Information Processing Letters 47 (1993) 237–244
Elsevier

8 October 1993

Static filtering on stratified programs

Byeong-Mo Chang, Kwang-Moo Choe and Taisook Han

Programming Languages Laboratory, Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-dong, Yusung-ku, Taejon 305-701, South Korea

Communicated by K. Ikeda
Received 24 August 1992
Revised 22 April 1993 and 12 July 1993

Abstract

Chang, B.-M., K.-M. Choe and T. Han, Static filtering on stratified programs, Information Processing Letters 47 (1993) 237–244.

We extend *static filtering* in [8], a query optimization strategy based on seminaive evaluation on *system graphs*, so that it can efficiently handle *stratified programs* without extra overhead. The computation of static filters is formalized as a transformation so that the least fixed point of the transformation can be the static filters. The static filtering on stratified programs is shown to be complete with respect to the *iterated fixed point semantics*.

Keywords: Applicative (logic) programming; stratified programs; query optimization; static filtering; system graphs

1. Introduction

General logic programs, in which rules may have negative literals, have received much attention from many researchers [1,5,9,10]. As a useful subclass of general logic programs, *stratified programs* (*stratified databases*) was introduced which allow no recursive negation. A simple and natural semantics called *iterated fixed points* was defined for stratified programs [1,9]. When Herbrand base is finite, as in the function-free case, the iterated fixed points are finite and can be found by bottom-up computation.

Bottom-up computation may generate many facts irrelevant to a query. To prevent such facts from being generated, several methods have been proposed for stratified programs by extending the magic set method [3,6] (Note that programs after the magic sets transformation to stratified programs may not be stratified.) A preliminary technique to the magic sets method called *BPR labeling* relabels predicates in a stratified program so that the transformed can be stratified [2,3]. This method involves duplications which cause some inefficiency. It is shown in [6] that programs after the magic sets transformation or stratified programs are in the new class called *weakly stratified*. This requires a new semantic model and a new evaluation method.

In this paper, we extend *static filtering* in [8], a graph-based query optimization strategy based on *system graphs*, so that it can efficiently handle stratified programs without overhead unlike other methods. *Static filters* are computed at compile time by propagating data-independent bindings in a given query on system graphs [4,8]. We formalize the computation of static filters as a transformation so that the least fixed point of the transformation can be the static filters. Based on the

Correspondence to: B.-M. Chang, Programming Languages Laboratory, Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-dong, Yusung-ku, Taejon 305-701, South Korea.

formalism, we prove the completeness of the static filtering on stratified programs with respect to the iterated fixed point semantics.

The next section describes basic definitions and notations. Section 3 describes query evaluation of stratified programs. Section 4 gives a formalism on static filter computation. Section 5 proves the completeness of the static filtering on stratified programs. Section 6 summarizes results and give further research topics.

2. Notations and definitions

Consider well-formed formula in the first-order logic. Variables are denoted by V, X, Y and Z, terms by s and t, and atomic formula by A, B and C. A *literal* is an atom or its negation. A *clause* is a formula of the form

$$A_0 \leftarrow L_1, \ldots, L_n \quad (n \geq 0),$$

where A_0 is an atom called the head, and $L_1, \ldots, L_n$ are literals called the body. We assume function-free and range restricted clauses. If $n = 0$, then the clause is called a *fact*, otherwise a *rule*. The number of body literals of a rule α is denoted by n_α. A *general program* P is a finite set of clauses.

Dependency graph for a program P is $DG = (V, E)$ where V is the set of predicates in P, and $E = \{(p, q) \mid$ a clause in P contains q in its heads and p in its body$\}$. An edge (p, q) in the dependency graph is *negative* if $\neg p$ occurs in the body of the clause. A program P is said to be *stratified* if there are no cycles containing a negative edge in the dependency graph. Let a program P be partitioned into $P_1, \ldots, P_n$. The partition is called a *stratification* if the following two conditions hold for $i = 1, \ldots, n$:

(1) if a predicate p occurs positively in a clause in P_i then its definition is contained within $\bigcup_{j \leq i} P_j$,

(2) if a predicate p occurs negatively in a clause in P_i then its definition is contained within $\bigcup_{j < i} P_j$.

(The definition of p is the subset of P which consists of all clauses in P containing p on the head.) Each P_i is called a *stratum* of P. A predicate is said to be in P_i if its definition is in P_i. It is shown in [1] that a program has a stratification iff it is stratified.

Example 2.1. Consider the following program and query.

$\alpha: p(X, Y) \leftarrow r(X, Y).$
$\beta: p(X, Y) \leftarrow r(X, Z), p(Z, Y).$
$\gamma: q(X, Y) \leftarrow s(X, Z), \neg p(Z, Y).$
$\delta: \leftarrow q(X, a).$

A possible stratification is $\{\alpha, \beta\}$ and $\{\gamma, \delta\}$.

Let $P_1, \ldots, P_n$ be a stratification of a program P. For every $i = 1, \ldots, n$, T_i is the operator defined as follows: for every Herbrand model M and every ground atom A, $A \in T_i(M)$ iff $A \in M$, or for some clause $A_0 \leftarrow L_1, \ldots, L_n$ in P_i and some ground substitution θ, $L_1\theta, \ldots, L_m\theta$ are true in M and $A = A_0\theta$.

The sets $M_0, \ldots, M_n$ of ground atoms are defined by the equations:

$$M_0 = \emptyset,$$

$$M_i = \bigcup_{k \geq 0} T_i^k(M_{i-1}) \quad (i = 1, \ldots, n)$$

It is shown in [1] that M_n is a minimal model of P, and it does not depend on the choice of stratification. This model M_n is proposed to be the denotation M_P of P. The relation for a predicate q is $Q = \{\bar{t} \mid q(\bar{t}) \in M_P\}$, where $\bar{t}$ is a list of ground terms called a *tuple*. Let DOM be the union of the ground terms appearing in P. The relation for a negated predicate $\neg q$ is $\overline{Q} = DOM^k - Q$ where $DOM^k = DOM \times \cdots \times DOM$.

3. System graph evaluation of stratified programs

We represents rules in the *AC-notation* in [8] which is more suitable for data flow approach to query evaluation. All arguments of each literal are replaced by distinct variables associated with the predicate of the literal, and conditions between arguments in a rule are represented by *conjunctive formula* on the new variables associated with that rule. If an atom has n-ary predi-

cate q, its arguments are replaced by the variables $Q1, \ldots, Qn$ (slashed variables $Q1/i, \ldots, Qn/i$ if the atom is the ith body atom). A formula is defined as follows:

(1) Atomic formula is *true*, *false*, or $g\ \theta\ s$ where t and s are terms and θ is, $<$, or $>$.

(2) If ϕ and ϕ' are formulas, $\phi \wedge \phi'$ and $\phi \vee \phi'$ are formulas.

A formula ϕ is said to *imply* a formula ϕ', written $\phi \rightarrow \phi'$, if every substitution satisfying ϕ satisfied ϕ'. We denote the associated formula of a rule α by $Cond_\alpha$. A query Q is represented as $ans \leftarrow Q$ with a new prediction *ans*. We assume a program includes its rule-form query.

Example 3.1. Consider the rules and query in Example 2.1. They are represented in the AC-notation as

$$\alpha: p(P1, P2) \leftarrow r(R1/1, R2/1),\ P1 = R1/1,\ P2 = R2/1.$$

$$\beta: p(P1, P2) \leftarrow r(R1/1, R2/1),\ p(P1/2, P2/2),\ P1 = R1/1,\ R2/1 = P1/2,\ P2 = P2/2.$$

$$\gamma: q(Q1, Q2) \leftarrow s(S1/1, S2/1),\ \neg p(P1/2, P2/2),\ Q1 = S1/1,\ S2/1 = P1/2,\ Q2 = P2/2.$$

$$\delta: ans(Ans1) \leftarrow q(Q1/1, Q2/1),\ Ans1 = Q1/1,\ Q2/1 = a.$$

A *system graph* for a program is defined to be a quintuple for data flow evaluation. A function $pred(\alpha, i)$ is defined to be the ith predicate symbol, positive or negative, in the body of α, and $head(\alpha)$ is the predicate in the head of α. A function $Var(\alpha, i)$ is the list of all variables in the ith literal of α.

Definition 3.2. A system graph for a program P is $SG = (V_P, V_R, E_{P,R}, E_{R,P}, F)$ where

V_P and V_R are the sets of predicates and rules in P respectively,

$E_{P,R} = \{(p, \alpha)/i \mid p \in V_P,\ \alpha \in V_R,\ pred(\alpha, i) = p \text{ or } \neg p,\ 1 \leqslant i \leqslant n_\alpha\}$,

$E_{R,P} = \{(\alpha, p) \mid \alpha \in V_R,\ p \in V_P,\ head(\alpha) = p\}$,

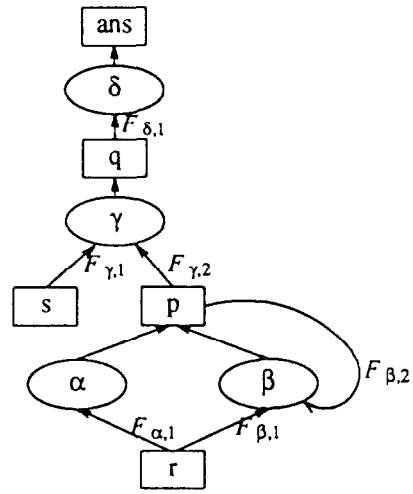

Fig. 1. System graph for Example 3.1.

F is a *filter function* that associates a formula over $Var(\alpha, i)$ with each arc $(p, \alpha)/i$ in $E_{P,R}$.

The node for a predicate is called a *pred-node* and the node for a rule is a *rule-node*. Arcs are called *ports*. The slashed pair $(p, \alpha)/i$ in $E_{P,R}$ denotes the ith input port (incoming arc) or the rule-node α from the pred-node p. A port $(p, \alpha)/i$ is *negative* if $\neg p = pred(\alpha, i)$ and *positive* if $p = pred(\alpha, i)$. A port $(p, \alpha)/i$ is denoted by α/i shortly since p is $pred(\alpha, i)$. The set of all output ports of a pred-node p is denoted by

$$E_{P,R}[p] = \{(p, \delta)/k \mid (p, \delta)/k \in E_{P,R}\}.$$

The value of a filter function F on a port $(p, \alpha)/i$, $F((p, \alpha)/i)$, is called the filter of the port and it represents a formula over $Var(\alpha, i)$ which tuples have to satisfy in order to pass the port. The filter $F((p, \alpha)/i)$ is denoted by $F_{\alpha,i}$ shortly. Figure 1 shows a system graph for the program in Example 3.1.

This algorithm shows a seminaive bottom-up evaluation of stratified programs on system graphs. A predicate in the head of a rule is called a *derived predicate*, and a predicate in a fact is a *base predicate*. Each base pred-node is assumed to have its tuples initially. We assume for easy presentation that a pred-node q with negative output ports computes $\overline{Q}$ and sends the relation

via the negative ports at a time after the relation Q is received. In implementation, it is possible that the pred-node sends Q and the rule-node computes $\overline{Q}$.

Algorithm Simplistic seminaive evaluation
Let $P_1, \ldots, P_n$ be a stratification of a program P.
for $i = 1$ to n **do**
1. Each base pred-node in P_i sends its tuples to its positive output ports.

repeat
2. Each rule-node in P_i stores new tuples from input ports into its local buffers associated with those ports, performes the join, and sends newly generated tuples to its output ports.
3. Each derived pred-node in P_i stores new tuples from its input ports, and sends them to its positive output ports.

until no change in the nodes in P_i.
4. For each pred-node q in P_i which has negative output ports, the pred-node computes and sends the relation $\overline{Q}$ through its negative output ports where Q is the relation received.

endfor.

The evaluation finds the iterated fixed point and terminates for function-free stratified programs. If the system graph has filters, a pred-node sends tuples to each output port only if they satisfy the filter of the port.

4. Static filter computation

Static filters are computed by extending the idea of pushing selection in relational algebra [8]. Selections in a query are pushed downward on system graphs along the opposite directions of ports. When a selection is pushed on a port, it is imposed on it. This procedure continues until more pushing imposes no more new selections. When this procedure ends, the disjunction of imposed selections on a port is the static filter of the port.

Example 4.1. Consider the positive rules α and β in Example 3.1, and a query $\gamma: \leftarrow p(X, a)$ which is $ans(Ans1) \leftarrow p(P1, P2), P2 = a, Ans1 = P1$ in the AC-notation. If the selection $P = a$ is pushed downward, it imposes the selection $P2 = a$ on $\gamma/1$, $R2 = a$ on $\alpha/1$, *true* (i.e. open) on $\beta/1$ and $P2 = a$ on $\beta/2$. On the other hand, if the query is $\gamma: \leftarrow p(a, X)$, pushing the selection is not simple. Pushing selection $P1 = a$ on $\gamma/1$ through α results in $R1 = a$ on $\alpha/1$, and through β results in $R1 = a$ on $\beta/1$ and *true* on $\beta/2$. The selection *true* on $\beta/2$ can be pushed further through α and β because it imposes new selection *true* on $\alpha/1$ and $\beta/1$. Therefore every static filters are *true* except the filter on $\gamma/1$ when the pushing is terminated. See [4,8] for details.

We consider the computation of static filters as simulating top-down evaluation on system graphs, assuming that every body atom in a rule is executed simultaneously when it is called. The simulation terminates when there are no more new bindings for body atoms, and the logical formula for all possible bindings of each body atom after the simulation is the static filter for that atom. We formalize the computation of static filters so that the least fixed point of a transformation *PUSH* can be the static filters (see Definition 4.6).

As a basic operation to simulate top-down evaluation, we first refine a transformation $PUSH_\alpha$ associated with a rule-node α in [8]. The filters of all input ports of α in a system graph SG are represented as the *filter vector* of α, $\vec{F}_\alpha = \langle F_{\alpha,1}, \ldots, F_{\alpha,n_\alpha} \rangle$. We denote by $\vec{F}_\alpha \leqslant \vec{F}'_\alpha$ if $F_{\alpha,i}$ for $i = 1, \ldots, n_\alpha$. We review *$\overline{X}$-consequence* in [8] for the definition of $PUSH_\alpha$. Let $\overline{X}$ be a list of some variables in a formula ϕ. A formula ϕ' is an $\overline{X}$-consequence of ϕ if $\phi \rightarrow \phi'$ and every variable in ϕ' is in $\overline{X}$. A formula ϕ' is the *most general* $\overline{X}$-consequence of ϕ if ϕ' is an $\overline{X}$-consequence of ϕ and for every other $\overline{X}$-consequence ϕ'' of ϕ, $\phi' \rightarrow \phi''$. The most general $\overline{X}$-consequence of ϕ is denoted by $\phi[\overline{X}]$, which represents the bindings of the variables of $\overline{X}$ in ϕ.

Definition 4.2. Let α be a rule-node in a system graph $SG = (V_P, V_R, E_{P,R}, E_{R,P}, F)$. Let $F_{\delta,k}$ be an output filter of the pred-node $p = head(\alpha)$. $PUSH_\alpha$ is a transformation associated with α

which maps $F_{\delta,k}$ into $\vec{F}'_{\alpha} = PUSH_{\alpha}(F_{\delta,k})$ such that for each port α/i,

$$F'_{\alpha,i} = (Cond_{\alpha} \wedge F_{\delta,k})[Var(\alpha, i)].$$

(We assume that $F'_{\alpha,i}$ is a formula over the variables $Var(\alpha, i)$ with the slashed part deleted, so every filter is a formula over the variables without slashed part. See Example 4.3.)

$PUSH_{\alpha}(F_{\delta,k})$ shows bindings of the body atoms when α is called by the calling pattern corresponding to $F_{\delta,k}$. If α is the query-node, we assume that $F_{\delta,k}$ is *true*, so $F'_{\alpha,i} = Cond_{\alpha}[Var(\alpha, i)]$. It will be shown that Definition 4.2 is valid for negative ports as well as positive ones.

Let all output filters of a pred-node p be denoted by $F(E_{P,R}[p]) = \bigvee_{(p,\delta)/k \in E_{P,R}[p]} F_{\delta,k}$. Then, $PUSH_{\alpha}$ is extended to $PUSH_{\alpha}(F(E_{P,R}[p]))$ where $p = head(\alpha)$, in which

$$F'_{\alpha,i} = (Cond_{\alpha} \wedge F(E_{P,R}[p]))[Var(\alpha, i)].$$

It is shown in [4,8] that $PUSH_{\alpha}(F(E_{P,R}[p]))$ is equivalent to $\bigvee_{(p,\delta)/k \in E_{P,R}[p]} PUSH_{\alpha}(F_{\delta}, k)$.

Example 4.3. Consider the program in Example 3.1. If $\vec{F}_{\delta} = PUSH_{\delta}(true)$, then $F_{\delta,1} = (Q2 = a)$. If $\vec{F}_{\gamma} = PUSH_{\gamma}(F_{\delta,1})$, then $F'_{\gamma,1} = true$ and $F'_{\gamma,2} = (P2 = a)$.

If $A_0\theta \leftarrow L_1\theta, \ldots, L_n\theta$ is a ground instance of a rule α, then the tuple $\bar{\mu}_i$ of $L_i\theta$, $1 \leqslant i \leqslant n$, is a *contributor* of the tuple $\bar{\mu}_0$ of $A_0\theta$ via α. When $\vec{F}_{\alpha} = PUSH_{\alpha}(F(E_{P,R}[p]))$, $F'_{\alpha,i}$ is said to be *safe* with respect to $F(E_{P,R}[p])$ if a tuple which does not satisfy $F'_{\alpha,i}$ can not be a contributor of any tuples satisfying $F(E_{P,R}[p])$.

Lemma 4.4 [8]. Let $\vec{F}_{\alpha} = PUSH_{\alpha}(F(E_{P,R}[p]))$ *where* $p = head(\alpha)$. $F'_{\alpha,i}$ *is safe with respect to* $F(E_{P,R}[p])$.

Consider $F'_{\alpha,i}$ in $\vec{F}'_{\alpha} = PUSH_{\alpha}(F(E_{P,R}[p]))$ when $pred(\alpha, i)$ is $\neg q$ with arity k. Let Q be the relation for q. Recall that the pred-node sends $\overline{Q}$ through the negative port α/i after Q is received. If we use the selection operator σ, $\sigma_{F'_{\alpha,i}}(\overline{Q})$ means the set of tuples in $\overline{Q}$ satisfying $F'_{\alpha,i}$. By Lemma 4.4, is is safe with respect to $F(E_{P,R}[p])$ that the pred-node q sends only the tuples in $\sigma_{F'_{\alpha,i}}(\overline{Q})$ through the port α/i. Moreover, the following lemma shows that the pred-node q can compute $\sigma_{F'_{\alpha,i}}(\overline{Q})$ using only the tuples in $\sigma_{F'_{\alpha,i}}(Q)$, that is, the pred-node q may receives only the tuples in $\sigma_{F'_{\alpha,i}}(Q)$. So it can be justified that $F'_{\alpha,i}$ is pushed further.

Lemma 4.5. *Let* Q *be the relation for a predicate* q *with arity* k *and* $\overline{Q}$ *be* $DOM^k - Q$. *Let* ϕ *be a formula on the variables* $Q1, \ldots, Qk$ *of* q.

$$\sigma_{\phi}(\overline{Q}) = \sigma_{\phi}(DOM^k) - \sigma_{\phi}(Q).$$

Proof. The lemma is simply proved by the well-known lemma in relational algebra that $\sigma_{\phi}(A - B) = \sigma_{\phi}(A) - \sigma_{\phi}(B)$ where A and B are relations with arity k. □

We define a transformation *PUSH* which maps all filters in a system graph into new ones simultaneously. All the filters in a system graph $SG = (V_P, V_R, E_{P,R}, E_{R,P}, F)$ are represented as the filter vector $\vec{F}$ of SG which is defined as $\vec{F} = \langle \vec{F}_{\delta_1}, \ldots, \vec{F}_{\delta m} \rangle$ where $V_R = \{\delta_1, \ldots, \delta_m\}$. The filter vector of a system graph is one representation of the filter function of that system graph, so that they can be converted to each other. The set of the filter vectors corresponding to all possible filter functions for a program P is denoted by $S(P)$.

Definition 4.6. Let $SG = (V_P, V_R, E_{P,R}, E_{R,P}, F)$ be a system graph for a program P. $PUSH : S(P) \rightarrow S(P)$ is a transformation which maps a filter vector $\vec{F}$ into $\vec{F}' = PUSH(\vec{F})$ such that

$$\vec{F}'_{\delta_1} = PUSH_{\delta_1}(F(E_{P,R}[p_1])),$$

$$\vec{F}'_{\delta_2} = PUSH_{\delta_2}(F(E_{P,R}[p_2])),$$

$$\ldots,$$

$$\vec{F}'_{\delta_m} = PUSH_{\delta_m}(F(E_{P,R}[p_m])),$$

where $V_R = \{\delta_1, \ldots, \delta_m\}$ and $p_i = head(\delta_i)$ for $i = 1, 2, \ldots, m$.

PUSH shows new bindings of the body atoms of all rules when they are called simultaneously by the calling patterns corresponding to the current filters.

Corollary 4.7. *If $\vec{F} = PUSH(\vec{F})$, then every filter $F'_{\alpha,i}$ in $\vec{F}'$ is safe with respect to $F(E_{P,R}[p])$ where $p = head(\alpha)$.*

Let $\vec{F}$ and $\vec{F}'$ be two filter vectors in $S(P)$ for a program P and $V_R = \{\delta_1, \ldots, \delta_m\}$ be the set of all rule-nodes in a system graph for P. A relation $\leqslant$ on filter vectors in $S(P)$ is defined as

$$\vec{F} \leqslant \vec{F}' \quad \text{if} \quad \vec{F}_{\delta_i} \leqslant \vec{F}'_{\delta_i} \text{ for } i = 1, 2, \ldots, m.$$

If $\vec{F} \leqslant \vec{F}'$ and $\vec{F}' \leqslant \vec{F}$, then $\vec{F}$ is said to be *equivalent* to $\vec{F}'$, written $\vec{F} \equiv \vec{F}'$. If equivalent filter vectors are considered as one, a pair $(S(P), \leqslant)$ is a partially ordered set (*poset*) *with a unique minimum element* $\vec{F}^{\min} = \langle\langle false, \ldots, false\rangle, \ldots, \langle false, \ldots, false\rangle\rangle$.

Fixed point of *PUSH* is a filter vector $\vec{F}$ such that $\vec{F} \equiv PUSH(\vec{F})$. We compute the least fixed point of *PUSH* in the poset $(S(P), \leqslant)$. Each filter in the least fixed point of *PUSH* represents all possible bindings of a body atom during the top-down simulation, so it is the static filter we want to compute.

Theorem 4.8. *If lfp(PUSH) is the least fixed point of PUSH in the poset $(S(P), \leqslant)$ for a program P with the minimal element $\vec{F}^{\min}$, then lfp(PUSH) is $PUSH^n(\vec{F}^{\min})$ for some finite n.*

Proof. The transformation *PUSH* is a monotone increasing function in the poset $(S(P), \leqslant)$ for P [4], and $(S(P), \leqslant)$ satisfies the *ascending chain condition* that there cannot be an infinite sequence of strictly ascending elements in $(S(P), \leqslant)$. Therefore, by the Tarski theorem [11], *lfp(PUSH)* is $PUSH^n(\vec{F}^{\min})$ for some finite n. □

The system graph with the static filter function for a program P is denoted by $SG^{static} = (V_P, V_R, E_{P,R}, E_{R,P}, F^{static})$ where $\vec{F}^{static} = lfp(PUSH)$ in the poset $(S(P), \leqslant)$.

Example 4.9. Consider the rules and query in Example 3.1. This example illustrates the static filter computation by the least fixed point of *PUSH*. Each iteration of *PUSH* are as follows.

Initialize: $F_{\delta,1} = false$ $F_{\gamma,1} = false$ $F_{\gamma,2} = false$
$F_{\alpha,1} = false$ $F_{\beta,1} = false$ $F_{\beta,2} = false$

1st iteration: $F_{\delta,1} = (Q2 = a)$ $F_{\gamma,1} = false$
$F_{\gamma,2} = false$ $F_{\alpha,1} = false$
$F_{\beta,1} = false$ $F_{\beta,2} = false$

2nd iteration: $F_{\delta,1} = (Q2 = a)$ $F_{\gamma,1} = true$
$F_{\gamma,2} = (P2 = a)$ $F_{\alpha,1} = false$
$F_{\beta,1} = false$ $F_{\beta,2} = false$

3rd iteration: $F_{\delta,1} = (Q2 = a)$ $F_{\gamma,1} = true$
$F_{\gamma,2} = (P2 = a)$ $F_{\alpha,1} = (R2 = a)$
$F_{\beta,1} = true$ $F_{\beta,2} = (P2 = a)$

4th iteration: the same as the 3rd iteration (the least fixed point, the static filters)

Every filter in SG^{static} satisfies the safety which is a basis of the completeness of our method.

Lemma 4.10. *Let $SG^{static} = (V_P, V_R, E_{P,R}, E_{R,P}, F^{static})$ be the system graph with the static filter function F^{static} for a program. Every filter $F^{static}_{\alpha,i}$ is safe with respect to $F^{static}(E_{P,R}[p])$ where $p = head(\alpha)$.*

Proof. By Corollary 4.7 and the definition of the fixed point. □

Consider $F^{static}_{\alpha,i}$ when $pred(\alpha, i)$ is $\neg q$. It is sufficient by Lemma 4.10 that the pred-node q sends only the tuples in $\sigma_{F^{static}_{\alpha,i}}(\overline{Q})$ via α/i. It can compute $\sigma_{F^{static}_{\alpha,i}}(\overline{Q})$ by Lemma 4.5 using only the tuples in $\sigma_{F^{static}_{\alpha,i}}(Q)$.

5. Completeness proof

We show the static filtering is complete with respect to the iterated fixed point semantics. When $P_1, \ldots, P_n$ is a stratification for a program P and $M_1, \ldots, M_n$ are the models, a tree is a *derivation tree* for $A \in M_i - M_{i-1}$ over M_{i-1} if

(1) Every vertex has a label, which is a ground instance of a literal in a rule in P_i.

(2) The label of the root is A.

(3) If a vertex with label A has the children

with labels $L_1, L_2, \ldots, L_n$, respectively, $A \leftarrow L_1, L_2, \ldots, L_n$ must be a ground instance of a rule in P_i.

(4) The label of every leaf vertex is true in M_{i-1} or a fact in P_i.

Lemma 5.1. *Let SG^{static} be the system graph with the static filter function F^{static} for a program P. Consider a stratification $P_1, \ldots, P_n$ for a program P and the models $M_1, \ldots, M_n$. Assume every fact in M_{i-1} is stored in the corresponding pred-node if it satisfies $F^{static}(E_{P,R}[q])$ where q is its predicate. Then, the evaluation of P_i on SG^{static} generates all facts $p(\overline{\mu}) \in M_i - M_{i-1}$ which satisfy $F^{static}(E_{P,R}[p])$.*

Proof. We show that for any derivation tree over M_{i-1} whose root is $p(\overline{\mu})$, if $\overline{\mu}$ satisfies $F^{static}(E_{P,R}[p])$, then it is generated by the rule-node corresponding to the rule applied at the root during the evaluation of P_i. The proof is by induction on the height of derivation tree using Lemma 4.10. See [4] for the complete proof. □

It should be noted in the lemma that the evaluation does not generate "only" the facts.

Theorem 5.2. *Let SG^{static} be the system graph with the static filter function for a program P. The evaluation on SG^{static} is complete with respect to M_P.*

Proof. Let $P_1, \ldots, P_n$ be a stratification for P and $M_1, \ldots, M_n$ be the models. The proof shows by induction on strata that the evaluation of P generates all facts $p(\overline{\mu}) \in M_P$ satisfying $F^{static}(E_{P,R}[p])$. In case p is *ans*, we assume that $F^{static}(E_{P,R}[ans]) = true$, that is, the evaluation generates all query answers in M_P.

Induction basis: Let p be a derived predicate in P_1. The evaluation of P_1 generates all facts $p(\overline{\mu}) \in M_1$ which satisfy $F^{static}(E_{P,R}[p])$ by Lemma 5.1.

Induction step: Let p be a derived predicate in P_k. Let M'_{k-1} be the set of tuples generated by the evaluation of $P_1, \ldots, P_{k-1}$. By induction hypothesis, M'_{k-1} contains all $q(\overline{\mu})$'s $\in M_{k-1}$, which satisfy $F^{static}(E_{P,R}[q])$. Therefore, the evaluation of P_k generates all facts $p(\overline{\mu}) \in M_k - M_{k-1}$ which satisfy $F^{static}(E_{P,R}[p])$ by Lemma 5.1. □

6. Conclusion

We have shown that the static filtering efficiently handles stratified programs without overhead. The computation of static filters is formalized as a transformation so that the least fixed point of the transformation can be the static filters. The completeness of the static filtering on stratified programs is proved with respect to the iterated fixed point semantics.

Static filtering is suitable for stratified programs since static filters do not change during evaluation. If filters change during evaluation like dynamic filter [7], stratified programs might not be handled in such an efficient way. It is interesting to find a new evaluation method for dynamic filtering without sacrificing the original semantics. We are currently interested in improving filtering using abstract interpretation.

Acknowledgements

We are grateful to M. Kifer at SUNY, Stony Brook and I. Balbin at University of Melbourne for their help in preparing this paper. We are grateful to anonymous referees for valuable comments.

References

[1] K.R. Apt, H. Blair and A. Walker, Towards a theory of declarative knowledge, in: *Foundation of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988) 89–148.

[2] I. Balbin, Efficient bottom-up computations on recursively defined deductive databases, Tech. Rept. 89/17, Dept. of Computer Science, University of Melbourne, 1989.

[3] I. Balbin, K. Meenakshi and K. Ramamohanarao, A query independent method for magic set computation on stratified databases, in: *Proc. Internat. Conf. on Fifth Generation Computer Systems* (1988) 711–718.

[4] B.-M. Chang, K.-M. Choe and T. Han, Fixed point computation of static filters for stratified programs, Tech.

Rept. CS-TR-92-72, Dept. of Computer Science, KAIST, 1992.

[5] K.L. Clark, Negation as failure, in: *Logic and Database* (Plenum Press, New York, 1978) 293–322.

[6] J.-M. Kerisit and J.M. Pugin, Efficient query answering on stratified databases, in: *Proc. Internat. Conf. on Fifth Generation Computer Systems* (1988) 719–726.

[7] M. Kifer and E.L. Lozinskii, A framework for an efficient implementation of deductive database systems, in: *Proc. 6th Advanced Database Symp.* (1986) 109–116.

[8] M. Kifer and E.L. Lozinskii, On compile time query optimization in deductive databased by means of static filtering, *ACM Trans. Database Systems* **15** (3) (1990) 385–426.

[9] V. Lifschitz, On the declarative semantics of logic programs with negation, in: *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988) 177–192.

[10] J.C. Shepherdson, Negation as failure: a computation of Clark's completed data base and Reiter's closed world assumption, *J. Logic Programming* **2** (1984) 51–81.

[11] A. Tarski, A lattice theoretical fixed point theorem and its applications, *Pacific J. Math.* **5** (1955) 289–321.