

Ročníkový projekt Datalog->RA

vývoj projektu v druhom semestri

Ciele projektu v letnom semestri

Cieľom projektu v tomto semestri bolo zamerať sa najskôr na manuálnu evaluáciu (aj rekurzívnych) Datalogových programov a napredovať k ich automatickej evaluácii. Dôraz sa pri tom kládol na konformitu s well-founded modelom.

Stručný popis práce počas semestra

Predovšetkým za účelom testovania, ale aj spohodnenia práce s projektom, som počas letného semestra vytvoril aj triedy na jednoduchšiu obsluhu už existujúcich nástrojov zo zimného semestra. K tomuto patrí implementácia triedy *Database*, ktorá slúži na načítavanie externých databáz a manipuláciu s nimi. Na obsluhu testov som vytvoril interface *Test*, pomocou ktorého je možné pridávať manuálne implementované testy.

Samotná evaluácia dotazov je vykonávaná triedami *Predicate* a *Query*. Úlohou triedy *Predicate* je vybudovať operátor relačnej algebry prevažne za použitia prostriedkov z minulého semestra. Trieda *Predicate* je taktiež rozumným spôsobom rozšíriteľná o automatický preklad Datalogu. Trieda *Query* má za úlohu vykonávať evaluačný cyklus a vyhodnotiť finálny výsledok dotazu. V procese vývoja som postupne implementoval niekoľko spôsobov evaluácie aby som sa dopracoval k evaluácii podľa well-founded modelu.

API projektu

Vytvoril som veľmi jednoduché minimalistické API, ktoré slúži na načítavanie projektu a spúšťanie testov. Toto bolo nevyhnutné na umožnenie testovania konkrétnych funkcionalít projektu.

- Príkaz „load <adresár>” načíta všetky relácie v zadanom adresári do momentálne používanej databázy. Adresár musí byť uložený v priečinku resources. Relácie musia mať príponu „.txt” a ich názov sa odvádza z názvu priečinku. Riadky relácie v dokumente sú oddelené koncom riadku a jednotlivé atribúty sú oddelené znakom „|”.
- Príkaz „test <test>” inicializuje a spustí zadaný test. Testy sú napevno zakódované v balíčku datalog_ra.Test.
- Príkaz „save <adresár>” uloží momentálne používanú databázu do zadaného adresáru. Keďže databáza sa za behu nemení, tento príkaz nemá zatiaľ využitie.

Stručný popis funkcionality nových tried projektu

- *Initialization*. Jednoduchý interface obsahujúci jedinú funkciu „loadRelation”. Táto funkcia bola v podobnom tvare už súčasťou main a bolo potrebné ju zaradiť inam. Ako argument dostane textový súbor a vráti reláciu, ktorú tento súbor obsahuje.
- *Database*. Reprezentuje jednu databázu alebo súbor relácií. Obsahuje relácie, ktorým prislúchajú názvy v podobe stringu. Relácie je možné adresovať iba podľa názvu metódou get. Umožňuje načítanie a uloženie do súboru, prístup k reláciám a zoznamu relácií, pridávanie, nahrádzanie a rozširovanie relácií, nahrádzanie množiny relácií (updateFrom), porovnávanie s druhou databázou a vytvorenie identickej databázy.

- *Predicate*. Trieda reprezentujúca Datalogový predikát. Využil som spôsob akým funguje program RAM-ková databáza a tak som zjednodušil výpočet výslednej relácie. Predikát obsahuje kladný počet podcieľov, ktoré sa delia na pozitívne a negatívne. Výsledok je možné vypočítať ako antijoin joinu všetkých pozitívnych podcieľov a joinu všetkých negatívnych podcieľov (toto vysvetľujem podrobnejšie v štruktúre prekladu pravidla). Týmto sa výpočet prirodzene rozdelil na niekoľko častí:
 - Parsovanie podcieľov zo zdrojového reťazca v Datalogu. (*Zatiaľ neimplementované.*)
 - Budovanie podmienok z podcieľov a ich argumentov. Sú 3, pre pozitívny join, pre negatívny join, pre výsledný antijoin. (*Zatiaľ neimplementované.*)
 - Vybudovanie operátora nad vstupnými databázami. Vstupné databázy sú dve, pozitívna a negatívna, čo je nevyhnutné pre správne vyhodnocovanie rekurzívnej podčiary podľa well-founded modelu.
 - Ostatné nástroje potrebné pre funkčnosť *Predicate*: aktualizovanie databáz, materializácia výsledku atď.
- *Query*. trieda reprezentujúca celý Datalogový dotaz. Obsahuje celý evaluačný cyklus a stará sa o aktualizáciu databáz a operátorov v predikátoch medzi cyklami a nakoniec vyskladanie výsledku. Cyklus evaluácie je popísaný v nasledujúcej časti.

Evaluácia dotazov podľa well-founded modelu

Well-founded model vyžaduje aby každý riadok (ďalej nazývam tuple) výsledku bol odvoditeľný z pôvodných relácií alebo ostatných, už určite pravdivých, výsledkov. Problém výpočtu tohto modelu je silno spätý s rekúziou a negáciou.

Prvým krokom v evaluácii bol vytvoriť metódy na skladanie operátorov relačnej algebry. Túto úlohu plní trieda *Predicate*. V teórii je možné pomocou takto vyskladaného operátora počítať akýkoľvek nerekurzívny program Datalogu. Na výpočet rekurzívnych dotazov je ale nevyhnutné zaobchádzať s každým pravidlom (predikátom) samostatne. Triedu *Predicate* je možné rozšíriť o niektoré metódy na umožnenie automatického prekladu ako som už spomínal v predošlej časti.

Druhým krokom bola implementácia takzvanej naivnej evaluácie. Pri tejto metóde sa jednoducho v cykle vyhodnocujú všetky predikáty a finálny výsledok sa dosiahne ak sa od minulej iterácie nezmenil medzivýsledok. Výhodou tejto metódy je predovšetkým jednoduchosť implementácie. Nevýhodou však je, že v mnohých prípadoch (napríklad ak sa dva predikáty v rekúzii navzájom vylučujú) bude program počítať donekonečna. Tento spôsob používa dnes veľká časť databázových jazykov s pridaním obmedzení na programy aby nebolo možné napísať program, ktorý neskončí. Test, ktorý demonštruje fungovanie tejto metódy, je „winmove“ s databázou „winmove“ alebo „winmove2“, avšak s databázou „winmove3“ sa ten istý test zacyklí.

Tretí krok je prechodom k well-founded modelu. Hlavný cyklus naivnej evaluácie ostáva nezmenený, ale zmení sa testovanie ukončenia: Nech R_i je výsledok i -tej iterácie cyklu, potom cyklus skončí, keď nájdeme $R_k = R_{k-2}$ a výsledok vypočítame ako $R_k \cap R_{k-1}$. Takýmto spôsobom evaluácia skončí na každom programe. Pre niektoré z pridaných programov ale dostaneme výsledky, ktoré nie sú pravdivé vo well-founded modeli. Táto metóda by zastala a vrátila správny výsledok na kombinácii „winmove“/„winmove3“ ale na teste „well“/„well“ by sa tuple „1“ nesprávne dostal do výsledku.

Finálna modifikácia naivnej evaluácie, ktorá pracuje na základe well-founded modelu, spočíva v konzervatívnej manipulácii s negáciou. Evaluácia má 2 vnorené cykly. Vo vonkajšom cykle sa zafixujú zdrojové relácie pre negatívne podciele. Tieto relácie sa berú z predošlej iterácie. Vo vnútornom cykle sa začne pracovať vždy nanovo so vstupnými reláciami a medzi iteráciami sa aktualizujú iba pozitívne podciele predikátov. Takto sa vypočíta ďalší medzivýsledok naivnou evaluáciou. Tá vždy skončí, nakoľko sú fixované hodnoty negatívnych podcieľov. Vonkajší cyklus končí rovnako ako pri predošlom kroku.

Nasledujúci pseudokód popisuje finálnu verziu well-founded evaluácie. Naďalej R_i označuje obsah databázy R po i -tej iterácii (vonkajšieho) cyklu.

```

do {
    Rk-2 = Rk-1;
    Rk-1 = Rk;

    Rk = SourceEDB;

    for (p in predicates) {
        p.updateNegatives(Rk-1 ); //nahrad' zdroje negatívnych podcieľov reláciami z Rk-1
    }

    Database R'k-1; //databáza predošlého medzivýsledku vnútorného cyklu
    do { //vnútorný cyklus
        R'k-1 = Rk;
        for (p in predicates) {
            p.updatePositives(Rk); //nahrad' zdroje pozitívnych podcieľov reláciami z Rk
            p.buildOperator(); //operátor bol postavený na starých reláciách, treba ho prerobiť
            Rk.append(p , p.result()); //výsledky sa pridajú k už existujúcim
        }
    }
    while (R'k-1 ≠ Rk);
} while (Rk-2 ≠ Rk);

```

Štruktúra prekladu pravidla

Nech pravidlo P má podciele $p_1, \dots, p_k, n_1, \dots, n_l$, kde p_1, \dots, p_k sa v pravidle vyskytujú v pozitívnom význame a n_1, \dots, n_l v negatívnom. Prirodzeným spôsobom by sme pravidlo do relačnej algebry preložili ako $(\dots(p_1 \bowtie p_2) \bowtie \dots \bowtie p_k) \triangleright n_1 \triangleright n_2 \dots$. Na takúto implementáciu je potrebné vytvoriť približne $l+k$ podmienok. Vytváranie podmienok tvorí veľkú časť zložitosti prekladu. Počet podmienok vieme redukovať takto: $(p_1 \bowtie p_2 \bowtie \dots \bowtie p_k) \triangleright (n_1 \bowtie n_2 \bowtie \dots \bowtie n_l)$.

Preklad teraz vyžaduje vytvoriť jednu podmienku pre join pozitívnych podcieľov, jednu pre join negatívnych podcieľov, jednu pre výsledný antijoin a nakoniec transformáciu pre projekciu na výsledok. Vytváranie podmienok predvediem na príklade.

Uvažujme relácie: *kopal*(človek, jama), *padol*(človek, jama)

Preložme Datalogový predikát

result(X, Y) \Leftarrow *padol*(X, Z), *padol*(Y, Z), not *kopal*(X, Z), not *kopal*($Y, _$).

do relačnej algebry.

Najprv urobíme *padol* \bowtie *padol*, pričom v podmienke c podľa indexov uvedieme $1 = 3$ (čo znamená $Z = Z$). Za účelom vyjadrenia podmienky c vytvoríme triedu *CompareCondition*(1, 3).

Teraz analogicky vytvoríme negatívny join *kopal* \bowtie *kopal*, pričom podmienka $c = \text{TRUE}$ (čo vyjadruje kartézsky súčin, nie prirodzený join). Na to použijeme predvytvorenú *TrueCondition*.

Teraz urobíme antijoin oboch joinov a v podmienke uvedieme $0 = 4, 2 = 6, 1 = 5$ ($1 = 3$ už je obsiahnuté v pozitívnom joinu a $3 = 5$ vyplýva z tranzitívnosti rovnosti). Aby sme spojili podmienky použijeme *TransformationSequence*, do ktorého zadáme zoznam všetkých podmienok. Nakoniec urobíme projekciu výsledku. Tá ako podmienku dostane

ProjectionTransformation s postupnosťou *true, false, true* (čiže projekcia na prvé stĺpce X a Y , zvyšných 5 stĺpcov je implicitne *false*).

Ostatné poznámky

- Vytvoril som operátor *Intersection* (prienik). Ten bol užitočný na vyskladanie finálne výsledku v query.
- Vytvoril som štruktúru metód *copy()* a *compare()*, pomocou ktorých sa dá vytvoriť kópia resp. porovnať databázy, relácie, tuple a atribúty. Dva objekty nasledujúcich typov sú rovné ak:
 - *Attribute*: obsahujú rovnaký (nie nutne ten istý) reťazec
 - *Tuple*: obsahujú rovnaký počet atribútov a atribúty s rovnakým indexom sa rovnajú
 - *Relation*: obsahujú rovnakú množinu tuple bez ohľadu na ich poradie
 - *Database*: obsahuje rovnakú množinu relácií a relácie s rovnakým názvom sa rovnajú

Dodatok k prvému semestru

Popis funkcionality tried

- *Attribute* reprezentuje základný element databázy. Obsahuje informáciu uloženú ako string. Umožňuje porovnanie s druhým atribútom a vytvorenie identickej kópie.
- *Tuple* reprezentuje jeden riadok relácie. Obsahuje atribúty, ktorých význam závisí od poradia v tuple. Umožňuje porovnanie s druhým Tuple a vytvorenie identickej kópie.
- *Relation* reprezentuje jednu tabuľku databázy. Obsahuje unikátne tuple, v ktorých atribúty s rovnakým indexom majú význam podľa významu príslušného stĺpca relácie. Umožňuje porovnanie s druhou reláciou, vytvorenie identickej kópie, materializáciu operátora, vytvorenie nezávislého operátora prislúchajúceho relácii, kontrolu duplicity tuplov a otázky na obsah.
- *Operator* je schopný iterovať cez množinu výsledkov, ktorú reprezentuje. Zachováva integritu pôvodných relácií a funguje nezávisle od ostatných operátorov na tých istých reláciách. Požaduje aby vstupné operátory nevytvárali vo výsledku duplikáty. Na iteráciu cez celú množinu výsledkov je možné použiť metódu `nonDistinctNext()` (táto metóda aj metóda `next()` sú bližšie opísané v dokumente k prvému semestru). Na iteráciu cez množinu všetkých prvých unikátnych výskytov výsledkov je možné použiť metódu `next()`. Umožňuje vytvoriť kópiu, ktorá iteruje od začiatku a resetovať operátor. Pri novom iterovaní sa zachová množina výsledkov aj poradie v ktorom sú vrátené metódami `next()` a `nonDistinctNext()`.
- *Operator Relation* je vnútorná trieda relácie. Nereprezentuje skutočný operátor RA. Je potrebný iba kvôli uniformite tried v jazyku java. *Relation* spĺňa požiadavky kladené na operátor, ktoré relácia nespĺňa. Jeho množina výsledkov je množina riadkov relácie.
- *Operator Join* reprezentuje operátor relačnej algebry join. Jeho výsledky sú riadky kartéziánskeho súčinu množín výsledkov vstupných operátorov, ktoré spĺňajú podmienku vo vstupnej *TupleTransformation*.
- *Operator Antijoin* reprezentuje operátor relačnej algebry antijoin. Jeho výsledky sú tie výsledky prvého vstupného operátora, pre ktoré neexistuje výsledok druhého vstupného operátora, s ktorým spĺňa podmienku vo vstupnej *TupleTransformation*.
- *Operator Selection_Projection* reprezentuje operátory relačnej algebry selekciu a projekciu. Jeho výsledkami sú výsledky vstupného operátora, ktoré spĺňajú selekčnú podmienku vo vstupnej *TupleTransformation* projektované pomocou transformácie v *TupleTransformation*.
- *Operator Union*, *Operator Intersection* reprezentujú zjednotenie a prienik v relačnej algebre. Realizujú tieto množinové operácie na výsledkoch vstupných operátorov.
- *TupleTransformation* je transformačná metóda pre operátory. Jediná metóda `transform()` berie ako parameter tuple a vracia tuple zmenený podľa pravidla alebo null, čo reprezentuje hodnotu false.
- *ProjectionTransformation* je transformačná metóda, ktorá dostane pri vzniku postupnosť pravdivostných hodnôt. Tieto hodnoty určujú indexy atribútov, ktoré sa majú objaviť vo výsledkoch. Ak sú vstupné tuple dlhšie ako postupnosť hodnôt, chýbajúce hodnoty sú implicitne false.
- *Condition* je transformácia, ktoré vracia nezmenený tuple ak je podmienka pravdivá, inak vracia null.