Improving Execution Models of Logic Programs by Two-phase Abstract Interpretation

Byeong-Mo Chang, Kwang-Moo Choe and Roberto Giacobazzi

CONTENTS

ABSTRACT

This paper improves top-down execution models of logic programs based on a two-phase abstract interpretation which consists of a bottom-up analysis followed by a top-down one. The two-phase analysis provides an approximation of all (possibly non-ground) success patterns of clauses *relevant* to a query. It is specialized by considering Sato and Tamaki's depth k abstraction as abstract function. By the ability of the analysis to approximate possibly non-ground success patterns of clauses relevant to a query, it can be statically determined whether some subgoals will fail during execution and some succeeding subgoals do not participate in success patterns of program clauses relevant to a given query. These properties are utilized to improve execution models. This approach can be easily applied to any top-down (parallel) execution models. As instances, it is shown to be applicable to linear execution model and AND/OR Process Model.

I. INTRODUCTION

Abstract interpretation of a program approximates its standard semantics by fixpoint execution over an abstract domain rather than a possibly infinite concrete domain [1]. An abstract domain is typically a lattice which is finite or does not allow infinite ascending chains even if it is infinite. Abstract interpretation provides a safe and finite approximation of some runtime behavior of the program. For instance, a number of abstract interpretations have been proposed to approximate success patterns of atoms or clause [2]-[9]. The ones in [6], [7], [9] are based on top-down evaluation while the ones in [2]-[5], [8] on bottom-up evaluation.

Recently, bottom-up abstract interpretation of logic programs has gained much attention. The analysis is based on a concrete semantics which propagates the information in the opposite direction with respect to the Selective Linear resolution for Definite clauses (SLD) refutation. This process corresponds precisely to the least fixpoint computation of the standard T_P operator as introduced by van Emden and Kowalski in [10]. This approach has been considered in [2], [3], [8] as a basis for bottom-up dataflow analysis of logic programs. It usually provides an approximation of success patterns by evaluating an abstract version of T_P over a (possibly simpler) abstract domain.

In this paper we first provide a *two-phase abstract interpretation* introduced in [11] with more clarification, which consists of a bottom-up analysis followed by a top-down one. A two-phase analysis can be designed with considering only abstract atoms as in [12]. However, incorporating abstract clauses (abstract instances of clauses) in the abstract analysis allows more intelligence than abstract atoms, when they are applied to improve some execution models (see discussion in Section V for more details). Therefore, for the bottom-up phase, the basic bottom-up abstract interpretation in [2] is extended so as to incorporate abstract clauses. It approximates, by abstract clauses, success patterns of clauses in the possible successful computations for every possible query, so it is *query independent*. The extended bottom-up abstract interpretation is able to handle possibly non ground syntactic objects since the framework in [2], which is based on the concrete semantics in [13], can handle non ground syntactic objects. The top-down analysis collects *relevant* abstract success patterns of clauses for a given query from the result of the bottom-up analysis, so it is *query dependent*. Therefore, the two-phase analysis provides an approximation of success patterns of clauses *relevant* to a given query. The framework is specialized by considering the depth k abstraction [9] as abstraction function. It should be noted that the idea of two phases in abstract interpretation is not new in this research area even though there is no literature published in the form of this paper.

The two-phase analysis is used to improve the top-down execution of logic programs, keeping advantage from the ability of the analysis to approximate answer substitutions and (possibly non-ground) success patterns of program clauses relevant to a query. Several optimizations can be applied based on the approximation of success patterns of program clauses relevant to a query. In particular it allows us to detect whether some calls to specific program clauses will never succeed in the real execution and some succeeding subgoals do not participate in any success patterns of program clauses relevant to a given query. A suitable execution model is provided based on these properties. The new execution model uses two kinds of condition check, *call check* and *exit check*. The call check does not allow unnecessary calls to *future failing* program clauses. Using the exit check, even if a subgoal succeeds with a solution, the new execution model discards the solution if it is certain from the analysis that the resulting subgoal does not participate in any relevant success patterns of program clauses to a given query. The new execution model can be easily applied to any top-down (parallel) execution models (*e.g.* those introduced in [14]-[18], [22]) of logic programs, provided that they are forwarding schemes. As an example, we improve the AND/OR process model for parallel interpretation of logic programs in [14] by using the two principles above. The improvement described above can be suitably based on any type-analysis (not necessarily depth k).

This paper has a contribution in the sense that it introduces a practical use, that is the new improved execution model, of abstract interpretation, even though it is not very new to prune off the search with the result of static analysis. Moreover, the proposed approach has an advantage over top-down based analyses [6], [7], [9] in the sense that the result of the bottom-up analysis of a program can be utilized for any query to the program due to the goal independence of the bottom-up abstract analysis. Only additional top-down *collecting* phase over the result of the bottom-up analysis is necessary for a different query. Since many different queries are usually asked over the same program, this is a good property. In addition, it should be noted that applications of the two-phase dataflow analysis is not necessarily confined to the new execution model. The two-phase analysis is also applied to other compile-time optimization like query optimization in bottom-up evaluation as in [19].

The paper is structured as follows. The next section gives preliminaries on logic program and abstract interpretation. Section III describes a two-phase abstract interpretation, where a bottom-up abstract interpretation is extended and relevant abstract success patterns for a query are collected from the result of the bottom-up analysis. Section IV describes an efficient execution model based on the abstract analysis. In Section V, the proposed approach is compared with other related works. Section VI presents the concluding remarks.

II. PRELIMINARIES

1. Basic Notations

Let (Π, Σ, Var) denote a first order language, namely a (finite) set Π of predicate symbols, a set Σ of function symbols, and a denumerable set of variables Var. With each function symbol $f \in \Sigma$ and predicate symbol $p \in \Pi$ is associated a unique natural number called its *arity*: a (predicate or function) symbol f with arity n is written f/n. The set of all terms constructed from Σ and Var is denoted by $Term(\Sigma, Var)$ (or *Term* for short). An atom is a syntactic object of the form $p(t_1, \ldots, t_n)$ where $p/n \in \Pi$ and $t_1, \ldots, t_n \in Term(\Sigma, Var)$. We denote *Atoms* the set of all atoms constructed from Π and $Term(\Sigma, Var)$. A *goal* is a (possibly empty) sequence of atoms, and is typically written as $\langle B_1, \ldots, B_n \rangle$ or simply as $B_1, \ldots, B_n$. A *Horn clause* is a syntactic object of the form $H \leftarrow \bar{B}$ where H is an atom, called the *head*, and $\bar{B}$ is a goal, called the *body*. If the body is empty, the clause is denoted H and called a *fact*; otherwise it is called a *rule* . Rules are sometimes denoted with the Greek letter δ. The set of clauses constructed from elements of *Atoms* is denoted $Clause(\Pi, \Sigma, Var)$ or *Clause* for short. The set of variables occurring in a syntactic object t is denoted by $var(t)$. We define a function *gr* which maps any set of syntactic objects into the corresponding set of their ground instances. A *substitution* $\vartheta(x)$ is a mapping from *Var* to *Term*, such that $\{x \in Var \mid \vartheta(x) \neq x\}$ is finite. It extends to apply to any syntactic object in the usual way. The identity substitution is denoted ϵ. The set of idempotent substitutions is denoted *Sub*. Following tradition, the application of a substitution θ to an object t will be written $t\theta$. We fix a partial function *mgu* which maps a pair of syntactic objects to an idempotent most general unifier of the objects. A statement $\theta = mgu(s, t)$ implies that s and t are unifiable. The notation for *mgu* is extended as usual for sets of equations. We write $mgu(\langle A_1, \ldots, A_n \rangle, \langle B_1, \ldots, B_n \rangle)$ to denote the most general unifier of the set of equations $\{A_1 = B_1, \ldots, A_n = B_n\}$. Note that $mgu(\langle\ \rangle, \langle\ \rangle) = \epsilon$. In the following, $\vartheta_{|s}$ will denote the restriction of the substitution ϑ to the variables occurring in the syntactic object s, extended as an identity for each variable $x \in var(s)$ such that $\vartheta(x)$ is undefined. When S is a set and $\sim$ is an equivalence relation on S, $S/_\sim$ is the set of equivalence classes on S with respect to $\sim$. For an element $a \in S$, $[a]_\sim$ denotes the equivalence class of a with respect to $\sim$. We let $\wp(S)$ denote the power set of a set S.

2. Concrete Semantics

As first shown in [13], *"the van Emden and Kowalski's semantics is not correct with respect to the observational equivalence based on computed answer substitutions,"* while it is correct with respect to the one based on successful refutations; namely, there exist programs which have the same least Herbrand model semantics, yet compute different answer substitutions. When trying to understand the meaning of programs, when analyzing and transforming programs, this semantics cannot be taken as the reference semantics.

The concrete semantics introduced in [13] is closed to the operational behavior of logic programs being able to model computed answer substitutions. The idea is to enhance the standard semantics in [10] to deal with possibly non ground semantic objects. This provides a bottom-up semantics which fully characterizes the ability of logic programs to compute substitutions (which are non ground in general). Let the *extened Herbrand universe* U_P be $Term(\Sigma, Var)/_\sim$, where $\sim$ is the *variance* relation (*i.e.* $t_1 \sim t_2$ iff $\exists\vartheta_1, \vartheta_2 | t_1\vartheta_1 = t_2 \wedge t_2\vartheta_2 = t_1$). The variance relation can be extended easily to any syntactic object (atoms, clauses, etc.). The *extended Herbrand base* B_P is $Atoms/_\sim$ where $\sim$ is the variance relation extended on atoms. An interpretation I is a subset of B_P. An interpretation I is a model for a program P iff $gr(I)$ is a Herbrand model for P.

Given an equivalence class $[A]_\sim$ of atoms and a finite set of variables V, it is always possible to find a representative A' in $[A]_\sim$ that contains no variables from V. For a syntactic object s and a set of equivalence classes of atoms (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, namely, that: $[A_i]_\sim \in I$; $var(A_i) \cap var(s) = \emptyset$, for $1 \le i \le n$; and $i \ne j$ implies $var(A_i) \cap var(A_j) = \emptyset$, for $1 \le i, j \le n$. For simplicity of exposition, we will abuse notation and assume that an atom represents its equivalence class and write A rather than $[A]_\sim$.

Definition 1 [13]
$T_P : \wp(B_P) \to \wp(B_P)$ is a transformation associated with a program P such that

$$T_P(I) = \left\{ A\vartheta \;\middle|\; \begin{array}{l} c = A \leftarrow B_1, \ldots, B_n \in P \\ \langle B_1', \ldots, B_n'\rangle \ll_c I \\ \vartheta = mgu(\langle B_1, \ldots, B_n\rangle, \\ \qquad \langle B_1', \ldots, B_n'\rangle) \end{array} \right\}.$$

The semantics of a program P is determined by $lfp(T_P) = T_P^\omega(\emptyset)$[1], the least fixpoint of T_P [13]. $lfp(T_P)$ is a model for P, which is the *fully abstract semantics* with respect to computed answer substitutions.

3. Abstract Interpretation

In the following, we assume the standard framework of abstract interpretation in [1]. This framework presupposes a least fixpoint characterization of the collecting semantics. We assume that a concrete interpretation for a program P can be defined in terms of a monotonic operator $E_P : E \to E$ on a concrete domain E, and an abstract analysis can be defined in terms of a monotonic operator $D_P : D \to D$ on an abstract domain D.

[1]We denote by f^α the ordinal power of a function f, such that $f^0(X) = X$, $f^\alpha(X) = f(f^{\alpha-1}(X))$ for every successor ordinal α and $f^\alpha(X) = \bigcup_{\delta<\alpha} f^\delta(X)$ for every limit ordinal α. We also denote by ω the second limit ordinal.

Definition 2 [1], [8]
An abstract interpretation $((E, \sqsubseteq), E_P, (D, \preceq), D_P, \alpha, \gamma)$ *consists of a complete lattice* $(E, \sqsubseteq)$*, a monotonic operator* $E_P : E \rightarrow E$*, a complete lattice* $(D, \preceq)$*, a monotonic operator* $D_P : D \rightarrow D$*, an abstraction function* $\alpha : E \rightarrow D$ *and a concretization function* $\gamma : D \rightarrow E$*, such that*

(1) α *and* γ *are monotonic,*

(2) $d = \alpha(\gamma(d))$ *for all* $d \in D$*,*

(3) $e \sqsubseteq \gamma(\alpha(e))$ *for all* $e \in E$ *and*

(4) $E_P(\gamma(d)) \sqsubseteq \gamma(D_P(d))$ *for all* $d \in D$*.*

Conditions (1)-(3) state that $((E, \sqsubseteq), \alpha, (D, \preceq), \gamma)$ forms a *Galois insertion*. Condition (4) is the *safeness* criterion that ensures that D_P faithfully mimics E_P.

Proposition 1 [1]
If $((E, \sqsubseteq), E_P, (D, \preceq), D_P, \alpha, \gamma)$ *is an abstract interpretation, then* $lfp(E_P) \sqsubseteq \gamma(lfp(D_P))$*.*

III. TWO-PHASE ABSTRACT INTERPRETATION

The static analysis technique in [2] is intended to provide an approximated description of a program model, by computing an abstract interpretation I^A (an abstract model) for the program, such that $lfp(T_P) \subseteq \gamma(I^A)$. However, this approach is not adequate in pruning off unnecessary calls in goals derived from a given query. Indeed, because of the correctness, any atom in $lfp(T_P)$ satisfies some abstract atoms in the abstract model. What we need is a weaker notion of correctness, which is specialized for a given query. Because of this observation, we introduce a two-phase abstract interpretation, collecting only those abstract descriptions for the relevant atoms for the query.

Even though some two-phase analysis can be designed with considering only abstract atoms as in [12], two-phase abstract interpretation is designed by incorporating abstract clauses, since incorporating abstract clauses in the abstract interpretation allows more intelligence than abstract atoms when they are applied to improve some execution models. The two-phase abstract interpretation consists of a bottom-up analysis followed by top-down one. The bottom-up phase is designed by extending the basic bottom-up abstract interpretation in [2] so as to incorporate abstract clauses. It approximates, by abstract clauses, success patterns of clauses in the possible successful computations for every possible query, so it is *query independent*. The top-down analysis is designed to collect *relevant* abstract success patterns of clauses for a given query from the result of the bottom-up analysis. The framework is specialized by considering the depth k abstraction [9] as abstraction function.

1. Basic Framework

In this subsection we first review the bottom-up abstract interpretation in [2] and extend it so as to include clauses as well as atoms as semantic objects. The framework is specialized for depth k abstractions. Of course, this framework can be easily extended to deal with different abstract interpretations, provided that the soundness conditions specified in [2] are satisfied.

Definition 3 [9]
A level k subterm is defined as follows:

(a) for a given term t, t is a level 0 subterm of t.

(b) 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.

We use the depth k abstraction [9] as abstraction function. 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*. Let $\rho(t)$ be a function which maps a term t to its depth k abstract term. Then *abstract universe* of a program P is defined by $U_P^A = \rho(Term(\Sigma, Var))$. The *abstract base* B_P^A of a program P is defined by $Atoms^A/_\sim$ where $Atoms^A = \{p(t_1^A, \ldots, t_n^A) | p/n \in \Pi, \{t_1^A, ..., t_n^A\} \subseteq U_P^A\}$, and $\sim$ is the variance relation on atoms. 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 equivalence class is represented as $A^A = [p(\rho(t_1), \ldots, \rho(t_n))]_\sim$. 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

$$\gamma_a(I^A) = \{A' \in B_P | A^A \in I^A, \alpha_a(A') = A^A\}.$$

Lemma 1
$((\wp(B_P), \subseteq), \alpha_a, (\wp(B_P^A), \subseteq), \gamma_a)$ is a Galois insertion.

Proof
See Appendix. □

An *abstract substitution* is a mapping from a given finite set of variables $V \subseteq Var$ into U_P^A. Given a substitution $\vartheta = \{t_1/x_1, \ldots, t_n/x_n\}$, α_Φ is defined by $\alpha_\Phi(\vartheta) = \vartheta^k$ where $\vartheta^k = \{\rho(t_1)/x_1, \ldots, \rho(t_n)/x_n\}$. A concretization mapping can be associated with α_Φ: γ_Φ such that $(\alpha_\Phi, \gamma_\Phi)$ is a Galois insertion. An abstract operator T_P^A is defined in [2] to approximate T_P.

Definition 4 [2]
$T_P^A : \wp(B_P^A) \rightarrow \wp(B_P^A)$

$$T_P^A(I^A) = \left\{ \alpha_a(A\vartheta^A) \;\middle|\; \begin{array}{l} c = A \leftarrow B_1, \ldots, B_n \in P, \\ \langle B_1^A, \ldots, B_n^A \rangle \ll_c I^A, \\ \vartheta^A = mgu^A(\langle B_1, \ldots, B_n \rangle, \\ \quad \langle B_1^A, \ldots, B_n^A \rangle) \end{array} \right\}$$

where $mgu^A(\langle B_1, \ldots, B_n \rangle, \langle B_1^A, \ldots, B_n^A \rangle) = \alpha_\Phi(\vartheta)$ and $\vartheta = mgu(\langle B_1, \ldots, B_n \rangle, \langle B_1^A, \ldots, B_n^A \rangle)$.

Theorem 1 [2]
Let P be a logic program. $lfp(T_P^A) =$

$(T_P^A)^n(\emptyset)$ for some finite n.

Proof

Since T_P^A is monotone and $\wp(B_P^A)$ is a finite lattice, $lfp(T_P^A) = (T_P^A)^n(\emptyset)$ for some finite n.

$\square$

Example 1

Let P be the following logic program

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

When the depth of abstraction is 2, $lfp(T_P^A)$ is as follows:

$$lfp(T_P^A) = \left\{ \begin{array}{l} arc(a, b), arc(a, c), arc(b, e), \\ arc(c, b), arc(c, d), arc(d, f), \\ arc(g, d),\ final(f), \\ path(f, [f]),\ path(d, [d|\_]), \\ path(c, [c|\_]),\ path(g, [g|\_]), \\ path(a, [a|\_]) \end{array} \right\}$$

We show the safeness of the abstract interpretation by the following lemma and theorem, since this property is not shown explicitly in [2].

Lemma 2

$T_P(\gamma_a(I^A)) \subseteq \gamma_a(T_P^A(I^A))$ for all $I^A \in \wp(B_P^A)$.

Proof

See Appendix. $\square$

Theorem 2

$\gamma_a(lfp(T_P^A)) \supseteq lfp(T_P)$ for any program P.

Proof

It follows by Lemma 1, Lemma 2 and Proposition 1. $\square$

The meaning of abstraction is captured by a suitable notion of *abstract model* for a program. $I^A \in \wp(B_P^A)$ is an *abstract model* for P iff there exists a concrete interpretation I which is a model for P (i.e. $gr(I)$ is a Herbrand model for P) and $\alpha_a(I) = I^A$. As shown in [2], any safe bottom-up abstract interpretation of a program P provides an abstract model for P.

Theorem 3 (strong soundness [2])

Let P be a logic program and let $G = B_1, \ldots, B_n$ be a goal. If ϑ is the substitution computed in a refutation of G in P, then

$$\{\vartheta_{|G}\} \in \gamma_\Phi(\{mgu^A\ (\langle B_1, \ldots, B_n\rangle, \langle B'_1, \ldots, B'_n\rangle)_{|G} \mid \langle B'_1, \ldots, B'_n\rangle \ll_G lfp(T_P^A)\}).$$

2. Extending the Framework

We now extend this framework by considering clauses as semantic objects. Let C be the set of (equivalence classes of) clauses up to renaming: $Clause/_\sim$. The set of *abstract clauses* is defined by $C^A = \{\alpha_c(C) | C \in C\}$ where for any clause $A \leftarrow B_1, \ldots, B_n$: $\alpha_c(A \leftarrow B_1, \ldots, B_n)$ is $[\alpha'_a(A) \leftarrow \alpha'_a(B_1), \ldots, \alpha'_a(B_n)]_\sim$ and $\alpha'_a(B_i)$

maps B_i into its depth k abstract term. We extend the bottom-up abstract interpretation to an *extended abstract interpretation*

$$((\wp(B_P) \times \wp(C), \subseteq), U_P, (\wp(B_P^A) \times \wp(C^A), \subseteq), U_P^A, \alpha, \gamma),$$

where, following the approach in [8], we extend T_P by introducing a concrete operator U_P so as to include clauses as well as atoms as semantic objects.

Definition 5
Let $U_P : \wp(B_P) \times \wp(C) \to \wp(B_P) \times \wp(C)$ *such that* $U_P(I, J) = (T_P(I), J')$ *where*

$$J' = \left\{ C\vartheta \;\middle|\; \begin{array}{l} C = A \leftarrow B_1, \ldots, B_n \in P \\ \langle B_1', \ldots, B_n' \rangle \ll_C I \\ \vartheta = mgu(\langle B_1, \ldots, B_n \rangle, \\ \quad \langle B_1', \ldots, B_n' \rangle) \end{array} \right\}.$$

The concrete collecting semantics of a program P is $lfp(U_P)$, and its first part is denoted by $lfp_a(U_P)$ and its second part by $lfp_c(U_P)$.

The abstraction function $\alpha : \wp(B_P) \times \wp(C) \to \wp(B_P^A) \times \wp(C^A)$ is defined by $\alpha(I, J) = (\alpha_a(I), \alpha_c(J))$. The concretization function $\gamma : \wp(B_P^A) \times \wp(C_P^A) \to \wp(B_P) \times \wp(C_P)$ is $\gamma(I^A, J^A) = (\gamma_a(I^A), \gamma_c(J^A))$ where $\gamma_c(J^A) = \{C | C' \in J^A, \alpha_c(C) = C'\}$. The abstract operator U_P^A is defined by extending T_P^A, and it contains abstract clauses as well as abstract atoms.

Definition 6
Let $U_P^A : \wp(B_P^A) \times \wp(C^A) \to \wp(B_P^A) \times \wp(C^A)$ *such that* $U_P^A(I^A, J^A) = (T_P^A(I^A), J^{A'})$ *where*

$$J^{A'} = \left\{ \alpha_c(C\vartheta^A) \;\middle|\; \begin{array}{l} C = A \leftarrow B_1, \ldots, B_n \in P, \\ \langle B_1^A, \ldots, B_n^A \rangle \ll_C I^A, \\ \vartheta^A = mgu^A(\langle B_1, \ldots, B_n \rangle, \\ \langle B_1^A, \ldots, B_n^A \rangle) \end{array} \right\}.$$

The abstract semantics is defined as the least fixed point $lfp(U_P^A)$ which is shown to be terminating in the following theorem. Its first part is denoted by $lfp_a(U_P^A)$ and the second part by $lfp_c(U_P^A)$. $lfp_a(U_P^A)$ is the same as $lfp(T_P^A)$ and provides an abstract model for the program, while $lfp_c(U_P^A)$ contains all abstract instances of clauses in P whose body goals are in $lfp(T_P^A)$. $lfp_c(U_P^A)$ provides the approximated success patterns of each (program) clause. The proof of the following theorem is similar to that of Theorem 1.

Theorem 4
Let P be a logic program. $lfp(U_P^A) = (U_P^A)^n(\emptyset)$ *for some finite n.*

Example 2
We give the least fixpoint of U_P^A for the program in Example 1. Let the depth of abstraction be 2.

$$lfp_a(U_P^A) = \left\{ \begin{array}{l} arc(a, b), arc(a, c), arc(b, e), \\ arc(c, b), arc(c, d), arc(d, f), \\ arc(g, d), \; final(f), \\ path(f, [f]), path(d, [d|\_]), \\ path(c, [c|\_]), path(g, [g|\_]), \\ path(a, [a|\_]) \end{array} \right\},$$

$$lfp_c(U_P^A) = \left\{ \begin{array}{l} arc(a,b).,arc(a,c).,arc(b,e).,\\ arc(c,b).,arc(c,d).,arc(d,\ f).,\\ arc(g,d),\ final(\ f),\\ path(\ f,[\ f]) \leftarrow\ final(\ f).,\\ path(d,[d|\_]) \leftarrow arc(d,\ f),\\ path(\ f,[\ f]).,\\ path(c,[c|\_]) \leftarrow arc(c,d),\\ path(d,[d|\_]).,\\ path(g,[g|\_]) \leftarrow arc(g,d),\\ path(d,[d|\_]).,\\ path(a,[a|\_]) \leftarrow arc(a,c),\\ path(c,[c|\_]). \end{array} \right\}.$$

The instances of path in $lfp_a(U_P^A)$ represent vertices, which are connected to the vertex f. The number of the facts of path in $lfp_a(U_P^A)$ is bounded by the number of vertices (see Section V).

The following lemma and theorem show the safeness of the bottom-up analysis.

Lemma 3
$\gamma(U_P^A(I^A, J^A)) \supseteq U_P(\gamma(I^A, J^A))$ *for all* $I^A \in \wp(B_P^A)$, $J^A \in \wp(C^A)$.
Proof
The proof is similar to that of Lemma 2. □

Theorem 5
$\gamma(lfp(U_P^A)) \supseteq lfp(U_P)$ *for any program* P.
Proof
It is straightforward to show that $((\wp(B_P) \times \wp(C), \subseteq), (\wp(B_P^A) \times \wp(C^A), \subseteq), \alpha, \gamma)$ is a Galois insertion. Therefore the theorem is proved by Lemma 3 and Proposition 1. □

3. Collecting Relevant Success Patterns for a Query

The bottom-up abstract analysis computes approximated success patterns without regard to a query. So, many of them are not relevant to the given query. The following top-down analysis over the result of the bottom-up analysis finds a subset of $lfp_c(U_P^A)$, which consists of abstract success patterns of program clauses relevant to a query. An abstract clause $H^A \leftarrow B_1^A, \ldots, B_n^A \in lfp_c(U_P^A)$ is said to be *relevant* to a query Q if

1. H^A is unifiable with some atom in Q.
2. H^A is unifiable with some body atom of some abstract clause relevant to Q.

We define an abstract interpretation

$$((\wp(lfp_c(U_P)), \subseteq), D_{P,Q}, (\wp(lfp_c(U_P^A)), \subseteq), D_{P,Q}^A, \alpha_c, \gamma_c)$$

for the top-down collecting phase. It is also possible to define the abstract operator on $\wp(lfp(U_P^A))$ rather than $\wp(lfp_c(U_P^A))$ (the same is applied to the concrete operator). We, however, adopted the abstract interpretation for simplicity and for the reason that information on atoms is easy to derive from it.

The concrete operator $D_{P,Q}$ is defined for a concrete collecting semantics for the top-down phase.

Definition 7

$D_{P,Q} : \wp(lfp_c(U_P)) \rightarrow \wp(lfp_c(U_P))$ *Consider a given query Q. Let body_atoms(J) be the set of body atoms of clauses in J.*

$$\begin{aligned} D_{P,Q}(J) = \{C | C = A \leftarrow & \\ & B_1, \ldots, B_n \in lfp_c(U_P), \\ & A' \in body\_atoms(Q \cup J), \\ & \exists mgu(A, A')\} \end{aligned}$$

The concrete collecting semantics for the top-down phase is determined by $lfp(D_{P,Q})$.

The abstract operator $D^A_{P,Q}$ is defined for the top-down phase.

Definition 8

$D^A_{P,Q} : \wp(lfp_c(U^A_P)) \rightarrow \wp(lfp_c(U^A_P))$ *Consider a given query Q. Let body_atoms(J^A) be defined as Definition 7.*

$$\begin{aligned} D^A_{P,Q}(J^A) = \{C^A | C^A = A^A \leftarrow & \\ & B^A_1, \ldots, B^A_n \in lfp_c(U^A_P), \\ & A^{A'} \in body\_atoms(Q \cup J^A), \\ & \exists mgu(A^A, A^{A'})\} \end{aligned}$$

The following theorem shows that the least fixpoint $lfp(D^A_{P,Q})$ can be computed in finite time. It contains all the abstract clauses in $lfp(U^A_P)$ which are relevant to the query.

Theorem 6

Let P be a logic program. $lfp(D^A_{P,Q}) = (D^A_{P,Q})^n(\emptyset)$ *for some finite n.*

Proof

Since $D^A_{P,Q}$ is monotone and $\wp(lfp_c(U^A_P))$ is a finite lattice, $lfp(D^A_{P,Q}) = (D^A_{P,Q})^n(\emptyset)$ for some finite n. □

Example 3

Consider a query of the form $p(a, Z)$. The result of the top-down phase for the program P in Example 2 is as follows.

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

We prove the safeness of the top-down phase in the following lemma and theorem.

Lemma 4

$\gamma_c(D^A_{P,Q}(J^A)) \supseteq D_{P,Q}(\gamma_c(J^A))$ *for all* $J^A \in \wp(G^A_P)$.

Proof

See Appendix. □

Theorem 7

$\gamma(lfp(D^A_{P,Q})) \supseteq lfp(D_{P,Q})$ *for any program P.*

Proof

It is easily shown that $((\wp(lfp_c(U_P)), \subseteq), \alpha_c, (\wp(lfp_c(U^A_P)), \subseteq), \gamma_c)$ is a Galois insertion. The proof is the Galois insertion, Lemma 4 and Proposition 1. □

IV. EFFICIENT EXECUTION MODELS OF LOGIC PROGRAMS

We first consider the linear execution of logic program like Prolog. A model of linear execution is provided as a transition system in [20], which is a pair (X, R) consisting of a set X of states and a transition relation $R \subseteq X \times X$ on states.

Definition 9 [20]
Let P be a logic program. The transition system Ψ_P is $(Atom^*, \rightarrow_P)$ where $\rightarrow_P \subseteq Atom^* \times Atom^*$ is the smallest relation such that $G \rightarrow_P G'$ iff

1. $G = < A_1, \ldots, A_i, \ldots, A_k >, 1 \leq i \leq k$;
2. $H \leftarrow B_1, \ldots, B_n \ll_G P$ *such that* $\varphi = mgu(A_i, H)$; *and*
3. $G' = < A_1, \ldots, B_1, \ldots, B_n, \ldots, A_k > \varphi$.

The substitution φ is referred as the associated substitution of the transition $G \rightarrow_P G'$. A successful derivation can be stated as: $G \rightarrow_P^*$ which means the empty clause is derivable from G.

In SLD resolution, the definition is refined by introducing a *selection function* on the atoms in a state. In particular, Prolog assumes a left-to-right selection function. Moreover, logic programming languages typically introduce a control strategy to specify which clauses should be applied to the selected atom. As well known, Prolog applies a depth-first search strategy. We assume Prolog's left-to-right and depth-first search strategy for simplicity, even though the following result is not restricted to Prolog. More details on semantic definitions for logic programming languages are found in [20], [21].

We improve the execution by applying the semantics-based dataflow analysis as presented in the previous section. The safeness of the two-phase analysis provides the following theorem which is the basis for the new execution model.

Theorem 8
Let $\delta = H \leftarrow A_1, \ldots, A_n$ be a rule and A_i' be an instance of A_i, for some $1 \leq i \leq n$. Let $lfp(D^A_{P,Q})[\delta][i]$ denote the set of all (abstract) instances of A_i for the rule δ in $lfp(D^A_{P,Q})$. If A_i' is not unifiable with any of the abstract atoms in $lfp(D^A_{P,Q})[\delta][i]$, then for each $\sigma \in Sub$, $A_i'\sigma \notin lfp(D_{P,Q})[\delta][i]$.

Proof
Because the subgoal A_i' is not unifiable with any abstract atom in $lfp(D^A_{P,Q})[\delta][i])$, there exist no common instances for A_i' and for some abstract atom in $lfp(D^A_{P,Q})[\delta][i]$. By the definition of γ_a, $\gamma_a(lfp(D^A_{P,Q})[\delta][i])$ contains all instances of abstract atoms in $lfp(D^A_{P,Q})[\delta][i]$. So, no instances of A_i' are in $\gamma_a(lfp(D^A_{P,Q})[\delta][i])$. Moreover, it can be easily shown by Theorem 7 that $\gamma_a(lfp(D^A_{P,Q})[\delta][i]) \supseteq lfp(D_P)[\delta][i]$. Therefore, no instances of A_i' are in $lfp(D_{P,Q})[\delta][i]$. □

Based on the theorem, the following provides two principles in the new execution model. Let us consider the assumptions in the Theorem 8.

- **call check** : A subgoal A_i' need not be called if it is not unifiable with any abstract atom in $lfp(D^A_{P,Q})[\delta][i]$ since it will fail.
- **exit check** : When a subgoal A_i' is called and succeeds with a solution σ, the solution is discarded if $A_i'\sigma$ is not unifiable with any of the abstract atoms in $lfp(D^A_{P,Q})[\delta][i]$ since it will not participate in any of the success patterns of δ.

Based on the two principles, the new execution model tests the call check condition whenever a call is made and test the exit check condition whenever a call succeeds. Consider the transition system in the Definition 9. The new model tests the call check for the subgoal A_i before the call in the second step. If the subgoal passes the call check and succeeds, the exit check is tested for the subgoal with the answer substitution being applied. In the implementation, if we consider the general selection rule, then in order to perform the checks we should record in the execution model whose clauses introduced the atoms of the current goal. However this can be easily implemented. The given query $Q = \leftarrow B_1, \ldots, B_n$ can also be checked with its abstract instances :

$$\left\{ \alpha_c(Q\vartheta^A) \;\middle|\; \begin{array}{l} \{B_1^A, \ldots, B_n^A\} \ll_Q lfp(T_P^A), \\ \vartheta^A = mgu^A(\langle B_1, \ldots, B_n\rangle, \\ \quad \langle B_1^A, \ldots, B_n^A\rangle) \end{array} \right\}.$$

The following example shows the new execution over a program and query easily.

Example 4

Let us consider the program in Example 1. We consider the new execution for the given query $\leftarrow path(a, P)$. *In the evaluation of* $path(a, P)$, *a call* $arc(a, N)$ *in* δ_1 *succeeds with* $arc(a, b)$ *and* $arc(a, c)$, *but* $arc(a, b)$ *is discarded since it is not unifiable with any abstract atoms in* $lfp(D^A_{P,Q})[\delta_1][1]$. *So, only* $path(c, P)$ *is called. In the evaluation of the call* $path(c, P)$, *the new evaluation calls* $arc(c, N)$ *and succeeds with* $arc(c, b)$ *and* $arc(c, d)$, *but discards* $arc(c, b)$ *based on the principle above. So, only* $path(d, P)$ *is called. From the goal* $path(a, P)$ *until the call* $path(f, P)$, *the new execution makes 5 calls for* $arc(X, N)$ *and 3 calls for* $path(X, N)$. *On the other hand, the naive execution makes 9 calls for* $arc(X, N)$ *and 7 calls for* $path(X, N)$. *See Section V for another example.*

The two-phase abstract analysis can be easily applied to improving other top-down (parallel) execution models (*e.g.* those introduced in [14]-[18], [22]), if they are forwarding schemes. As an example, we consider the *AND/OR Process Model* [14] which exploits AND and OR parallelism in logic programs. In the model, AND/OR process tree forms a bipartite graph: an AND process for a clause creates child OR processes for each body atom in the clause, and each OR process for an atom creates child AND processes for candidate clauses simultaneously. A parent process and its child processes communicate by exchanging four types of messages: *success*, *fail*, *redo* and *cancel*. *Success* and *fail* messages are sent by a child process to a parent process to notify that the child process succeeds to find a solution, or fails. *Redo* and *cancel* messages are sent by a parent process to a child process to ask the child process to produce another solution, or to terminate.

To deal with shared variables within a clause, an AND process maintains the data dependency graph, which represents *producer/consumer* relationship among body atoms sharing variables in the clause [14], [22]. The data dependency graph can be constructed at compile or run time. We assume the static data dependency graph [22] for easy presentation though our method is applicable to dynamic one. AND process works in iterations of two phases: *forward execution* and *backward execution*. Forward execution is a graph reduction procedure: OR processes are created for body atoms having no unsolved producers in the data dependency graph. When a child OR process fails, the backward execution selects a backtrack atom and redoes the OR process for the backtrack atom [14], [23], [24]. The followings are principles for the new forward execution based on Theorem 8.

- **call check** : An OR process need not be created for A_i' if it is not unifiable with any abstract atoms in $lfp(D^A_{P,Q})[\delta][i]$.
- **exit check** : When an OR process is created for A_i' and a success message comes from it with a solution σ, the solution is discarded if $A_i'\sigma$ is not unifiable with any of the abstract atoms in $lfp(D^A_{P,Q})[\delta][i]$.

We now describe a new forward execution algorithm in AND process. The body atoms of an AND process can be in one of the following three states: *blocked*, *pending* or *solved*. An atom is *blocked* when an OR process is not created for it yet. An atom is *pending* when an OR process was created for it but has not yet sent back any message. An atom is *solved* after an OR process created has sent back a *success* message. The new forward execution in AND process consists of two parts. The first part to process start message is executed once when the AND process is created. The second part is executed to process a success message from a child OR process. In the algorithm, $pred(A_k)$ means a set of all predecessors of A_k in the data dependency graph DDG_δ for a rule δ.

New Forward Execution Algorithm in AND Process for a rule δ: $H \leftarrow A_1, \ldots, A_n$.

DDG_δ : the data dependency graph of the rule δ.

$Solved$, $Blocked$, $Pending$: state variables of the AND process, representing sets of atoms in each status.

$\theta_{current}$: substitution maintaining current solution of an AND process for the rule δ.

To process a start message:

1. Initialization: $Solved = \{H\}$, $Pending = \emptyset$, $Blocked = \{A_1, \ldots, A_n\}$, and $\theta_{current} = \theta_{initial}$ where $\theta_{initial}$ is the mgu of the atom of the parent OR process and H.

2. For every atom A_i such that $pred(A_i) \subseteq Solved$,

   if $A_i' = A_i\theta_{current}$ is unifiable with an abstract atom in $lfp(D^A_{P,Q})[\delta][i]$,

   then create OR process for the atom and move A_i from *Blocked* to *Pending*,

   else call *backward execution algorithm*.

To process a success message with solution σ from OR process for A_i:

1. If $A_i' = A_i\theta_{current}\sigma$ is not unifiable with any of the abstract atoms in $lfp(D^A_{P,Q})[\delta][i]$

   then discard it and ask one more solution (redo) to the OR process.

2. Otherwise $\theta_{current} = \theta_{current}\sigma$

   2.a Move the solved atom from *Pending* to *Solved*.

   2.b If all the body atoms are in *Solved*,

   then send a success message to the parent process,

   else continue.

   2.c For each body atom $\{A_k | A_k \in Blocked, \ pred(A_k) \subseteq Solved\}$

   If $A_k' = A_k\theta_{current}$ is unifiable with an abstract atom in $lfp(D^A_{P,Q})[\delta][k]$,

   then create an OR process and move A_k from *Blocked* to *Pending*,

   else call *backward execution algorithm*.

In the above algorithm, the backward execution algorithm is called instead of creating a child OR process when we can know from the analysis that the OR process will fail. We assume the backward execution algorithm in [24].

V. RELATED WORKS AND DISCUSSION

1. Related Works

There are several general frameworks for bottom-up semantics-based abstract interpretations of logic programs [2], [3], [8]. The frameworks in [2], [3] are based on the concrete semantics in [13] which defines a bottom-up semantics over a domain of non-ground Herbrand interpretations. We extend the bottom-up abstract interpretation framework in [2] so as to approximate possibly non-ground success patterns of program clauses also. Abstract interpretation for finding success patterns of clauses is *not* an original idea, and it was first introduced in [7]. However, any standard (concrete) semantics based on ground semantic objects (like that used in [8]) fails in modeling answer substitutions for a program. As shown in [13], this requires a more refined notion of interpretation (semantics), being able to handle possibly non ground semantic objects. Moreover, the two-phase framework based on the extended bottom-up analysis collects a subset of approximated success patterns of clauses *relevant* to a query, so it is more likely to give better performance for a number of applications of compile-time optimization than the approach based on the bottom-up analysis as in [25]. Even though the idea of two phases in abstract interpretation is not very new, the two-phase framework in this paper would give considerable impacts on compile-time optimizations of logic programs.

The idea of using depth k information for more efficient execution is originated from Sato and Tamaki's work [9], in which depth k success patterns are computed by item set construction algorithm similar to the construction of LR item sets and future failing execution branches are pruned off by taking advantage of them. Kanamori uses depth k abstraction in [6] in which depth k success patterns are computed based on OLDT resolution [17]. A framework for abstract interpretation is also provided based on OLDT resolution in [7].

Over the analyses in [6], [7], [9], the proposed approach has an advantage in the sense that the result of the bottom-up abstract analysis can be utilized for any queries to the same program due to the goal independence of the bottom-up abstract analysis. Whenever another query is given to the same program, it is sufficient to execute only the top-down phase again over the original result of the bottom-up abstract analysis. This is a nice property since many different queries are usually asked over the same program.

In addition, we can provide a simpler version of our approach. As a good approximation of the new execution model, we can execute it by applying the result of the bottom-up abstract analysis without the top-down analysis phase. The two principles of the execution can be stated based on the result of the bottom-up analysis as follows:

- **call check** : A subgoal A_i' need not be called if it is not unifiable with any abstract atom in $lfp_c(U_P^A)[\delta][i]$.
- **exit check** : When a subgoal A_i' is called and succeeds with a solution σ, the solution is discarded if $A_i'\sigma$ is not unifiable with any of the abstract atoms in $lfp_c(U_P^A)[\delta][i]$.

There are, of course, examples for which the top-down abstract analysis adds more intelligence over the approach without it. However, only the bottom-up abstract analysis can also give intelligence for many programs. In this simpler approach, only one execution of the bottom-up analysis is sufficient for any queries for a program even though it could lose some intelligence for some programs.

2. Discussion

The enumeration of all the success patterns in the bottom-up abstract interpretation may be time consuming. This problem is partly overcome by using the depth k abstraction as abstraction function. Consider the rules in Example 1 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, the number of instances of $path(X, [X|P])$ in $lfp(T_P)$ is bounded by $|V|!$ when the graph is acyclic, and may be infinite when the graph is cyclic. The number of iterations to compute $lfp(U_P^A)$ is basically the same as the number to compute $lfp(T_P^A)$, which is clear from the definition of U_P^A. Even though some two-phase analysis can be designed with considering only abstract atoms [12], incorporating abstract clauses in abstract interpretation allows more intelligence than abstract atoms when they are applied to improve some execution models.

If we consider some two-phase analysis which considers only abstract atoms, the improved execution model has to check a call with all abstract atoms which have the same predicate as the call. On the other hand, the improved execution model with the proposed two-phase analysis checks a call with some relevant abstract atoms in abstract clauses as described in Theorem 8. They are much smaller than all abstract atoms with the same predicate as the call. Therefore, abstract clauses deserve to be incorporated in the two-phase analysis. It should be noted that, as shown in the examples, small depth of abstraction is useful in eliminating unnecessary calls (processes).

The proposed approach has merit related with *goal independence*. The bottom-up analysis of a program can be used for any query. Once the initial query is changed, only the top-down abstract analysis is required over the result of the additional bottom-up analysis. Since many different queries are usually asked over the same program, goal independence of the bottom-up analysis is an important property to provide compile-time optimizations.

The improved execution model can be implemented efficiently in the following senses : (1) the abstract atoms in $lfp(D_{P,Q}^A)[\delta][i]$ can be searched efficiently by using some index techniques, and (2) overhead due to the checks is not so large when we consider the benefit by pruning the search of whole subtrees starting from unnecessary calls. Moreover, the proposed method prevents even infinite calls in the evaluation of some programs.

Example 5

Let us consider the rules in Example 1, a query $\leftarrow path(a, X)$, and the following graph which contains a cycle.

$$arc(a, b).\ arc(a, c).\ arc(b, e).\ arc(c, b).$$
$$arc(c, d).\ arc(d, f).\ arc(e, b).$$

In the seminaive evaluation, the call $path(b, P)$ in the rule δ_1 generates infinite calls due to the cycle. In the new evaluation, $path(b, P)$ is not called since it is not in the approximated success patterns.

VI. CONCLUDING REMARKS

We have presented the two-phase analysis which provides an approximation of success patterns of every clause relevant to a query. With this dataflow analysis, a new model of execution is proposed which does not allow unnecessary calls and succeeding subgoals irrelevant to a query utilizing the depth k success patterns information. It is shown as well that the new model is applicable to other forwarding top-down execution. Only if the bottom-up analysis of a program is done once, our approach allows only additional top-down collecting analysis over the result of the bottom-up analysis to be sufficient for any query to the program.

As well as the proposed dataflow analysis can be applied to other compile-time optimizations [19], it can be easily extended to any safe approximation such as ground dependencies, sharing, type inference, *etc*. This might have some very interesting outcomes. The result of an abstract interpretation can be viewed as a *constraint* on the set of all the possible answers for a predicate. While our approach applies the depth k abstraction, it seems natural to extend this technique to any *type*-approximation for success patterns like [26], [27]. For instance, if we have a type information on the success patterns for a predicate p (a bottom-up abstract interpretation for type inference in logic programming is in [26]) the abstract atom $p(int, char)$ may represent a constraint for p specifying that x and y will have type *int* (integer) and *char* (character), respectively, in any successful computation. This information can also be used to improve the efficiency of top-down executions, following the same model described before.

APPENDIX

A. Proof of Lemma 1

Notice that α_a is additive (*i.e.* for any $D \subseteq \wp(B_P)$: $\alpha_a(\cup D) = \cup_{I \in D}\alpha_a(I)$). By α_a additivity it is always possible to define a concretization function γ_a which corresponds to the definition and such that for any $I^A \subseteq B_P^A$, $I^A = \alpha_a(\gamma_a(I^A))$, and for any $I \subseteq B_P$, $I \subseteq \gamma_a(\alpha_a(I))$. $\square$

B. Proof of Lemma 2

We prove the lemma by showing the following.

$$\left\{ \begin{array}{c} C = A \leftarrow B_1, \ldots, B_n \in P \\ A\vartheta \,|\, \langle B_1', \ldots, B_n' \rangle \ll_C \gamma_a(I^A), \\ \vartheta = mgu(\langle B_1, \ldots, B_n \rangle, \langle B_1', \ldots, B_n' \rangle) \end{array} \right\} \subseteq$$

$$\gamma_a\left(\left\{\alpha_a(A\vartheta^A)\middle|\begin{array}{c}C=A\leftarrow B_1,\ldots,B_n\in P,\\ \langle B_1^A,\ldots,B_n^A\rangle \ll_C I^A,\\ \vartheta^A=mgu^A(\langle B_1,\ldots,B_n\rangle,\\ \langle B_1^A,\ldots,B_n^A\rangle)\end{array}\right\}\right).$$

If $B'_1,\ldots,B'_n$ are in $\gamma_a(I^A)$, then $B_1^A(=\alpha_a(B'_1)),\ldots,B_n^A(=\alpha_a(B'_n))$ are in I^A because $\alpha_a(\gamma_a(I^A))=I^A$. By mgu^A definition and by Galois insertion:

$$mgu(\langle B_1,\ldots,B_n\rangle,\langle B'_1,\ldots,B'_n\rangle)\in$$
$$\gamma_\Phi(\alpha_\Phi(mgu(\langle B_1,\ldots,B_n\rangle,\langle B'_1,\ldots,B'_n\rangle))).$$

Since for any atom A, there exists θ such that: $A=\alpha_a(A)\theta$, we have:

$$\left\{A\vartheta\,\middle|\,\vartheta=mgu(\langle B_1,\ldots,B_n\rangle,\langle B'_1,\ldots,B'_n\rangle)\right\}\subseteq$$
$$\{A\vartheta'\,|\,\vartheta^A=mgu^A(\langle B_1,\ldots,B_n\rangle,\langle B_1^A,\ldots,B_n^A\rangle).$$
$$\vartheta'\in\gamma_\Phi(\vartheta^A)\}.$$

Moreover, by Galois insertion, it is trivial to prove that $\gamma_a(\alpha_a(A\vartheta^A))\supseteq\{A\vartheta'|\vartheta'\in\gamma_\Phi(\vartheta^A)\}$. Thus:

$$\left\{A\vartheta\,\middle|\,\vartheta=mgu(\langle B_1,\ldots,B_n\rangle,\langle B'_1,\ldots,B'_n\rangle)\right\}\subseteq$$
$$\gamma_a\left(\left\{\alpha_a(A\vartheta^A)\,\middle|\,\vartheta^A=mgu^A(\langle B_1,\ldots,B_n\rangle,\langle B_1^A,\ldots,B_n^A\rangle)\right\}\right).$$

Therefore, $T_P(\gamma_a(I^A))\subseteq\gamma_a(T_P^A(I^A))$ for all $I^A\in\wp(B_P^A)$. □

C. Proof of Lemma 4

We prove the lemma by showing the following.
$\{C|C=A\leftarrow B_1,\ldots,B_n\in lfp_c(U_P),A'\in body\_atoms(Q\cup\gamma_c(J^A)),\exists mgu(A,A')\}\subseteq\gamma_c(\{C^A|C^A=A^A\leftarrow B_1^A,\ldots,B_n^A\in lfp_c(U_P^A),A^{A'}\in body\_atoms(Q\cup J^A),\exists mgu(A^A,A^{A'})\})$.
Suppose $A\leftarrow B_1,\ldots,B_n$ in $lfp(U_P)$ satisfies the condition that A is in $body\_atoms(J_{query}\cup\gamma_c(J^A))$. The abstract clause $\alpha_c(A\leftarrow B_1,\ldots,B_n)$ is in $lfp(U_P^A)$ by Theorem 5. It also satisfies the condition that $\alpha_a(A)$ is in $body\_atoms(J^A_{query}\cup J^A)$ since $C^A=\alpha_c(\gamma_c(C^A))$ for any abstract clause $C^A\in J^A$ by Galois insertion $((\wp(lfp_c(U_P)),\subseteq),\alpha_c,(\wp(lfp_c(U_P^A)),\subseteq),\gamma_c)$. The Galois insertion can be easily proved. Therefore, $\gamma_c(D^A_{P,Q}(J^A))\supseteq D_{P,Q}(\gamma_c(J^A))$. □

REFERENCES

[1] P. Cousot and R. Cousot., "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points," In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, New York : ACM, 1977, pp. 238–252.

[2] R. Barbuti, R. Giacobazzi and G. Levi, "A general framework for semantics-based bottom-up abstract interpretation of logic programs," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 1, pp. 133–181, 1993.

[3] M. Codish, D. Dams and E. Yardeni., "Bottom-up abstract interpretation of logic programs," to appear in *Theoretical Computer Science*.

[4] M. Codish and B. Demoen, "Analysing logic programs using "prop"-ositional logic programs and a magic wand," In *Proceedings of the 1993 International Logic Programming Symposium*, MIT Press, 1993.

[5] M. Gabbrielli and R. Giacobazzi, "Goal independency and call patterns in the analysis of logic programs," In *Proceedings of ACM Symposium on Applied Computing*, ACM Press, 1994.

[6] T. Kanamori and T. Kawamura, "Analyzing success patterns of logic programs by abstract hybrid interpretation," ICOT Report TR-279, ICOT, Tokyo, Japan, 1987.

[7] T. Kanamori and T. Kawamura, "Abstract interpretation based on OLDT resolution," *Journal of Logic Programming*, vol. 15, pp. 1–30, 1993.

[8] K. Marriott and H. Sondergaard, "Bottom-up abstract interpretation of logic programs," In *Proceedings of the 5th International Conference on Logic Programming*, Cambridge, MA : MIT Press, 1988, pp. 733–748.

[9] T. Sato and H. Tamaki, "Enumeration of success patterns in logic programs," *Theoretical Computer Science*, vol. 34, pp. 227–240, 1984.

[10] M. H. van Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," *Journal of the ACM*, vol. 23, no. 4, 1976, pp. 733–742.

[11] B.-M. Chang, "Efficient Bottom-up Execution of Logic Programs using Compile-time Analysis," Ph.D. dissertation, Korea Advanced Institute of Science and Technology, Feb. 1994.

[12] P. Cousot and R. Cousot., "Abstract interpretation and application to logic programs," *Journal of Logic Programming*, vol. 13, 1992, pp. 103–179.

[13] M. Falaschi, G. Levi, C. Palamidessi and M. Martelli, "Declarative modeling of the operational behavior of logic languages," *Theoretical Computer Science*, vol. 69, pp. 289–318, 1989.

[14] J. Conery, "The AND/OR process model for parallel interpretation of logic programs," Ph.D. thesis, Univ. of California, Irvine, 1983.

[15] D. DeGroot, "Restricted AND-parallelism," In *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, Tokyo, 1984, pp. 471–478.

[16] S.W. Dietrich and D. S. Warren, "Extension tables: memo relations in logic programming," In *Proceedings of the 4th IEEE Symposium on Logic Programming*, Washington : IEEE Comp. Soc. Press, 1987, pp. 264–273.

[17] H. Tamaki and T. Sato, "OLD resolution with tabulation," In *Proceedings of the 3rd International Conference on Logic Programming*, Lecture Notes in Computer Science, vol. 239. Berlin : Springer-Verlag, 1986, pp. 84–98.

[18] K. Zhang, "A review of exploitation of AND-parallelism and combined AND/OR-parallelism in logic programs," *ACM SIGPLAN Notices*, vol. 29, no. 2, Feb. 1994, pp. 25–32.

[19] B.-M. Chang, K.-M. Choe and T. Han., "Efficient bottom-up execution of logic programs using abstract interpretation," *Information Processing Letters*, vol. 47, Sep. 1993, pp.149-157.

[20] R. Barbuti, M. Codish, R. Giacobazzi and M. Maher, "Oracle semantics for Prolog," in *Proceedings of International Conference of Algebraic and Logic Programming*, Lecture Notes in Computer Science 632, Springer-Verlag, 1992.

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

[22] J.-H. Chang, A. M. Despain and D. DeGroot, "AND-parallelism of logic programs based on static data dependency analysis," In *Proceedings of the 30th IEEE Computer Society International Conference*, 1985, pp. 218-226.

[23] D.-H. Kim, K.-M. Choe and T. Han, "Refined mark(s)-set-based backtrack literal selection for AND parallelism in logic programs," *Parallel Processing Letters*, vol. 2, pp. 71–79, 1992.

[24] N. Woo and K.-M. Choe, "Selecting the backtrack literal in the AND/OR process model," In *Proceed-*

ings of the 3th IEEE International Symposium on Logic Programming, Washington : IEEE, Comp. Soc. Press, 1986, pp. 200–210.

[25] K. Marriott and H. Sondergaard, “Bottom-up dataflow analysis of normal logic programs,” *Journal of Logic Programming*, vol. 13, pp. 181–204, 1992.

[26] R. Barbuti and R. Giacobazzi, “A bottom-up polymorphic type inference in logic programming,” *Science of Computer Programming*, vol. 19, no. 3, pp. 281–313, 1992.

[27] J. Bruynooghe and G. Janssens, “An instance of abstract interpretation integrating type and mode inferencing,” In *Proceedings of the 5th International Conference on Logic Programming*, Cambridge, Mass. : MIT Press, 1988, pp. 669–683.

Byeong-Mo Chang graduated from Seoul National University in 1988 with his B.S. degree in computer engineering. He received the M.S. and Ph.D. degrees in computer science from Korea Advanced Institute of Science and Technology in 1990 and 1994, respectively. Since 1994, he has been working on parallelizing compilers as a Postdoctoral Researcher in ETRI. His research interests include parallelizing compilers, abstract interpretation and logic programming.

Kwang-Moo Choe received the B.S. degree in electronic engineering from Seoul National University in 1976, and the M.S. and Ph.D. degrees in computer science from Korea Advanced Institute of Science and Technology(KAIST) in 1978 and 1984, respectively. From 1984 to 1990, he was an Assistant Professor of KAIST, and has been an Associated Professor of KAIST since 1990. From 1985 to 1986, he was a Member of Technical Staff at AT & T Bell Laboratories. His research interests include compiler construction and logic programming.

Roberto Giacobazzi received the Ph.D. degree in computer science from University of Pisa, Italy, March 1993. He has been a Visiting Researcher at LIX, École Polytechnique, France, since September 1993. His research interests include semantics and abstract interpretation of logic programs.