Abstract Filters: Improving Bottom-up Execution of Logic Programs by Two-phase Abstract Interpretation

Byeong-Mo Chang Kwang-Moo Choe
Korea Advanced Institute of Science and Technology

Roberto Giacobazzi *
École Polytechnique

(Extended Abstract)

Abstract

Bottom-up evaluation of logic programs may be inefficient as it may generate many irrelevant facts to a given query. The filtering strategy prevents possible irrelevant facts from being processed in the bottom-up evaluation on system graphs. This paper defines the filtered bottom-up evaluation formally, and provides a new filter called *abstract filter* which is computed by *two-phase abstract interpretation* at compile time. The abstract filter is defined on a generic abstract domain. This framework is specialized on the depth k abstract domain of Sato and Tamaki. The abstract filter is shown to be at least as powerful as the static filter on the depth k abstract domain. Some performance results are provided.

Keywords: *abstract interpretation, bottom-up evaluation, filtering, logic programming.*

1 Introduction

Because of a number of advantages, the bottom-up evaluation of logic programs has recently attracted much attention in logic programming and deductive database [8, 9, 10]. It guarantees logical completeness which strict top-down approaches sacrifice in favor of implementational efficiency as in Prolog. In particular, if the number of all possible facts is finite, the bottom-up evaluation terminates while strict top-down one like Prolog may not. In addition, the bottom-up strategy is free from nonlogical features like backtracking. However, bottom-up query evaluation may be inefficient since it may generate many irrelevant facts to a given query. In order to overcome this drawback, a number of optimization strategies has been proposed, in particular for seminaive bottom-up evaluation [1, 2, 7, 8, 10]. For example, the program transformation strategy, like Magic Sets in [1], transforms original programs so that the transformed programs can compute call patterns (which are called magic sets in Magic Sets) and use them to reduce the generation of irrelevant facts during the bottom-up query evaluation. Another approach has been introduced to solve these problems: the filtering strategy. It prevents irrelevant facts from being generated at evaluation time by applying *filters* over the bottom-up evaluation on system graphs [7, 8]. A filter is a kind of selection attached on an arc of a system graph, which is able to restrict the data flow through the arc. The basic proposal is *static filtering* [7, 8]. It restricts data flow during evaluation by using *static filters*, which are computed at compile-time by propagating data-independent bindings (selections) in the query and rules. Static filters, however, have their own limit since they are always computed using only *data-independent* bindings. This drawback can be cured by *dynamic filtering*, which however incurs runtime overhead because *dynamic filters* are always computed during program evaluation [4, 7].

In this paper we introduce a new family of filters called *abstract filters*. They are obtained at compile time by abstract interpretation. The definition of abstract filter is presented for a generic abstract domain and provides a suitable generalization of the (static) filter notion. Because abstract filters are independent of the specific data abstraction, different abstract domains may provide different abstract filters at different layer of abstraction. However, because of the range of applications of abstract filters, we believe that *type* abstractions (e.g. depth k or *type graphs*) may provide more appropriate abstract filters. The precision of an abstract filter depends on the chosen abstract domain and on the way the analysis is performed. As a specialization, we compute abstract filters using a *two-phase abstract interpretation*. The two-phase abstract interpretation consists of a bottom-up phase followed by a top-down one, and it provides an approximation of success patterns of clauses which are *relevant* to a given query. We will specialize this framework by considering the depth k abstraction [11] as abstraction function. The abstract filter constructed from the two-phase analysis prevents facts not generated by the analysis (i.e. facts which are neither success patterns nor relevant to the query) from being processed in the bottom-up evaluation. From the theoretical point of view, the abstract filter is shown to be at least as powerful as the static filter on the depth k abstract domain. assumption. Both the two-phase abstract analyzer and the bottom-up evaluator have been implemented, providing some experimental results which further on compare the performances of the static and abstract filters. In particular we observed that: (1) the abstract filter does not incur runtime overhead except filter passing, while Magic Sets does, due to the computation of magic sets during evaluation, (2) due to the goal independence of the bottom-up abstract interpretation, the result of the bottom-up phase can be used for any query to the same program. This is a good property in program compilation, since many different queries are usually asked over the same program.

*The work of R. Giacobazzi has been partly supported by the EEC *Human Capital and Mobility* individual grant: "Semantic Definitions, Abstract Interpretation and Constraint Reasoning".

2 Logic programming

Let Π be a (finite) set of predicate symbols, Σ a set of function symbols, and Var a denumerable set of variables. n-ary predicates or functions are denoted f/n. The set of all terms constructed from Σ and Var is denoted $Term$. An *atom* is a syntactic object of the form $p(t_1,\ldots,t_n)$ where $p/n \in \Pi$ and $t_1,\ldots,t_n \in Term$. We denote $Atoms$ the set of all atoms. We also associate a tuple $\langle t_1,...,t_n\rangle$ with each atom $p(t_1,...,t_n)$. A *clause* has the form $h \leftarrow b_1,\ldots,b_n$ $(n \geq 0)$ where h is a *head* atom and b_i's are *body* atoms. The set of clauses constructed from elements of $Atoms$ is denoted $Clause$. If $n = 0$ then the clause is simply h and called an *unit clause* or *fact* . Otherwise it is called a *rule* . We consider atoms and facts as equivalent notions. The set of variables occurring in a syntactic object t (e.g. a term, atom, clause *etc.*) is denoted by $var(t)$. Sequences of syntactic objects b are often denoted $\bar{B}$. We abuse by treat sequences as sets. A rule is usually denoted by δ. A function $pred(\delta, i)$ is defined to be the i-th predicate symbol in the body of δ, and a function $head(\delta)$ is the predicate of the head of δ. In this case i is an *index* for δ. A *query* is a sequence of atoms, sometimes denoted $\leftarrow G$. Sometimes we abuse by calling query a clause of the form $ans \leftarrow G$ where $var(ans) = var(G)$. A *logic program* P is a finite set of clauses. A *substitution* is a mapping from Var to $Term$ which acts as the identity almost everywhere. The set of idempotent substitutions is denoted by Sub. Following tradition, the application of a substitution θ to an object t will be written $t\theta$ rather than $\theta(t)$. We fix a partial function mgu which maps a pair of syntactic objects to an idempotent most general unifier of the objects if it exists. The relation $\sim$ is defined for any syntactic object as $s \sim t$ iff $\exists\vartheta,\sigma$: $s\vartheta = t \wedge t\sigma = s$.

Let $B_P = Atoms/\sim$ where $Atoms/_\sim$ is the set of equivalence classes on $Atoms$ with respect to $\sim$. The (bottom-up) semantics of a logic program P, characterizing computed answer substitutions is defined in [6] in terms of an immediate consequence operator T_P on the domain of interpretations $\wp(B_P)$. In the following we omit equivalence classes in syntactic objects and for a syntactic object s and an interpretation I we denote by $\langle a_1,\ldots,a_n\rangle \ll_s I$ that $a_1,\ldots,a_n$ are representatives of elements of I renamed apart from s and from each other.

$$T_P(I) = \left\{ h\vartheta \;\middle|\; \begin{array}{l} c \;=\; h \leftarrow \bar{B} \in P,\; \bar{B}' \ll_c I \\ \vartheta = mgu(\bar{B}, \bar{B}') \end{array} \right\}.$$

The semantics of a program P is $lfp(T_P)$. It is known ([6]) that if G is a goal, ϑ is the substitution computed in a (SLD) refutation ([9]) of G in P iff there exists $\bar{B} \ll_G lfp(T_P)$ such that $\sigma = mgu(G, \bar{B})$ and $G\vartheta \sim G\sigma$.

3 Filtered bottom-up evaluation

3.1 Formalism

A *filter* is usually a logical formula mapping tuples of terms into the truth values. It is assumed that a filter is defined for every body atom of every rule. The value of a filter function F on the i-th body atom of a rule δ, denoted by $F(\delta, i)$, is called the *filter* of that body atom. A filter $F(\delta, i)$ can also be denoted by $F_{\delta,i}$ for short. If (the tuple associated with) an atom b *satisfies* a filter $F_{\delta,i}$, we write $b \in F_{\delta,i}$.

We formally define filtered bottom-up evaluations by modifying the T_P operator in [6].

Definition 1 *Let P be a program and F be a filter. We define $T_P^F : \wp(B_P) \rightarrow \wp(B_P)$ such that*

$$T_P^F(I) = \left\{ h\vartheta \;\middle|\; \begin{array}{l} c \;=\; h \leftarrow b_1,...,b_n \in P \\ \langle b'_1,...,b'_n\rangle \ll_c I, b'_i \in F_{C,i} (1 \leq i \leq n) \\ \vartheta = mgu(\langle b_1,\ldots,b_n\rangle, \langle b'_1,\ldots,b'_n\rangle) \end{array} \right\}$$

The filtered bottom-up evaluation computes the least fixpoint $lfp(T_P^F)$ (the filtered fixpoint semantics), which is a subset of $lfp(T_P)$. Indeed T_P^F instantiated with the always *true* filter corresponds to T_P. Notice that T_P^F is continuous.

Definition 2 *Let P be a program and $ans \leftarrow G$ be a query. For a filter F, $lfp(T_P^F)$ is* equivalent *to $lfp(T_P)$ with respect the query iff $T^F_{\{ans\leftarrow G\}}(lfp(T_P^F)) = T_{\{ans\leftarrow G\}}(lfp(T_P))$. In this case we say that $lfp(T_P^F)$ is* complete *for P and G.*

Equivalence means that the answers for the query are the same in both the fixpoint semantics.

3.2 Implementation on system graphs

The filtering on system graphs has been introduced as a tool for the filtered bottom-up evaluation [4, 7, 8]. It computes the least fixpoint $lfp(T_P^F)$ in a seminaive manner. Let $Arg\_Var(\delta, i)$ denote the argument variables of the i-th body atom of a rule δ. An argument variable is a variable ranging on $Term$, which corresponds to an argument position of an atom. If an atom has an m-ary predicate p, its argument variables are $P1,\ldots,Pm$, named after the predicate symbol. Let n_δ denote the number of body atoms in a rule δ.

Definition 3 *A system graph for a program P is $SG_P = (V_p, V_r, E_{p,r}, E_{r,p}, F)$ where V_p and V_r are the sets of predicates and rules in P respectively, $E_{p,r} = \{(p,\delta)/i | p \in V_p, \delta \in V_r, p = pred(\delta, i), 1 \leq i \leq n_\delta\}$, $E_{r,p} = \{(\delta, p) | \delta \in V_r, p \in V_p, p = head(\delta)\}$ and F is a filter function that associates a filter over $Arg\_Var(\delta, i)$ with every arc $(p,\delta)/i \in E_{p,r}$.*

The slashed pair $(p,\delta)/i$ in $E_{p,r}$ denotes the i-th input port (incoming arc) of a rule-node δ from the pred-node p, which is also an output port(outgoing arc) of the pred-node p. The value of a filter function F on a port $(p,\delta)/i$, $F((p,\delta)/i)$, is called the *filter* of that port and it represents a logical formula over $Arg\_Var(\delta, i)$. A filter $F((p,\delta)/i)$ is denoted by $F_{\delta,i}$ for short. We determine a filter function by computing filters for every input port of every rule-node. The filter of each port in $E_{p,r}$ is used to restrict data flow through the port. The system graph with a filter function F for a program P is denoted by SG_P^F.

The simplistic bottom-up evaluation evaluates a query in terms of bottom-up data flow on system graphs [4, 8]. In the following a pred-node is called a *base pred-node* if it does not have its input ports, and called a *derived pred-node* otherwise. Each base pred-node in P has its facts initially.

Algorithm the simplistic bottom-up evaluation on system graphs SG_P:

(1) initially each base pred-node in P sends its tuples to its output ports,

repeat

(2) each rule-node stores new tuples from input ports into its local buffers associated with the ports, performs the unification and sends the newly generated facts via its output ports;

(3) each derived pred-node in P stores new tuples from its input ports, and sends them to its output ports.

until no new changes are produced on nodes in SG_P.

The algorithm can be easily modified so that the filter of each port in $E_{p,r}$ restricts data flow through the port, by not passing facts or tuples if they do not satisfy the filter. The evaluation on SG_P^F computes $lfp(T_P^F)$ in a seminaive manner. As an example, we briefly review the static filter computation in [8]. A level k subterm ([11]) is defined as follows: (1) For a given term t, t is a level 0 subterm of t. (2) If a level k subterm of t is $f(t_1, \ldots, t_n)$, then t_i is a level $k+1$ subterm of t. The *depth k abstract term* of a term t is the term t' which is obtained by substituting every level k subterm of t with a *fresh variable. The k-signature* of a term is defined in [8] which extends the depth k abstraction in [11]. The basic idea of the static filter computation is to simulate top-down execution on the system graph. It propagates bindings in the query backward on the system graph until no more new calling (binding) patterns are found, with every calling pattern being maintained in its k-signature form [8]. The abstraction by k-signature guarantees the termination of the algorithm.

Example 1 *Consider the following program:*

$\leftarrow p(X, f(f(b)))$.
$\delta_1 : p(X, f(Y)) \leftarrow r(X, Z), p(Z, Y)$.
$\delta_2 : p(X, Y) \leftarrow r(X, Y)$.

The selection $P2 = f(f(b))$ in the query imposes a selection $P2 = f(b)$ on $(p, \delta_1)/2$ if pushed backward via the rule δ_1. The new selection $P2 = f(b)$ imposes a selection $P2 = b$ if pushed backward via δ_1. The selections $P2 = f(f(b)), P2 = f(b)$ and $P2 = b$ can be pushed similarly via the rule δ_2. The static filters computed are $SF_{\delta_1,2} = (P2 = f(b) \vee P2 = b)$ and $SF_{\delta_2,1} = (R2 = f(f(b)) \vee R2 = f(b) \vee R2 = b)$.

Theorem 1 *Let SF be the static filter for a program P, and T_P^{SF} be the operator which is defined by replacing F with SF in Definition 1. $lfp(T_P^{SF})$ is equivalent to $lfp(T_P)$ with respect to any query.*

4 Abstract filters

Abstract interpretation can be used to define a general notion of abstract filter. An abstract interpretation of a program is an approximation of its standard semantics on an abstract (usually simpler) domain ([5]) so as to statically derive information about the runtime behavior of a program. The overall abstract interpretation methodology can be summarized as follows: (1) a concrete (collecting) semantics is selected which correctly characterizes the observable property of interest; (2) an abstract semantics is defined by providing approximated descriptions of semantic objects and operators. We assume that a concrete interpretation for a (logic) program P can be defined in terms of a monotone operator $E_P : E \rightarrow E$ on a concrete domain $(E, \sqsubseteq)$. An abstract interpretation $((E, \sqsubseteq), E_P, (D, \preceq), D_P, \alpha, \gamma)$ consists of a complete lattice $(D, \preceq)$, a monotone operator $D_P : D \rightarrow D$, a pair of monotone functions $\alpha : E \rightarrow D$ and $\gamma : D \rightarrow E$ (abstraction and concretization) such that
(1) $((E, \sqsubseteq), \alpha, (D, \preceq), \gamma)$ is a *Galois insertion*, i.e. for all $d \in D$, $e \in E$: $d = \alpha(\gamma(d))$ and $e \sqsubseteq \gamma(\alpha(e))$, and
(2) $E_P(\gamma(d)) \sqsubseteq \gamma(D_P(d))$ (correctness), specifying that D_P faithfully mimics E_P.
¿From the definition it follows that $lfp(E_P) \sqsubseteq \gamma(lfp(D_P))$ (see [5]).

We follow [3] by defining abstract interpretations as abstractions of concrete interpretations (i.e. subsets of B_P). Let P be a program and $((\wp(B_P), \subseteq), \alpha, (\mathcal{H}, \sqsubseteq), \gamma)$ be a Galois insertion. In this case, $(\mathcal{H}, \sqsubseteq)$ is a domain of abstract interpretations for P. In the following we abuse by calling *abstract interpretation* both such denotations and the well known static program analysis technique. Each abstract interpretation I^A represents the set of (concrete) atoms (i.e. a concrete interpretation): $\gamma(I^A)$. Moreover, with each concrete interpretation I we can always associate an abstract interpretation $\alpha(I)$.

Definition 4 *Let P be a program, δ be a rule and i be an index for δ. The abstract filter $AF_{\delta,i}$ is defined to be an abstract interpretation I^A for P, such that $\gamma(I^A)$ contains only atoms with predicate $pred(\delta, i)$.*

It is easy to see that static filters are abstract filters with k-signature abstraction. Let I^A be an abstract interpretation. We denote by $I^A[p]$ the set of all abstract denotations in I^A representing the predicate p (i.e. the largest abstract interpretation J^A such that: $J^A \sqsubseteq I^A[p]$ and $\gamma(J^A)$ contains only atoms with predicate p).

Definition 5 *A (concrete) atom A is said to* satisfy *the abstract filter $AF_{\delta,i}$, denoted by $A \in AF_{\delta,i}$, iff $A \in \gamma(AF_{\delta,i})$.*

The notions of completeness and equivalence can be extended from filters to abstract filters in the obvious way. The operator T_P^{AF} and the system graph evaluation can be formally defined by replacing the filter F with an abstract filter AF in Definition 1. The precision of abstract filters depends on the abstract domains and program analyses. Let $((\wp(B_P), \subseteq), \alpha, (\mathcal{H}, \sqsubseteq), \gamma)$ and $((\wp(B_P), \subseteq), \alpha', (\mathcal{H}', \sqsubseteq'), \gamma')$ be Galois insertions. We define an ordering relation among abstract filters such that given any two abstract filters AF_1 and AF_2 on $\mathcal{H}$ and $\mathcal{H}'$ respectively: $AF \preceq AF'$ iff for each rule δ and index i: $\gamma(AF_{\delta,i}) \subseteq \gamma'(AF'_{\delta,i})$. In this case we have that for any interpretation I: $T_P^{AF}(I) \subseteq T_P^{AF'}(I)$. Thus, more precise is the analysis, and more filtering is performed at evaluation time. More precise AF can be obtained either by a more refined abstract interpretation technique or by a more precise (less abstract) abstract domain.

5 Two-phase abstract interpretation

In this section we will consider abstract filters on the simple depth k type abstraction (see [11]) and show that, in relation with static filters, the relatively more abstract domain of depth k abstractions can provide relevant filters once a more precise (two-phase) abstract interpretation process is considered.

5.1 Bottom-up phase

In this section we extend the bottom-up abstract interpretation framework in [3] in order to include clauses as well as atoms as semantic objects. Let $\rho(t)$ be a function which maps a term t to its depth k abstract term. We define the *abstract base* B_P^A as $Atoms^A/\!\sim$ where $Atoms^A = \{p(\bar{t}^A) | p \in \Pi, \bar{t}^A \subseteq \rho(Term)\}$. The abstraction function $\alpha_a : B_P \rightarrow B_P^A$ is defined by $\alpha_a(p(t_1, \ldots, t_n)) = [p(\rho(t_1), \ldots, \rho(t_n))]_\sim$. The abstraction function is lifted on interpretations by defining $\alpha_a : \wp(B_P) \rightarrow \wp(B_P^A)$ such that $\alpha_a(I) = \{\alpha_a(a) | a \in I\}$. The corresponding concretization function $\gamma_a : \wp(B_P^A) \rightarrow \wp(B_P)$ is $\gamma_a(I^A) = \{a' \in B_P | a^A \in I^A, \alpha_a(a') = a^A\}$, and provides a Galois insertion. An *abstract substitution* is a mapping from a given finite set of variables $V \subseteq Var$ into $\rho(Term)$ which acts as the identity almost everywhere. Likewise we define abstraction and concretization on substitutions: α_S and γ_S. The abstract operator T_P^A on $\wp(B_P^A)$ can now be defined for depth k approximations of T_P:

$$T_P^A(I^A) = \left\{ \alpha_a(h\vartheta^A) \;\middle|\; \begin{array}{l} c = h \leftarrow \bar{B} \in P,\ \bar{B}^A \ll_c I^A \\ \vartheta^A = mgu^A(\bar{B}, \bar{B}^A) \end{array} \right\}$$

where $mgu^A = \alpha_S \circ mgu$. It is shown in [3] that T_P^A is correct i.e. $\gamma_a(lfp(T_P^A)) \supseteq lfp(T_P)$.
The bottom-up phase is defined by extending the framework in [3] to deal with clauses as semantic objects. Let $\mathcal{C}$ be the set of (equivalence classes of) clauses up to renaming:

i.e. $Clause/_\sim$. The set of *abstract clauses* is defined by $\mathcal{C}^A = \{\alpha_c(C) | C \in \mathcal{C}\}$ where for any clause $h \leftarrow b_1, \ldots, b_n$: $\alpha_c(h \leftarrow b_1, \ldots, b_n)$ is $[\alpha_a(h) \leftarrow \alpha_a(b_1), \ldots, \alpha_a(b_n)]_\sim$ The *extended bottom-up abstract interpretation* is:

$$((\wp(B_P) \times \wp(\mathcal{C}), \subseteq), \mathcal{U}_P, (\wp(B_P^A) \times \wp(\mathcal{C}^A), \subseteq), \mathcal{U}_P^A, \alpha, \gamma).$$

where we extend T_P by introducing a concrete operator $\mathcal{U}_P : \wp(B_P) \times \wp(\mathcal{C}) \rightarrow \wp(B_P) \times \wp(\mathcal{C})$ such that $\mathcal{U}_P(I, J) = (T_P(I), J')$ where

$$J' = \left\{ c\vartheta \;\middle|\; \begin{array}{l} c = h \leftarrow \bar{B} \in P,\ \bar{B}' \ll_c I \\ \vartheta = mgu(\bar{B}, \bar{B}') \end{array} \right\}$$

The concrete collecting semantics of a logic program P is determined by $lfp(\mathcal{U}_P)$. Its first part is denoted by $lfp_a(\mathcal{U}_P)$ and its second part by $lfp_c(\mathcal{U}_P)$. The abstraction function $\alpha : \wp(B_P) \times \wp(\mathcal{C}) \rightarrow \wp(B_P^A) \times \wp(\mathcal{C}^A)$ is defined by $\alpha(I, J) = (\alpha_a(I), \alpha_c(J))$. The concretization function $\gamma : \wp(B_P^A) \times \wp(\mathcal{C}^A) \rightarrow \wp(B_P) \times \wp(\mathcal{C})$ is $\gamma(I^A, J^A) = (\gamma_a(I^A), \gamma_c(J^A))$ where $\gamma_c(J^A) = \{c | c^A \in J^A, \alpha_c(c) = c^A\}$. They form a Galois insertion. The abstract operator $\mathcal{U}_P^A : \wp(B_P^A) \times \wp(\mathcal{C}^A) \rightarrow \wp(B_P^A) \times \wp(\mathcal{C}^A)$ is defined by extending T_P^A in a similar way: $\mathcal{U}_P^A(I^A, J^A) = (T_P^A(I^A), J^{A'})$ where

$$J^{A'} = \left\{ \alpha_c(c\vartheta^A) \;\middle|\; \begin{array}{l} c = h \leftarrow \bar{B} \in P, \bar{B}^A \ll_c I^A, \\ \vartheta^A = mgu^A(\bar{B}, \bar{B}^A) \end{array} \right\}$$

The abstract semantics is defined as the least fixed point $lfp(\mathcal{U}_P^A)$. The first part of $lfp(\mathcal{U}_P^A)$ is denoted by $lfp_a(\mathcal{U}_P^A)$ which is the same as $lfp(T_P^A)$, while the second part is $lfp_c(\mathcal{U}_P^A)$ which contains all abstract clauses whose body goals are in $lfp(T_P^A)$. $lfp_c(\mathcal{U}_P^A)$ provides the approximated success patterns of each (program) clause. Because the abstract domain is finite, $lfp(\mathcal{U}_P^A)$ terminates. Moreover, the bottom-up phase is correct, i.e. $\gamma(lfp(\mathcal{U}_P^A)) \supseteq lfp(\mathcal{U}_P)$.

5.2 Top-down phase

The bottom-up phase computes approximated success patterns without regard to the query. Therefore, many of them are not relevant to the given query. The following top-down analysis is based on the result of the bottom-up phase and it is designed to collect a subset of $lfp_c(\mathcal{U}_P^A)$ which consists of abstract success patterns of program clauses relevant to the query. An abstract clause $h^A \leftarrow \bar{B}^A \in lfp(\mathcal{U}_P^A)$ is *relevant* to a query Q iff h^A is unifiable with some atoms in Q, or h^A is unifiable with some body atom of some abstract clause relevant to Q.

The top-down phase is an abstract interpretation:

$$((\wp(lfp_c(\mathcal{U}_P)), \subseteq), \mathcal{D}_P, (\wp(lfp_c(\mathcal{U}_P^A)), \subseteq), \mathcal{D}_P^A, \alpha_c, \gamma_c)$$

where the concrete operator $\mathcal{D}_P : \wp(lfp_c(\mathcal{U}_P)) \rightarrow \wp(lfp_c(\mathcal{U}_P))$ is defined for a program P and query Q and $J \in \wp(lfp_c(\mathcal{U}_P))$:

$$\mathcal{D}_P(J) = \left\{ c \;\middle|\; \begin{array}{l} c = h \leftarrow \bar{B} \in lfp_c(\mathcal{U}_P) \\ h' \in body\_atoms(\{Q\} \cup J), \exists mgu(h, h') \end{array} \right\}$$

Here $body\_atoms(J)$ is the set of body atoms of clauses in J. The concrete collecting semantics for the top-down phase is determined by $lfp(\mathcal{D}_P)$. Analogously, the abstract operator $\mathcal{D}_P^A : \wp(lfp_c(\mathcal{U}_P^A)) \rightarrow \wp(lfp_c(\mathcal{U}_P^A))$ is defined for the top-down phase:

$$\mathcal{D}_P^A(J^A) = \left\{ c^A \;\middle|\; \begin{array}{l} c^A = h^A \leftarrow \bar{B}^A \in lfp_c(\mathcal{U}_P^A) \\ h^{A'} \in body\_atoms(\{Q\} \cup J^A) \\ \exists mgu^A(h^A, h^{A'}) \end{array} \right\}$$

It is straightforward to prove that both $\mathcal{D}_P$ and $\mathcal{D}_P^A$ are monotonic functions and D_P^A is correct (i.e. $\gamma_c(lfp(\mathcal{D}_P^A)) \supseteq lfp(\mathcal{D}_P)$). Moreover $\wp(B_P^A)$ is a finite lattice, thus the least fixpoint computation $lfp(\mathcal{D}_P^A)$ terminates. It contains all the abstract clauses in $lfp(\mathcal{U}_P^A)$ which are relevant to the query.

Example 2 *Let P be the following logic program*

$\delta_1 : path(X, [X|P]) \leftarrow arc(X, N), path(N, P).$
$\delta_2 : path(X, [X]) \leftarrow final(X).$
$final(f).$
$arc(a, b).\ arc(a, c).\ arc(c, b).\ arc(c, d).\ arc(d, f).$
$arc(e, c).\ arc(e, f).\ arc(g, e).$

Consider a query of the form $p(a, Z)$. The result of the two-phase abstract interpretation (i.e. $lfp(D_P^A)$) is:

$$\left\{ \begin{array}{l} path(a, [a|\_]) \leftarrow arc(a, c), path(c, [c|\_]).,\\ path(c, [c|\_]) \leftarrow arc(c, d), path(d, [d|\_]).,\\ path(d, [f|\_]) \leftarrow arc(d, f), path(f, [f]).,\\ path(f, [f]) \leftarrow final(f).,\\ final(f).,\ arc(a, c).,\ arc(c, d).,\ arc(d, f). \end{array} \right\}$$

6 Depth k abstract filters

The bottom-up phase provides the approximated success patterns for every clause. The atoms that do not participate in the approximated success patterns will not be considered in the query evaluation because they will not participate in some real success patterns. In addition, the top-down phase finds only the approximated success patterns relevant to a given query. The atoms that do not participate in the approximated success patterns relevant to the query will not be considered in the real query evaluation, as they are not relevant to a given query even though it can participate in some real success patterns. By correctness the abstract interpretation $lfp(\mathcal{D}_P^A)$ constitutes a suitable base for the definition of a corresponding complete abstract filter. In the following, the set of all (abstract) instances of a rule δ in $lfp(\mathcal{D}_P^A)$ is denoted by $lfp(\mathcal{D}_P^A)[\delta]$, while the set of all (abstract) instances of the i-th body atom in $lfp(\mathcal{D}_P^A)[\delta]$ is denoted by $lfp(\mathcal{D}_P^A)[\delta][i]$. Notice that if the i-th body atom of δ is A, then $lfp(\mathcal{D}_P^A)[\delta][i]$ contains (abstract) instances of A.

Definition 6 *Let P be a program, δ be a rule and i be an index for δ. The abstract filter $AF_{\delta,i}$ is defined by $AF_{\delta,i} = lfp(\mathcal{D}_P^A)[\delta][i]$.*

In case of input ports for the given query Q, we can define abstract filters by using $J_{query}^A(Q)$, where

$$J_{query}^A(Q) = \left\{ \alpha_c(Q\vartheta^A) \;\middle|\; \begin{array}{l} \bar{B}^A \ll_Q lfp(T_P^A), \\ \vartheta^A = mgu^A(Q, \bar{B}^A) \end{array} \right\}$$

Example 3 *The abstract filters with depth 2 for the program in Example 2 are computed using $lfp(\mathcal{D}_P^A)$: $AF_{\delta_1,1} = \{arc(a, c), arc(c, d), arc(d, f)\}$ and $AF_{\delta_1,2} = \{path(f, [f]), path(d, [d|\_]), path(c, [c|\_])\}$. In the system graph evaluation with the abstract filters, the rule δ_1 generates 3 facts, namely: $path(d, [d, f])$, $path(c, [c, d, f])$ and $path(a, [a, c, d, f])$. On the other hand, the static filters can not give any restrictions to the input ports of δ_1, so the evaluation with them degenerates to the naive evaluation, in which case the rule δ_1 generates 7 facts.*

In general, the difference between abstract filters and static filters is due to the fact that the bottom-up phase provides an approximation of all success patterns by considering

data-dependent bindings in their (depth k) abstracted form, while the static filter computation does not consider any data-dependent binding. This is a typical example showing the power of the bottom-up analysis and the difference between the static filter and abstract filter.
Let T_P^{AF} be the operator defined as in Definition 1. The completeness of the abstract filter with respect to a given query is a consequence of the following theorem.

Theorem 2 *Let P be a program and Q be a query. Let AF be computed as above for P and Q. Then, $lfp(T_P^{AF})$ is equivalent to $lfp(T_P)$ with respect to the query Q.*

Strictly speaking, the abstract filter can not be compared with the static filter since static filters are computed by the depth k signature, which extends the depth k abstraction. We can associate with the notion of static filter an abstract filter based on depth k signatures instead of the more abstract depth k abstraction. A simpler way to show that abstract filters are at least as powerful as the static filters is under the assumptions that static filters and abstract filters are both computed using the same (more simple) depth k abstraction. We denote a static filter on depth k abstraction as SF^k.

Theorem 3 *$lfp(T_P^{AF}) \subseteq lfp(T_P^{SF^k})$ for any program P.*

7 Experiments

The two-phase abstract interpreter has been implemented based on the simple depth-k abstract domain on SPARC station-2. It provides an abstract filter for each input port of every rule-node. The bottom-up evaluator reads (abstract or static) filters and finds solutions by the system graph evaluation with the filters. The abstract filtering has been compared with the Kifer's static filtering [8].
We use two measures: the number of generated tuples κ and the number of filtered tuples during evaluation ρ, which is a measure of the total traffic through ports. The performance of the evaluation is largely dominated by them. The following performance results for the abstraction of depth 2, 3 and 4 show effects of the depth k abstraction as k becomes higher. In the following table, in the cases of *rev* and *ackerman* programs, we represent the numbers until the first solution is found. The abstract filtering generates only about 30 percentage of the tuples generated by the static filtering for the program *parser* in [12], which also optimizes the naive system graph evaluation considerably. For the program *rev*, both the static filtering and abstract filtering make almost no improvement. It is, however, a matter of course since every atom generated during the evaluation is necessary for answering a given query. The program *path/sd* is a path program with a single destination node (like the program in Example 2) which contains a graph $G = (V, E)$ such that $|V| = 33$ and $|E| = 44$. The abstract filtering optimizes the bottom-up evaluation considerably for the program, while the static filtering falls into the naive evaluation. For the program *path/md* with multiple destinations in [12], the abstract filtering generates about half of the tuples generated by the static filtering, which also optimizes the naive system graph evaluation considerably. We can see through the table that the abstract filtering is useful in optimizing nondeterministic program rather than deterministic programs like the program *rev*. Due to the page limit, we cannot include the tables for the depth 2 and 4. However, notice that the depth 4 abstraction make almost no further improvement (only for *parser*, the abstract filtering is slightly improved to $\kappa = 26$ and $\rho = 37$) , while for the depth 2 we get similar results, except that (1) for *parser*, the static filtering is $\kappa = 91$ and $\rho = 69$ and the abstract filtering is $\kappa = 39$

depth 3		seminaive	static f.	abstract f.
parser	κ	1027 (11.2)	91 (1.00)	39 (0.42)
	ρ	1058 (15.33)	69 (1.00)	38 (0.55)
rev*	κ	42 (1.00)	42 (1.00)	42 (1.00)
	ρ	84 (1.31)	64 (1.00)	64 (1.00)
path/sd	κ	41 (1.00)	41 (1.00)	21 (0.51)
	ρ	125 (1.00)	125 (1.00)	40 (0.32)
path/md	κ	369 (9.20)	40 (1.00)	20 (0.50)
	ρ	824 (9.69)	85 (1.00)	38 (0.44)
ackerman*	κ	28 (1.12)	25 (1.00)	25 (1.00)
	ρ	116 (2.52)	46 (1.00)	44 (0.95)

and $\rho = 38$; (2) for ackerman, the static filtering is $\kappa = 25$ and $\rho = 46$ and the abstract filtering is $\kappa = 25$ and $\rho = 44$. Therefore, small depth of abstraction may be very useful in optimization.

8 Other related works

There have been two main approaches, static filtering and dynamic filtering, in filtering the bottom-up evaluation on system graphs. The basic idea of the static filter computation is to simulate top-down execution on system graphs. It may be be considered as a top-down phase without a bottom-up phase. *Dynamic filters* are computed at run-time by *sideways information propagation* and *backward information propagation* which propagate actual data generated during the evaluation [4, 7]. Roughly speaking, the abstract filters approximate dynamic filters. The bottom-up phase can be compared to the sideways information propagation while the top-down phase corresponds to the backward information propagation. However, since an approximation of all success patterns are known by the bottom-up phase before the top-down phase, it is possible that the abstract filter gives better optimization effects for some programs than the dynamic filtering and the magic set based methods. See Example 4 for details.
Magic Sets in [1] transforms original programs so that they can compute call patterns of atoms (which is called magic sets) and can reduce the generation of irrelevant facts using them. The transformed program computes the magic sets by simulating the sideways information passing of a top-down evaluation during the evaluation, and reduce the amount of potentially relevant data using them. The behavior of the transformed rule by Magic Sets is very similar to that of the dynamic filtering, if the same sideway passing graph is assumed. Magic Sets and the dynamic filtering, however, incur some runtime overhead for the computation of magic predicates and dynamic filters, respectively, while the static filter and abstract filter does not except filter passing. We will not consider the dynamic filtering in the following comparison, because it shows a very similar behavior to Magic Sets based methods.
We have already presented a typical example to show the difference between the static filter and abstract filter in Example 3. Even though Magic Sets computes calling patterns in the course of the query evaluation, it may create some facts which can not participate in any success pattern of a clause. This may happen also in our approach. The abstract filter, however, can prevent some programs from generating unnecessary facts which are generated by Magic Sets.

Example 4 *Consider the following program P and its transformed program by Magic Sets.*

$\leftarrow p(f(a), X).$
$\delta_1 : p(X, f(Y)) \leftarrow r(X, Z), p(Z, Y).$
$\delta_2 : r(X, f(Y)) \leftarrow r(X, Y).$

$r(f(a),a)$. $p(a,b)$.

$\delta_1' : p(X, f(Y)) \leftarrow magic\_p(X), r(X,Z), p(Z,Y)$.
$\delta_2' : r(X, f(Y)) \leftarrow magic\_r(X), r(X,Y)$.
$r(f(a),a)$. $p(a,b)$.
$magic\_r(X) \leftarrow magic\_p(X)$. *(from δ_1)*
$magic\_p(Z) \leftarrow magic\_p(X), r(X,Z)$. *(from δ_1)*
$magic\_r(X) \leftarrow magic\_r(X)$. *(from δ_2)*
$magic\_p(f(a))$. *(from the query)*

The rule δ_2' generates infinite facts in the query evaluation, and $magic\_p(Z)$ and $magic\_r(X)$ does infinite facts such as $magic\_p(f(a))$, $magic\_p(f(f(a)))$, and so on. The static filtering degenerates to the naive evaluation and generates infinite facts by the rule δ_2. On the other hand, when the depth of abstraction is 2, the abstract filters are as follows:

$AF_{\delta_1,1} = \{r(f(a),a), r(f(a),f(a))\}$
$AF_{\delta_1,2} = \{p(a,b), p(f(a),b), p(f(a),f(b)), p(f(a),f(f(\_)))\}$
$AF_{\delta_2,1} = \{r(f(a),a), r(f(a),f(a))\}$

The evaluation with the abstract filter generates only $r(f(a),a)$ and $r(f(a),f(a))$ by δ_2.

Magic Set could be more effective in many cases, but it should be noted that even though the abstract filter is computed at compile time, the optimization can be more effective for some programs than the magic set based methods.

9 Discussion

Consider the rules in Example 2 and suppose a graph $G = (V, E)$ is represented by facts in the program. In the abstract interpretation, when the depth of abstraction is k, the number of instances of $path(X, [X|P)$ in $lfp_a(U_P^A)$ is bounded by $|V|^{k-1}$ and that of the rule δ_1 in $lfp_c(U_P^A)$ by $|E| \times |V|^{k-2}$, whether the graph is acyclic or not. In case the depth of abstraction is 2, the numbers are linear in $|V|$ and $|E|$, respectively. On the other hand, in the real execution, when the graph is acyclic, the number of instances of $path(X, [X|P])$ in $lfp(T_P)$ is bounded by $(|V|)!$, and may be infinite when the graph is cyclic. In addition, it should be noted that, as shown in the experiment, small depth of abstraction is very useful in filtering nondeterministic programs. Moreover, the proposed approach has merit in the sense that, due to the goal independence of the bottom-up abstract interpretation, the result of the bottom-up phase can be used for any query to the same program. Since many different queries are usually asked over the same program, it is a good property.

Address:

- Byeong-Mo Chang and Kwang-Moo Choe
  Department of Computer Science, KAIST
  373-1, Kusung-dong, Yusung-ku, Taejon 305-701,
  South Korea, {`chang,choe`}`@plhae.kaist.ac.kr`;
- Roberto Giacobazzi
  LIX, École Polytechnique, 91128 Palaiseau cedex,
  France, `giaco@lix.polytechnique.fr`

References

[1] Bancilhon, F., Maier, D., Sagiv, Y. and Ullman, J.D , Magic sets and other strange ways to implement logic programs, *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1986, pp. 1–15.

[2] Bancilhon, F. and Ramakrishnan, R., An amateur's introduction to recursive query processing strategies, *Proc. SIGMOD Int. Conf. on Management of Data*, 1986, pp. 16–52.

[3] Barbuti, R., Giacobazzi R., and Levi, G. A general framework for semantics-based bottom-up abstract interpretation of logic programs, *ACM Transactions on Programming Languages and Systems*, 15:133–181 (1993).

[4] Chang, B.-M., Choe, K.-M., and Han., T. Optimized bottom-up evaluation of function-free logic programs with dynamic filtering: an incremental approach. Technical Report, CS-TR-92-66, Dept. of Computer Science, KAIST, Feb. 1992.

[5] Cousot, P., and Cousot., R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points, *Proc. of Fourth ACM Symp. Principles of Programming Languages*, 1977, pp. 238-252.

[6] Falaschi, M., Levi, G., Palamidessi, C. and Martelli, M., Declarative modeling of the operational behavior of logic languages, *Theoretical Computer Science*, 69:289–318 (1989).

[7] Kifer, M., and Lozinskii., E. L., A framework for an efficient implementation of deductive databases. In *Proc. of the 6-th Advanced Database Symposium*, 1986.

[8] Kifer, M., and Lozinskii., E. L., SYGRAF: Implementing logic programs in a database style. *IEEE Trans. on Soft. Eng.*, 1988, pp. 922–935.

[9] Lloyd, J.W., *Foundation of Logic Programming*, Springer-Verlag, 1984.

[10] Ramakrishnan, R., Magic Templates: A spellbinding approach to logic programs, *Proc. of the 1988 International Conference and Symposium on Logic Programming* (Seattle), 1988, pp.140–159.

[11] Sato, T., and Tamaki, H., Enumeration of success patterns in logic programs, *Theoretical Computer Science*, 34:227–240 (1984).

[12] Sterling, L., and Shapiro, E., *The art of Prolog.* The MIT Press, 1986.