická transformácia)**.** Nech P je ľubovoľný program, nech S je ľubovoľná sip stratégia, nech Q je lubovoľný dotaz a $MP = Magic(P, S, Q)$. Nech T a $\overline{F}$ sú dva dvojhodnotové modely, reprezentujúce pravdivé a možno pravdivé informácie, respektíve získané vypočítaním well-founded modelu magického programu MP použitím techniky

zdvojeného programu. Nech M reprezentuje všetky magické fakty, ktoré sa nachádzajú v $\overline{F}$.

*V*ýsledkom well-founded magickej transformácie je program $PM = WFMagic(P, S, Q)$, ktorý sa rovná programu PP s pridanými faktami M, pričom PP je množina modifikovaných pravidiel v zmysle definície 20.

Veta 1.2.3 ([7]). *Nech P je program, nech S je ľubovoľná sip stratégia, nech Q je dotaz a $PM = WFMagic(P, S, Q)$. Potom sa well-founded modely PM a P zhodujú v odpovedi na Q.*

Dôsledok 1 ([7]). Nech P je program, nech S je ľubovoľná sip stratégia, nech Q je dotaz a $MP = Magic(P, S, Q)$. Ak model M magických faktov v W^*_{MP} je dvojhodnotový, potom sa well-founded modely MP a P zhodujú v odpovedi na Q.

Dôsledok 2 ([7]). Nech P je program, nech S je ľubovoľná sip stratégia, nech Q je dotaz a $MP = Magic(P, S, Q)$. Ak M, množina magických faktov, je rovná, alebo je nadmnožinou magických faktov v $gfp(F^2_{MP})$, potom sa well-founded modely $M \cup PP$ a P zhodujú v odpovedi na Q.

Poznámka 1.2.4. *W^*_{MP} označuje well-founded model programu MP. Symbol gfp označuje najväčší pevný bod. Podrobný popis sémantiky sa nachádza v [7].*

Kapitola 2

Implementácia

V tejto kapitole detailne popíšeme implementáciu jednotlivých častí programu.

2.1 Základné dátové štruktúry

2.1.1 Literál

Trieda *Literal* je základnou dátovou štruktúrou. Každá jej inštancia reprezentuje jeden literál. Jej hlavnými atribútmi sú meno, zoznam argumentov, ozdoba a informácia o tom, či je literál negovaný.

Okrem základných konštruktorov, umožňujúcich vytvorenie novej inštancie pomocou mena a zoznamu argumentov, alebo ako kópiu iného existujúceho literálu, obsahuje táto trieda aj niekoľko metód, ktoré využívajú triedy popísané nižšie.

Metódy *getpredicate()* a *getpredicate2()* vracajú textový identifikátor predikátu, ktorý k literálu prislúcha, dvoma spôsobmi. V prvom prípade je identifikátor vygenerovaný z mena a ozdoby literálu, v druhom prípade z mena a arity. Prvú metódu využijeme pri magických transformáciách, druhú napríklad pri technike zdvojeného programu, keď budeme potrebovať vytvoriť zoznam IDB predikátov.

Metóda *bind(Vector)* dostane ako parameter zoznam viazaných premenných a upraví podľa neho atribút addorn, ktorý reprezentuje ozdobu. Toto využijeme pri vytváraní programu s ozdobami pre magické transformácie.

Metóda *makeMagic()* prerobí literál na magický. Podľa aktuálnej ozdoby

upraví zoznam argumentov, tak, aby obsahoval len tie viazané a upraví mu meno pridaním prefixu „magic_“. Samozrejme upraví aj ozdobu, aby korešpondovala s novým zoznamom argumentov.

Metóda *getRule()* vráti pravidlo (objekt typu *Rule*), ktorého hlavu tvorí aktuálny literál a ktorého telo je prázdne.

2.1.2 Pravidlo

Trieda *Rule* reprezentuje pravidlo datalógového programu. Skladá sa z literálu *head*, ktorý zodpovedá hlave pravidla a z vektora *goals* ktorý obsahuje literály zodpovedajúce podcieľom pravidla. Ak nie sú prítomné žiadne podciele, *goals* je prázdny.

Konštruktory *Rule(String)*, *Rule(Rule)* a *Rule(Literal,Vector)* vytvárajú novú inštanciu triedy na základe textového reťazca získaného napríklad z parsera, ako kópiu existujúcej inštancie, alebo priamo na základe dvoch parametrov: hlavového literálu a zoznamu literálov reprezentujúcich podciele vytváraného pravidla.

Metóda *bind()* nastaví ozdoby všetkých podcieľov tak, aby zodpovedali úplnej *zľava-doprava* sip stratégii.

2.1.3 Program

Trieda *Program* reprezentuje program v datalógu. Skladá sa z vektora *rules*, ktorý reprezentuje jednotlivé pravidlá programu a literálu *query*, ktorý reprezentuje dotaz (môže mať hodnotu *null*, ak žiadny dotaz nebol zadaný).

Trieda obsahuje dva konštruktory. Konštuktor *Program()* vytvorí prázdny program a konštruktor *Program(String)* dostane ako parameter meno súboru, z ktorého načíta pravidlá, prípadne aj dotaz.

Ďalej sú implementované metódy na pridávanie pravidiel. Metóda *add(Rule)* pridá jedno pravidlo a metóda *add(Program)* pridá všetky pravidlá existujúcej inštancie triedy Program.

Metódy *getPredicates()* a *getIdbPredicates()* vrátia zoznam všetkých IDB predikádov definovaných programom. Prvá využíva metódu *getpredicate()* a druhá *getpredicate2()* triedy *Literal*.

2.1.4 Relácia

Veľmi dôležitou netriviálnou dátovou štruktúrou je trieda *Relation*, reprezentujúca reláciu. Základné atribúty sú *name*, *arity*, *params*, *external* a *tuples*.

Reprezentujú meno relácie, aritu, zoznam atribútov, informáciu o tom, či sa jedná o externú reláciu, uloženú v sql a množinu n-tíc (faktov) uložených v relácii.

Konštruktory *Relation()* a *Relation(String, String[])* vytvoria prázdnu reláciu (s prázdnou množinou *tuples*). Konštruktor *Relation()* je prázdny, nevykonáva žiadnu inicializáciu. Slúži len na zjednodušenú implementáciu podtried triedy *Relation()*. Jedná sa o takzvaný *default* konštruktor. Druhý konštruktor vytvorí reláciu podľa zadaného mena a zoznamu argumentov.

Konštruktor *Relation(String)* dostane ako parameter meno binárneho súboru, v ktorom je uložená relácia metódou *saveToFile(String)*. Inštanciu triedy zostrojí načítaním tohto súboru. Načítaním sa inicializujú všetky atribúty, vrátane množiny *tuples*. Konštruktor priamočiaro využíva nižšie popísanú metódu *loadFromFile(String)*.

Metódy *addTuple(Vector)* a *addTuple(Literal)* slúžia na pridávanie faktov (n-tíc) do relácie. Prvá priamočiaro pridá n-ticu, reprezentovanú vektorom. Druhá pridá n-ticu, ktorá zodpovedá argumentom uvedeného literálu, v prípade, že meno a arita literálu sa zhodujú s menom a aritou relácie.

Metóda *saveToFile(String)* uloží aktuálnu reláciu na disk do súboru s menom, ktoré udáva parameter. Využíva sa tu serializácia objektov reprezentujúcich atribúty *name*, *params* a *tuples*.

Metóda *loadFromFile(String)* slúži na načítanie relácie uloženej uvedeným spôsobom. Toto je jeden zo spôsobov, ako do výpočtu zahrnúť EDB relácie. Inou možnosťou by bolo vytvoriť prázdnu reláciu s názvom a aritou, ktoré by zodpovedali nejakej relácii (tabuľke) v sql databáze a nastaviť atribút *external* na *true*. Samotným výpočtom faktov externej relácie sa však na tejto úrovni nezaoberáme. Podrobnejšie ho popíšeme v časti venovanej metóde *computeRelation(Literal)* triedy *NaiveEvaluation*.

Ďalšie dve metódy majú veľký význam pri vyhodnocovaní relácie v metóde *computeRelation(Literal)*, ako aj pri implementácii joinu a antijoinu.

Metóda *getRename(Vector, Vector)* očakáva na vstupe dva vektory, ktoré reprezentujú dve n-tice, pričom tieto môžu obsahovať premenné. Výstupom je zobrazenie (reprezentované pomocou hešovacej tabuľky), ktoré premenuje premenné obsiahnuté v druhej n-tici tak, aby boli rôzne od všetkých premenných v prvej n-tici.

Metóda *unify(Vector, Vector)* má rovnaké parametre ako predchádzajúca metóda a implementuje riešenie slabej unifikácie. Prvú substitúciu ι nájde pomocou metódy *getRename(Vector, Vector)*. Druhú substitúciu σ nájde Herbrandovou metódou hľadania najvšeobecnejšieho unifikátora. Množina E rovníc je tvorená rovnosťami prvkov zadaných dvoch vektorov po zložkách. Všeobecný algoritmus unifikácie je tu zjednodušený v dôsledku toho, že ako termy prakticky uvažujeme len premenné a konštanty. Konštanty zodpovedjú

triviálnym funkčným symbolom s nulovou aritou.

2.1.5 Spájanie relácií (Join)

Trieda *JoinRelation* je podtriedou triedy *Relation*. Obsahuje navyše konštruktor *JoinRelation(Relation, Relation)*, ktorý vytvorí inštanciu triedy prirodzeným spojením dvoch relácií, ktoré dostane ako parametre.

Uvažujme relácie $r_1(x_1, x_2, \ldots, x_n)$ a $r_2(y_1, y_2, \ldots, y_m)$. Ak by obsahovali len uzavreté termy (čo v našom prípade znamená, že ich n-tice by neobsahovali premenné), môžeme zostojiť ich prirodzené spojenie tak, že porovnáme každú n-ticu z r_1 s každou n-ticou z r_2. Nech $t_1 = (a_1, a_2, \ldots, a_n) \in r_1$ a $t_2 = (b_1, b_2, \ldots, b_m) \in r_2$. Ak implikácia $x_i = y_j \rightarrow a_i = b_j$ platí pre každé $1 \leq i \leq n$ a každé $1 \leq j \leq m$, tak do výslednej relácie, ktorej argumenty sú zjednotením argumentov relácií r_1 a r_2, môžme pridať n-ticu obsahujúcu hodnoty zodpovedajúce argumentom relácie.

Keďže potrebujeme spracovávať aj n-tice, ktoré obsahujú premenné, musíme nahradiť test na rovnosť slabou unifikáciou. Nech ι je taká substitúcia, že pre každé $1 \leq i \leq n$ a každé $1 \leq j \leq m$ platí, že ak a_i a b_j sú premenné, tak $a_i \neq b_i\iota$.

Zostrojíme n-tice $p = (x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m)$ a $t = (a_1, a_2, \ldots, a_n, b_1\iota, b_2\iota, \ldots, b_m\iota)$. Nech $\sigma = unify(p, t)$. Ak $\sigma \neq null$ a q je n-tica mien argumentov výslednej relácie, tak môžeme pridať do výslednej relácie n-ticu $q\sigma$.

2.1.6 Zovšeobecnený rozdiel relácií (Antijoin)

Trieda *AntijoinRelation* je tiež podtrieda triedy *Relation*. Podobne ako trieda *JoinRelation* obsahuje navyše len konštruktor *AntijoinRelation(Relation, Relation)*. Algoritmus výpočtu antijoinu je podobný, ako pri výpočte joinu. Tiež pre každú dvojicu n-tíc overí, či existuje ich unifikácia s argumentami oboch relácií. Nech parametre konštruktora sú relácie r_1 a r_2.

Najskôr inicializuje atribút *tuples* pridaním všetkých n-tíc relácie r_1. Potom porovnáva všetky dvojice n-tíc z $r_1 \times r_2$. Teda každá n-tica $t_1 \in r_1$ je porovnaná s každou n-ticou $t_2 \in r_2$. V prípade úspešnej unifikovateľnosti (pozri výpočet relácie *join*) odstráni t_1 z *tuples*. Po skončení algoritmu je vytvorená nová relácia, ktorej atribút *tuples* obsahuje n-tice $r_1 \triangleleft\triangleright r_2$

2.1.7 Projekcia

Poslednou podtriedou triedy *Relation* je trieda *ProjectRelation*. Parametre konštruktora *ProjectRelation(Relation, String[])* sú relácia r a zoznam argumentov X. Novovytvorená inštancia triedy bude reprezentovať reláciu $\Pi_X r$.

Pre každú n-ticu t v relácii r program najskôr vytvorí substitúciu τ, ktorá priradí argumentom relácie r ich hodnoty v t, a potom do výslednej relácie pridá n-ticu $X\tau$.

2.2 Pomocné triedy

2.2.1 Napojenie na SQL databázu

Napojenie na SQL databázu je realizované pomocou JDBC triedou *JdbcCreator*.

Dva varianty metódy *getConnection* vracajú objekt typu *Connection*. Prvý vytvorí spojenie na základe parametrov *driverClassName*, *host*, *port*, *databaseName*, *subprotocol*, *user* a *password*.

Druhý variant metódy načíta uvedené hodnoty z textového súboru zadaného v parametri *filename*. Nie všetky hodnoty musia byť zadané. Obidva varianty majú navyše booleovský parameter *interactive*. Ak má hodnotu *true*, tak chýbajúce parametre (tie, čo majú hodnotu *null*) budú interaktívne načítane zo štandartného vstupu. V opačnom prípade budú inicializované prednastavenými hodnotami.

2.2.2 Parser

Trieda *Parser* obsahuje niekoľko jednoduchých metód, ktoré sa používajú pri syntaktickej analýze datalógového programu. Využíva ju konštruktor triedy *Program*.

Najdôležitejšou metódou je *getToken(String source, int position, String regexp)*, vracajúca inštanciu triedy *Token* vytvorenej z reťazca *source*, prečítaním lexémy od pozície *position* až po oddelovač definovaný regulárnym výrazom *regexp*.

Metóda *noWhiteSpace(String)* vráti reťazec, vytvorený zo zadaného parametra, odstránením bieleho priestoru (medzier, tabulátorov,...).

Metóda *getName(String source, int position)* vráti token pre meno predikátu.

Metóda *getParams(String source, int position)* vráti špeciálny token pre parametre predikátu.

2.3 Implementácia kľúčových algoritmov

2.3.1 Naívna evaluácia

Trieda *NaiveEvaluation* implementuje naívnu evaluáciu. Poskytuje kompletné uživateľské, ako aj programátorké rozhranie na vyhodnocovanie datalógových programov metódou iterovania najmenšieho pevného bodu. Umožňuje aj napojenie na sql databázu pomocou JDBC rozhrania a tým rozširuje možnosti sql databázy o výhody deduktívneho prístupu. Napríklad umožňuje jednoduchým spôsobom pri písaní dotazu použiť rekurziu. Nie všetky sql systémy ju implementujú.

Všetky spracovávané relácie sú inštancie triedy *Relation*. Tieto sú uložené v množinách relácií, ktoré sú implementovaných ako inštancie triedy *java.util.Hashtable*. Kľúč určujúci reláciu je tvorený menom relácie a jej aritou (*key = relation.name + "/" + relation.arity*).

Keďže v termoch neuvažujeme funkčné symboly, nie je potrebné implementovať procedúry *atov* a *vtoa* v takej podobne ako sme uviedli v časti 1.2.1. Procedúra *atov* je obsiahnutá v metóde *computeRelation* a *vtoa* sme nahradili projekciou. Chceme, aby n-tice v reláciách mohli obsahovať aj funkčné symboly. Nahradíme preto v metóde *computeRelation* term-matching slabou unifikáciou.

Celá databáza je teda tvorená ako množina množín relácií. Reprezentuje ju atribút *database*, ktorý je tiež typu *Hashtable*. Kľúčom je meno množiny relácií. Množín relácií si môže programátor definovať ľubovoľné množstvo. Atribút *activedb* obsahuje zoznam mien množín relácií, ktoré majú vstupovať do výpočtu.

Základnými množinami relácií sú *edb*, *sql*, *idb* a *permanent*.

- Edb relácie, ktoré boli načítane z disku metódou *loadFromFile* triedy *Relation* sa nachádzajú v množine relácií *edb*.
- Edb relácie, ktorých fakty sú uložené v externej sql databáze, sú uložené v množine relácií *sql*.
- Všetky fakty, vypočítané naívnou evaluáciou sa ukladajú do množiny relácií *idb*.

- Fakty, ktoré sa počas celého výpočtu nemenia sú uložené v množine relácií *permanent*.

Množina relácií *permanent* obsahuje spravidla relácie uložené pomocou metódy *saveFacts()*. Táto metóda presunie všetky vypočítané relácie z *idb* do *permanent*. Inými slovami, uloží si aktuálny stav *idb*. Toto môžeme využiť, ak potrebujeme, aby výsledok výpočtu bol použitý v nasledujúcom výpočte ako pevne daný. Metóda *clearFacts()* vyprázdni množinu relácií *permanent*.

Metóda *addRelation(Relation, String)*, slúži na pridanie relácie do množiny relácií určenej menom.

Metódy *addEdb(Relation)* a *addIdb(Relation)* slúžia na pridanie relácie do množiny relácií *edb*, resp. *idb*. Volanie *addEdb(r)* je vlastne to isté ako *addRelation(r, "edb")* a podobne volanie *addIdb(r)* je to isté ako *addRelation(r, "idb")*.

Atribút *pr* obsahuje inštanciu triedy *Program*. Reprezentuje datalógový program, ktorý bude vyhodnotený pri najbližšom volaní metódy *eval()*. Môže byť inicializovaný priamo v konštruktore, alebo ho je možné inicializovať, prípadne zmeniť kedykoľvek neskôr.

Volanie metódy *computeRelation(Literal)* vráti reláciu reprezentujúcu zadaný literál na základe aktuálneho stavu databázy. Výsledná relácia bude obsahovať všetky n-tice obsiahnuté v databáze v reláciách nachádzajúcich sa v množinách relácii zo zoznamu *activedb*, ktoré sú unifikovateľné s argumentami zadaného literálu. Výpočet relácie *rel* k literálu *lit* prebieha v nasledovných krokoch.

1. $tmp = new\ Relation(lit)$
2. Pre každú reláciu r obsiahnutú v niektorej z množín relácií v *activedb*, ktorej meno a arita sa zhodujú s menom a aritou literálu *lit*, rob nasledovné:
   - Ak r nie je externá (sql) relácia, pridaj do *tmp* všetky jej n-tice
   - Ak r je externá relácia, vytvor sql dotaz zodpovedajúci argumentom *rel* a vykonaj sql volanie. Výsledné n-tice pridaj do *tmp*.
3. Pre každú n-ticu t v *tmp* rob nasledovné:
   - $\sigma := unify(lit.params, t)$
   - Ak $\sigma \neq null$ pridaj do *rel* n-ticu $(lit.params)\sigma$

Poznámka 2.3.1. *Pri výpočte sa neberie do úvahy, či je daný literál negovaný alebo nie.*

Metóda *iteration()* implementuje jednu iteráciu naívnej evaluácie. Algoritmus prejde všetkými pravidlami programu pr a pre každé pravidlo odvodí všetky možné fakty. Tieto ukladá do pomocnej množiny relácií *iteration*. Pri spracovaní pravidla postupne pomocou volania metódy *computeRelation(Literal)* vypočíta reláciu r_i pre každý podcieľ a G_i a zostrojí reláciu h pre hlavu pravidla, tak, aby platilo:

Nech $r_1, r_2, \ldots, r_k$ sú vypočítané relácie pre nenegované podciele a nech $r_{k+1}, r_{k+2}, \ldots, r_n$ sú vypočítané relácie pre negované podciele. Potom

- Ak pravidlo obsahuje aspoň jeden pozitívny podcieľ, tak $h = \Pi_{head.params}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_k \triangleleft\triangleright r_{k+1} \triangleleft\triangleright r_{k+2} \triangleleft\triangleright \ldots \triangleleft\triangleright r_n)$
- Ak pravidlo obsahuje iba negatívne podciele a $r_{k+1} = \emptyset, r_{k+2} = \emptyset, \ldots, r_n = \emptyset$, tak $h = \{head.params\}$
- Inak $h = \emptyset$

Pokiaľ je to možné, snaží sa predísť situácii, aby bol negovaný podcieľ vyhodnocovaný ako prvý. Preto negatívne podciele vyhodnocuje až po vyhodnotení všetkých pozitívnych.

Po vypočítaní celej množiny *iteration*, ktorá teraz obsahuje všetky fakty odvodené v jednej iterácii algoritmus overí, či nastala zmena. Teda či pribudli, alebu ubudli nejaké fakty. Toto urobí porovnaním množiny práve vypočítaných relácií *iteration* s množinou relácií z predchádzajúcej iterácie *idb*. Porovnanie prebehne v najviac troch krokoch. Inicializujme booleovskú premennú *change* na hodnotu $false$.

1. Ak $|iteration| \neq |idb|$ tak $change := true$
2. Inak, ak pre lubovoľné relácie $q \in iteration$ a $q' \in idb$ reprezentujúce rovnaký predikát platí $|q| \neq |q'|$ tak $change := true$
3. Inak, ak pre lubovoľné relácie $q \in iteration$ a $q' \in idb$ reprezentujúce rovnaký predikát platí $q \nsubseteq q'$ tak $change := true$

Na záver algoritmus nahradí množinu relácií *idb* množinou *iteration* a tejto priradí prázdnu množinu. Návratovou hodnotou volania *iteration()* bude hodnota *change*.

Metóda *eval()* implementuje samotný výpočet. Jej algoritmus vyzerá nasledovne:

1. Ak $pr = null$, vráť $null$.

2. Opakovane vykonávaj *iteration()*, pokiaľ nevráti hodnotu $false$.

3. Ak pr obsahuje dotaz $query$, vráť $answer := computeRelation(query)$

4. Inak vráť *null*

Metóda *eval(Program)* najskôr inicializuje atribút pr na zadaný parameter a potom zavolá metódu *eval()*.

2.3.2 Technika zdvojeného programu

Trieda *DoubledProgramEvaluation* implementuje výpočet well-founded modelu pomocou algoritmu, ktorý bol popísaný v časti 1.2.2.

Metóda *makeDoubled(Program)* zostrojí dvojicu programov $pr1$ a $pr2$, ktoré reprezentujú zdvojený program podľa definície 17. Zostrojí tiež inicializačný program *init* zložený z čiarkovaných atómov nečiarkovanej polovice programu v podobe triviálnych pravidiel bez pravej strany. Tento program sa vyhodnotí v inicializačnej fáze a tým zabezpečíme, aby v prvej iterácii algoritmu boli jeho fakty považované za pravdivé.

Atribút *bottomup* inicializujeme novou inštanciou triedy *NaiveEvaluation*.

Metóda *eval(Program, boolean)* vypočíta well-founded model zadaného programu technikou zdvojeného programu. Prvým parametrom je vyhodnocovaný program pr, druhým je booleovský parameter *initialize*, ktorý určuje, či má prebehnúť inicializačná fáza (Ak neprebehne, ostáva výpočet inicializovaný výsledkom predchádzajúceho výpočtu). Výsledok sa uloží do atribútu *wfmodel*. Nájdenie well-founded modelu prebehne v nasledujúcich krokoch:

1. Ak $pr = null$, vráť *null*

2. Inicializuj atribúty $pr1$, $pr2$ a *init* volaním $makeDoubled(pr)$.

3. Ak $initialize = true$, vykonaj nasledovné:

   - $bottomup.eval(init)$
   - $bottomup.saveFacts()$
   - $bottomup.eval(pr1)$
   - $bottomup.saveFacts()$

4. Pokiaľ sa mení obsah $wfmodel$, opakuj nasledovné:

   - $bottomup.eval(pr2)$
   - $bottomup.saveFacts()$

- Priraď do $wfmodel$ množinu relácií *bottomup.permanent* (t.j. vypočítané fakty)
- $bottomup.eval(pr1)$
- $bottomup.saveFacts()$
- Pridaj do $wfmodel$ všetky vypočítané fakty z *bottomup.permanent*

5. Ak pr obsahuje dotaz $query$, vráť $bottomup.computeRelation(pr.query)$
6. Inak vráť *null*

Implementovaná je aj verzia metódy $eval(Program)$ s jedným parametrom. Volanie $eval(pr)$ je ekvivalentné s volaním $eval(pr, true)$.

2.3.3 Magická transformácia

V implementácii magickej transformácie budeme vychádzať z definícií 19 a 20. Budeme uvažovať iba úplnú *zľava-doprava* sip stratégiu, ktorá je asi najpoužívanejšia v implementáciách deduktívnych databáz. To znamená, že podciele každého pravidla budú vyhodnocované v poradí, v akom sa vyskytujú v zápise pravidla.

V metóde *addorn* budeme namiesto generovania všetkých možných ozdobených verzií každého pravidla v prvom kroku a ich následnej eliminácie v poslednom kroku generovať len tie ozdobené pravidlá, ktorých prislúchajúce predikáty sú v grafe volania predikátov dosiahnuteľné z dotazu. Toto spravíme zhora-nadol prehľadávaním do šírky grafu volania predikátov.

Hotovú metódu *addorn* môžeme vidieť na obrázku 2.1. Premenná *front* reprezentuje FIFO rad, ktorý použijeme pri prehľadávaní grafu volania predikátov. Na začiatku ho inicializujeme vložením predikátu pre dotaz. V cykle vzdy vyberieme predikát zo začiatku radu, zmeníme ho na zodpovedajúci ozdobený predikát a všetky predikáty, na ktorých závisí, pridáme na koniec radu. Algoritmus končí spracovaním všetkých dosiahnuteľných predikátov, teda vyprázdnením radu.

Pri implementácii metódy *magic* postupujeme podľa definície 20.

1. Ozdobenú verziu programu získame jednoduchým volaním *addorn*.
2. Vytvorenie pravidla pre predikát budeme považovať súčasne aj za vytvorenie predikátu. Druhý krok algoritmu je teda obsiahnutý v nasledujúcich krokoch a nie je nutné ho robiť explicitne.
3. Nasleduje cyklus, v ktorom zostrojíme upravené pravidlá a pridáme ich do *PP*. Volanie metódy *makeMagic()* prerobí literál $p(\tilde{s})$ na $magic\_p(\tilde{s}^b)$.

```
public static Program addorn(Program p, Literal query) {
    Program ap = new Program();
    Vector front = new Vector();
    front.add(query.getPredicate());
    int i=0;
    while(i<front.size()) {
        String pattern=(String)front.elementAt(i);
        for(int j=0;j<p.rules.size();j++) {
            Rule r = (Rule)p.rules.elementAt(j);
            Rule rr=new Rule(r);
            if(rr.bind(pattern)) {
                ap.add(rr);
                for(int k=0;k<rr.goals.size();k++) {
                    Literal goal = (Literal)rr.goals.elementAt(k);
                    if(!front.contains(goal.getPredicate())) {
                        front.add(goal.getPredicate());
                    }
                }
            }
        }
        i++;
    }
    return ap;
}
```

Obr. 2.1: Vytvorenie programu s ozdobami

4. Ďalší cyklus implementuje štvrtý krok definície. Formulácia „Pre každé pravidlo R v AP s hlavou $p(\tilde{s})$ a pre každý literál $q(\tilde{t})$ alebo $\neg q(\tilde{t})$ v jeho tele“ zodpovedá *for*-cyklu s riadiacou premennou i a v ňom vnorenému *for*-cyklu s riadiacou premennou j.

   Formulácia „všetky literály tela R, ktoré sú na chvoste šípky v S s hlavou šípky $q(\tilde{t})$“ môžeme nahradiť formuláciou „všetky literály tela, ktoré sa v zápise pravidla nachádzajú naľavo od $q(\tilde{t})$“, keďže uvažujeme iba *zľava-doprava* sips. Z rovnakého dôvodu „keď chvost šípky obsahuje špeciálny literál p_h“ môžeme preformulovať ako „keď sa jedná o literál prvý z ľava v zápise tela pravidla“.

2.3.4 Zovšeobecnená mágia

Implementáciou zovšeobecnenej magickej transformácie je trieda *GeneralizedMagic*. Poskytuje rovnaké rozhranie, ako trieda *Magic*, líši sa len v implementácii. Konštrukciu načrtnutú v časti 1.2.4 sme dosiahli úpravou metódy *addorn* tak, že sme vo všetkých ozdobách nahradili výskyt znaku „f“ znakom „b“, čím boli všetky argumenty automaticky považované za viazané a teda boli ponechané.

```
public static Program magic(Program p, Literal query) {
    Program ap = addorn(p,query);
    Program pp = new Program();
    Program mm = new Program();
    Vector heads = new Vector();

    for(int i=0;i<ap.rules.size();i++) {
        Rule r = (Rule)ap.rules.elementAt(i);
        Rule rr = new Rule(r);
        Literal mfact = new Literal(rr.head);
        mfact.makeMagic();
        rr.goals.add(0,mfact);
        pp.add(rr);
        String head=r.head.name+"/"+r.head.addorn.length();
        heads.add(head);
    }
    for(int i=0;i<ap.rules.size();i++) {
        Rule r = (Rule)ap.rules.elementAt(i);
        Literal ph = new Literal(r.head);
        ph.makeMagic();
        for(int j=0;j<r.goals.size();j++) {
            Literal q=(Literal)r.goals.elementAt(j);
            Literal magicq = new Literal(q);
            magicq.makeMagic();
            magicq.negated=false;
            Vector v = new Vector();
            v.add(ph);
            for(int k=0;k<j;k++) {
                Literal lit=(Literal)r.goals.elementAt(k);
                v.add(lit);
            }
            Rule mr = new Rule(magicq,v);
            String qstring = q.name+"/"+q.addorn.length();
            if(heads.contains(qstring)) {
                mm.add(mr);
            }
        }
    }

    pp.query = new Literal(query);
    Literal seed = new Literal(query);
    seed.makeMagic();
    mm.add(new Rule(seed,new Vector()));
    pp.add(mm);
    return pp;
}
```

Obr. 2.2: Vytvorenie magického programu

2.3.5 Mágia zachovávajúca well-founded sémantiku

Trieda *WellFoundedMagicSetsEvaluation* implementuje hľadanie well-founded modelu použitím magických transformácií a zdvojeného programu na základe vety 1.2.3 a dôsledku 1.

Metóda *twoValuedMagicFacts()* vráti booleovskú hodnotu (*true/false*) reprezentujúcu skutočnosť že magické fakty sú dvojhodnotové. Inými slovami, či vo well-founded modeli pre každú reláciu reprezentujúcu čiarkovaný magický predikát existuje relácia reprezentujúca rovnaký nečiarkovaný predikát taký, že obe relácie majú rovnaké množiny n-tíc.

Samotný výpočet je realizovaný metódou *eval(Program)*.

Pri vyhodnocovaní well-founded modelu daného programu najskôr aplikujeme magickú transformáciu. Potom transformovaný program vyhodnotíme technikou zdvojeného programu. V zmysle dôsledku 1 overíme dvojhodnotovosť magických faktov. Ak sú všetky magické fakty dvojhodnotové, výpočet skončí. Ak nie, pridáme magické fakty do množiny *možno pravdivých* faktov a vyhodnotíme takto upravený program. Program nevyhodnocujeme odznova, ale ponecháme už vypočítané fakty ako počiatočný stav výpočtu. Na základe vety 1.2.3 dostaneme po skončení druhej fázy well-founded model zadaného programu.

Kapitola 3

Záver

3.1 Systémy implementujúce Datalóg

3.1.1 IRIS

IRIS je open-source uživateľské aj programátorské prostredie pre vyhodnocovanie datalógových programov obsahujúcich bezpečné, alebo nebezpečné pravidlá, rozšírené o funkčné symboly, XML dátové typy, zabudované predikáty a (lokálne) stratifikovanú alebo well-founded negáciu ako zlyhanie (*negation as failure*).

Za bezpečné je tu považované také pravidlo, v ktorom sú všetky vyskytujúce sa premenné ohraničené. Premenná je ohraničená, ak je splnená jedna z nasledujúcich podmienok

- Vyskytuje sa v pozitívnom obyčajnom predikáte
- Vyskytuje sa v pozitívnej rovnosti s konštantou
- Vyskutuje sa v pozitívnej rovnosti s inou premennou, ktorá je ohraničená

Program voliteľne umožňuje nasledovné rozšírenia bezpečných pravidiel:

- Pravidlá obsahujúce premenné, ktoré sa nachádzajú iba v obyčajných negovaných predikátoch (a nikde inde) môžu byť považované za bezpečné.

- Premenná v aritmetickom predikáte môže byť považovaná za ohraničenú, ak všetky ostatné premenné sú ohraničené.

Výpočet sa realizuje pomocou metód *zdola-nahor* a je optimalizovaný *magickými transformáciami.*[12]

3.1.2 Coral

Coral je deduktívny systém s podporou bohatého deklaratívneho jazyka a rozhrania pre C++, ktoré umožňuje kombináciu deklaratívneho a imperatívneho programovania. Deklaratívny dotazovací jazyk podporuje všeobecné *Hornove klauzuly* rozšírené komplexnými termami, množinovými zoskupeniami (*set-grouping*), agregáciou, negáciou a reláciami obsahujúcimi n-tice s premennými (kvantifikovanými všeobecným kvantifikátorom). Autori Coralu tvrdia, že nepoznajú žiaden ďalší systém okrem XSB, ktorý by umožňoval použitie neuzavretých termov v dátach.

Implementácia ponúka široké spektrum evaluačných stratégií. CORAL spracováva nielen dáta uložené v operačnej pamäti, ale aj dáta na pevnom médiu. Tieto sú podporované použitím ukladacieho manažéra EXODUS, ktorý navyše umožňuje spravovanie transakcií a uživateľské prostredie na báze *klient-server* architektúry. Umožňuje taktiež komunikáciu s externou databázou cez rozhranie ODBC.

Negácia je umožnená pre modulárne stratifikované programy.[14]

3.1.3 LDL++

Systém LDL++ obsahuje dve rozšírenia stratifikovaných programov. Tzv. *XY-stratifikáciu* a použitie konštrukcie *choice*, ktorá reprezentuje funkčné závislosti medzi atribútmi predikátov. Obidve konštrukcie sú založené na sémantike stabílnych modelov. Komunikácia s externou databázou je taktiež možná pomocou rozhrania JDBC.[13]

3.1.4 XSB

Systém XSB implementuje väčšinu funkcionality Prologu a obsahuje aj viacero rozšírení. V kontexte tejto práce sú významné nasledovné dve rozšírenia.

- Vyhodnocovanie dotazov v rámci well-founded sémantiky pomocou úplnej SLG rezolúcie

- Implementácia HiLog termov

[16]

3.1.5 DES

DES (Datalog Education System) je voľne šíriteľná, multiplatformová implementácia deduktívnej databázy založená na Prologu. Umožňuje stratifikovanú negáciu a deklaratívne ladenie programov.[15]

3.1.6 Glue-Nail

System Glue-Nail je kombinuje v sebe dva jazyky. Deklaratívny jazyk Nail a procedurálny Glue. Kompilátor Nail používa variant algoritmu magických transformácií a podporuje well-founded modely. Atribút n-tice je reprezentovaný uzavretým *Nailog* termom. Nailog je rozšírením obvyklej syntaxe a sémantiky termov v logickom programovaní a je podmnožinou HiLogu. HiLog a Nailog majú druhorádovú syntax, ale prvorádovú sémantiku. Povolené sú aj premenné ako meno predikátu. Interpreter pri porovnávaní termov používa term-matching namiesto úplnej unifikácie, čo mu umožňuje podmienka uzavretosti termov v n-ticiach. Vo všeobecnosti je term-matching efektívnejší, ako unifikácia.[17]

3.2 Dosiahnuté výsledky

Naša aplikácia umožňuje použitie triviálnych neuzavretých termov v dátach. To znamená, že n-tice v reláciách môžu obsahovať premenné, nie však funkčné symboly. Jediné systémy, o ktorých vieme, že toto umožňujú sú Coral a XSB. Tieto systémy dovoľujú navyše v termoch aj funkčné symboly.

Na rozdiel od systému Coral, naša aplikácia v rámci well-founded sémantiky nekladie obmedzenie na modulárnu stratifikovanosť programov.

Oproti XSB, naša aplikácia ponúka jednoduchšiu a intuitívnejšiu manipuláciu s reláciami uloženými v SQL databáze. V našej implementácii nie je nutné explicitne importovať každú reláciu, ktorú chceme použiť pri výpočte a tým dodržiavame ideu jednotného rozhrania.

3.3 Možné rozšírenia

Podobne ako Coral, aj naša aplikácia sa skladá z jednotlivých modulov, ktoré sú použiteľné aj samostatne. Súčasne vytvárajú nielen užívateľské, ale aj programátorské rozhranie. Programátor tak môže jednoducho použiť nami vytvorené triedy ako programátorské knižnice a zakomponovať uvedené techniky do vlastných programov.

Možným rozšírením by mohlo byť napríklad vytvorenie grafického používateľského rozhrania a tým zvýšiť komfort užívateľa.

Ďalej by sme mohli implementovať aj efektívnejšie techniky vyhodnocovania *zdola-nahor* ako napr. *seminaívnu evaluáciu*. Podobne je otvorená aj možnosť naprogramovať ďalšie varianty magických transformácií. Tieto techniky by bolo možné jednoduchým spôsobom kombinovať v záujme dosiahnutia čo najvyššej efektivity výpočtu.

Zoznam použitej literatúry

[1] A. Tarski *A lattice-theoretical fixpoint theorem and its applications*, Pacific Journal of Mathematics 5:2: 285–309.

[2] H. Garcia-Molina, J. D. Ullman *Database Systems - The Complete Book*, Prentice Hall PTR, Upper Saddle River, NJ, USA

[3] J. D. Ullman: *Database and Knowledge-Base Systems, Vol. II.*, Computer Science Press, New York, 1988

[4] T. Plachetka: *Prednášky z predmetu Úvod do databázových systémov*, domovská stránka <http://www.dcs.fmph.uniba.sk/∼plachetk/TEACHING/DB2007/>, 2007

[5] J. Šturc: *Prednášky z predmetu Relačné a logické bázy dát*, domovská stránka <http://www.dcs.fmph.uniba.sk/∼sturc/databazy/rldb/>, 2008

[6] M. H. Van Emden, R. A. Kowalski: *The semantics of predicate logic as a programming language*, Journal of the ACM, 23(4):733-742, October 1976

[7] D. B. Kemp, ,D. Srivastava, P. J. Stuckey, *Bottom-up Evaluation and Query Optimization of Well-Founded Models*, Elsevier Science Publishers Ltd. Essex, UK, 1995

[8] K. A. Ross, *Modular Stratification and Magic Sets for Datalog Programs with Negation*, Journal of the ACM (JACM), 1994

[9] M. Gelford, V. Lifschitz *The stable model semantics for logic programming*, Proc. Fifth ICLP, MIT Press, Cambridge MA, 1988

[10] A. Van Gelder *The Alternating Fixpoint of Logic Program with Negation*, ACM Press New York, NY, USA, 1989

[11] A. Van Gelder, K. A. Ross, J. S. Schlipf *The Well-Founded Semantics for General Logic Programs*, Journal of the ACM, 1991

[12] Projekt *IRIS reasoner*, domovská stránka <http://iris-reasoner.org/>

[13] F. Arni, K. Ong, S. Tsur, H. Wang, C. Zaniolo *The Deductive Database System LDL++*, <http://arxiv.org/abs/cs/0202001v1>, 2002

[14] R. Ramakrishnan, D. Srivastava, S. Sudarshan, P. Seshadri: *The CORAL Deductive System*, The VLDB Journal, Special Issue on Prototypes of Deductive Database Systems.

[15] Projekt *Datalog Educational System*, domovská stránka <http://www.fdi.ucm.es/profesor/fernan/DES/>

[16] Projekt *XSB*, domovská stránka <http://xsb.sourceforge.net/>

[17] M. A. Derr, S. Morishita, G. Phipps: *The glue-nail deductive database system: design, implementation, and evaluation.*, The VLDB Journal 3, 2 (Apr. 1994), 123-160.

Prílohy

K diplomovej práci je priložené CD, ktoré obsahuje:

- zdrojové kódy aplikácie,
- skompilovanú aplikáciu,
- binárne knižnice potrebné pre beh aplikácie,
- používateľský manuál,
- príklady,
- elektronickú verziu diplomovej práce.

V súbore obsah.txt v hlavnom adresári priloženého CD je popísaná štruktúra CD.