논리 프로그램의 두 단계 추상 해석 틀
(A Two-phase Abstract Interpretation Framework of Logic Programs)

창 병 모 †
(Byeong-Mo Chang)

요 약 추상 해석은 Cousot와 Cousot에 의해서 제안된 시맨틱스에 기초한 데이터 흐름 분석 틀이다 본 논문에서는 논리 프로그램을 위해서 상향 분석과 하향 분석으로 이루어진 두 페이즈 추상 해석 틀을 제안한다. 이 틀에 기초한 분석은 질의에 관련된 절의 모든 성공 패턴에 대한 근사값을 제공한다. 제안된 틀에 기초한 분석 예로서 타입 분석 방법을 제시한다. 또한 제안된 틀에 기초한 분석 결과를 이용하여 논리 프로그램의 수행을 최적화 하는 예들을 제시한다.

Abstract Abstract interpretation is a semantics-based data-flow analysis framework proposed by Cousot and Cousot. This paper provides a two-phase abstract interpretation framework, which consists of a bottom-up phase followed by a top-down phase. Analyses based on this framework give an approximation of all success patterns of clauses relevant to a query. A type analysis is given as an example analysis based on the framework. Optimization techniques for improving execution of logic programs are given, which use the results of analyses based on the framework.

1. Introduction

Abstract interpretation is a semantics-based data-flow analysis framework proposed by Cousot and Cousot [9]. Abstract interpretation of a program approximates its standard semantics by fixpoint execution over an abstract domain rather than a possibly infinite concrete domain [9]. An abstract domain is assumed to be typically a complete lattice which is finite. Abstract interpretation provides a safe and finite approximation of some runtime behavior of the program. For instance, a number of abstract interpretation have been proposed to approximate success patterns of atoms or clauses [1, 6, 15, 16, 18, 19]. The ones in [15, 16, 19] are based on top-down execution while the ones in [1,6,18] based on bottom-up execution.

· 본 연구는 한국과학재단 특정기초 연구비 지원에 의해수행되었음 (과제번호 95-0100-54-3)
† 정 회 원 숙명여자대학교 전산학과 교수
논문접수 1995년 6월 12일
심사완료 1996년 4월 4일

Recently, bottom-up abstract interpretation of logic programs has gained much attention. The analysis is based on the least fixpoint computation of the standard operator T_P as introduced by van Emden and Kowalski in [21]. This approach has been considered in [1,6,18] as a basis for bottom-up data-flow analysis of logic programs. It provides an approximation of success patterns by evaluating an abstract version of T_P over a (possibly simpler) abstract domain.

In this paper we provide a *two-phase abstract interpretation framework*, which consists of a bottom-up analysis followed by a top-down one. The framework is based on clauses rather than atoms, since incorporating abstract clauses (abstraction of clauses) allows more intelligence than abstract atoms, when they are applied to *improve real execution*. Analyses based on this two-phase framework give an approximation of success patterns of clauses *relevant* to a given query. In the design of the framework, the

bottom-up analysis is designed by extending the bottom-up abstract interpretation in [1] 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 analysis is able to handle possibly non ground syntactic objects since it is based on the concrete semantics in [13]. The top-down analysis is designed to collects abstract clauses *relevant* to a given query from the result of the bottom-up analysis, so it is *query dependent*. We give a type analysis based on the framework by considering the depth k abstraction in [19] as abstraction function.

As applications of the two-phase framework, we give examples to improve (top-down, bottom-up, parallel) execution models of logic programs. This paper has a contribution in the sense that it provides a *framework* for two-phase abstract interpretation and its practical use, i.e improving execution models. Moreover, the proposed approach has an advantage over top-down analyses [15,16,19] 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 analysis. Only additional top-down *collecting* analysis over the result of the bottom-up analysis is necessary for a different query. This is a good property, since many different queries are usually asked over the same program.

The paper is structured as follows. The next section gives preliminaries on logic programs and abstract interpretation. Section 3 describes a two-phase abstract interpretation framework. Section 4 provides applications of the framework. Section 5 provides related works and Section 6 concludes this paper.

2. Preliminaries

2.1 Basic Notations

Let (Π, Σ, Var) denote a first order language, namelya (finite) set Π of predicate symbols, a set Σ of function symbols, and a denumerable set Var of variables. With each function symbol $f \in \Sigma$ and predicate symbol $p \in \Pi$ is associated a unique natural number called its *arity*: (predicate or function) symbol f with arity n is written f/n. The set of all terms constructed from Σ and V 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 by $Atom$ the set of all atoms constructed from Π and $Term(\Sigma, Var)$. *A goal* Q is a (possibly empty) sequence of atoms, and is typically written $< a_1, \ldots, a_n >$ or simply as $a_1, \ldots, a_n$. 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 by h and called a *fact*; otherwise it is called a *rule*. Rules are sometimes denoted by the Greek letter δ. A logic program is a finite set of clauses. A logic program P with a query Q is denoted by P_Q. The set of clauses constructed from elements of $Atom$ is denoted $Clause(\Pi, \Sigma, Var)$ or $Clause$ for short.

The set of variables occurring in a syntactic object t is denoted by $var(t)$. A *substitution* $\theta(x)$ is a mapping from Var to $Term$, such that $\{x \in Var \mid \theta(x) \neq x\}$ is finite. It extends to apply to any syntactic object in the usual way. The identity substitution is denoted by ε. The set of idempotent substitutions is denoted by Sub. Following tradition, the application of a substitution θ to an object t will be written $t\theta$. We fix a partial function mgu: $Atom \times Atom \rightarrow Sub \cup \{fail\}$ 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 to $mgu: Atom^* \times Atom^* \rightarrow Sub \cup \{fail\}$ 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 $mgu(<>,<>) = \varepsilon$. 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 anelement $a \in S$, $[a]_\sim$ denotes the equivalence class of a with respect to

$\sim$. We let $\wp(S)$ denote the power set of a set S.

2.2 Concrete Semantics

As 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, and analyzing 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 [21] 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 *extended Herbrand universe* U_P be $Term(\Sigma, Var)/\sim$, where $\sim$ is the *variance* relation (i.e. $t_1 \sim t_2$ iff $\exists \upsilon_{1\ \upsilon_2} \mid t_1\upsilon_1 = t_2 \wedge t_2\upsilon_2 = t_1$).

The variance relation can be extended easily to any syntactic object (atoms, clauses etc.). The *extended Herbrand base* B_P is $Atom/\sim$ 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 the set of its ground instances 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 apartfrom s and from each other, namely,

1. $[a_i]\sim \in I$;
2. $var(a_i) \cap var(s) = \phi$, for $1 \le i \le n$ and
3. $i \neq j$ implies $var(a_i) \cap var(a_j) = \phi$, for $1 \le i, j \le n$.

For simplicity of exposition, we will abuse notation and assume that anatom represents its equivalence class and write $[a]\sim$.

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

$$T_P(I) = \left\{ A\vartheta \;\middle|\; \begin{array}{l} c = h \leftarrow a_1, \ldots, a_n \in P \\ \langle a'_1, \ldots, a'_n \rangle \ll_c I \\ \vartheta = mgu(\langle a_1, \ldots, a_n \rangle, \langle a'_1, \ldots, a'_n \rangle) \end{array} \right\}$$

The semantics of a program P is determined by $lfp(T_P) = T_P{}^{w}(\phi)$[1], the least fixpoint of T_P[13].

2.3 Bottom-up Abstract Interpretation Framework

In this subsection we review the bottom-up abstract interpretation framework in [1]. Before that, we first summarize the standard abstract interpretation framework in [9]. This framework presupposes a least fixpoint characterization of the collecting semantics. We assume that a concrete interpretation for aprogram 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.

Definiion 2 ([9]) *An abstract interpretation $((E, \sqsubseteq), E_P, (D, \preceq), D_P, \alpha, \gamma)$ consists of a complete lattice $(E, \sqsubseteq)$, a monotonic operator $E_P : E \to E$, a complete lattice $(D, \preceq)$, a monotonic operator $D_P : D \to D$, an abstraction function $\alpha : E \to D$ and a concretization function $\gamma : D \to E$, 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 *correctness* criterion that ensures that D_P faithfully mimics E_P.

---

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

Proposition 1 ([9]) *If* $((E,\sqsubseteq),E_P(D,\preceq),D_P,\alpha,\gamma)$ *is an abstract interpretation, then* $lfp(E_P)\sqsubseteq\gamma(lfp(D_P))$

In [1] the previous collecting semantics has been applied to the analysis of successful computations in definite logic programs This results in a simple bottom-up abstract interpreter. The idea is to consider a pair of Galois insertions for abstractly describing semantic objects: $((\wp(B_P),\subseteq),\alpha,(H,\sqsubseteq),\gamma)$ and $((\wp(Sub),\subseteq),\alpha_s,(S,\sqsubseteq),\gamma_s)$, respectively corresponding to interpretation abstraction and substitution abstraction. In the sequel, we assume that H is a finite set The abstract analysis is defined in terms of a pair of basic operators:

1. $apply^A : Atom \times S \rightarrow H$, and
2. $mgu^A : Atom^* \times H \rightarrow S$

respectively performing abstract substitution application and abstract unification of a (concrete) goal into an abstract interpretation. Each operator is assumed to be equipped with the two conditions, correctness and monotonicity, see [1] for more details. The abstract operator $T_P^A : H\rightarrow H$ is defined to approximate T_P, such that for each $I^A\in H$:

$$T_P^A(I^A) = \sqcup_{A,B_1,\ldots,B_n\in P} apply^A(A, mgu^A(\langle B_1 \ldots B_n\rangle, I^A))$$

The monotonicity of the basic abstract operators ensures that T_P^A is monotonic. For finite abstract domains, there exists $n<\omega$ such that $lfp(T_P^A) = (T_P^A)^n(\emptyset)$.

Theorem 1 ([1]) *Let P be a logic program.* $lfp(T_P^A) = (T_P^A)^n(\emptyset)$ *for some finite n.*

Moreover, the correctness of the analysis (i.e. $T_P(\gamma(I^A))\sqsubseteq\gamma(T_P^A(I^A))$ for each $I^A\in H$), is a consequence of the relative (local) correctness of the basic operators

Theorem 2 ([2]) $\gamma(lfp(T_P^A))\sqsupseteq lfp(T_P)$ *for any program P.*

We now give an instance of the framework based on the depth k abstraction This framework can deal with different abstract interpretations, provided the soundness conditions specified in [1] are satisfied.

Definition 3 ([19]) *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 [19] 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 $Atom^A/_\sim$ where

$$Atom^A = \{p(t_1^A,\ldots,t_n^A) \mid p/n \in \Pi, \{t_1^A,\ldots,t_n^A\}\subseteq U_P^A\},$$

and $\sim$ is the variance relation on atoms. The abstraction function $\alpha : B_P\rightarrow B_P^A$ is defined by $\alpha(p(t_1,\ldots,t_n)) = [\ p(\rho(t_1),\ldots,p(\rho(t_n))]_\sim$. The equivalence class is represented as $a^A = [\ p(\rho(t_1,\ldots,\rho(t_n))]_\sim$. The abstraction function is lifted to interpretations by defining $\alpha : \wp(B_P)\rightarrow\wp(B_P^A)$ such that $\alpha(I) = \{\alpha(a)\mid a\in I\}$. The corresponding concretization function $\gamma : \wp(B_P^A)\rightarrow\wp(B_P)$ is $\gamma(I^A) = \{a'\in B_P \mid a^A\in I^A, \alpha(a') = a^A\}$. It is shown in [1] that $((\wp(B_P),\subseteq),\alpha,(\wp(B_P^A),\subseteq),\gamma))$ is a Galois insertion.

An *abstract substitution* is a mapping from a given finite set of variables $V\subseteq Var$ into U_P^A. Given a substitution $\upsilon = \{t_1/x_1,\ldots,t_n/x_n\}$, α_ϕ is defined by $\alpha_\phi(\upsilon) = \upsilon^k$ where $\upsilon^k = \{\rho(t_1)/x_1,\ldots,\rho(t_n)/x_n\}$. A concretization mapping can be associated with α_ϕ γ_ϕ such that $(\alpha_\phi,\gamma_\phi)$ is a Galois insertion. The abstract unification function is defined by

$$mgu^A(\langle a_1,\ldots,A_n\rangle,\langle a_1^A,\ldots,a_n^A\rangle) = \alpha_\phi(\vartheta), \text{ where}$$
$$\vartheta(\langle a_1,\ldots,a_n\rangle,\langle a_1^A,\ldots,a_n^A\rangle).$$

The abstract unification function satisfies the

correctness and monotonicity conditions [1]. The abstract apply function $apply^A$ is defined by $apply^A(A, \vartheta^A) = \alpha(A\vartheta^A)$. A specialized version of the abstract operator T_P^A on the depth k abstraction can be defined using the above functions.

The monotonicity of the basic abstract operators ensures that T_P^A is monotonic, and because the abstract domain is finite, there exists $n < \omega$ such that $lfp(T_P^A) = (T_P^A)^n(\varnothing)$. Moreover, the correctness of the analysis (i.e. $T_P(\gamma(I^A)) \subseteq \gamma(T_P^A(I^A))$ for each $I^A \in H$), is a consequence of the relative (local) correctness of the basic abstract operators.

Example 1 Let P be the following logic program

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

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

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

3. Two-phase Abstract Interpretation Framework

The static analysis technique in [1] 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 order to prune 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 framework, collecting only those abstract descriptions relevant to the query. We design a two-phase abstract interpretation by incorporating abstract clauses, because abstract clauses in the abstract interpretation allows more intelligence than abstract atoms, when they are applied to improve execution models. The two-phase abstract interpretation consists of a bottom-up analysis followed by top-down one. The bottom-up analysis is designed by extending the basic bottom-up abstract interpretation in [1] so as to incorporate abstract clauses. 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.

3.1 Two-phase Framework

We first present two kinds of concrete collecting semantics for the bottom-up analysis and top-down analysis in the two-phase abstract interpretation The semantics is based on *clauses* rather than atoms, so we define an interpretation as follows Let C be the set of (equivalence classes of) clauses up to renaming: $Clause/_{\sim}$ where $\sim$ is the variance relation extended on clauses.

Definition 4 ([7]) *An* interpretation *is any element in* $Int = \wp(C)$.

The concrete semantics for the bottom-up analysis is defined in terms of an operator U_P, which infers success instances of the program clauses, given an interpretation (a set of clauses), while the standard T_P fixpoint operator infers success instances of atoms given an interpretation (a set of atoms).

Definition 5 *Let P be a logic program. Let U_P : $Int \rightarrow Int$ be such that*

$$U_P(I) = \left\{ [\ c\sigma]_{\sim} \;\middle|\; \begin{array}{l} c = h \leftarrow a_1, \ldots, a_n \in P \\ \langle h_1 \leftarrow \overline{b_1}, \ldots, h_n \leftarrow \overline{b_n} \rangle \ll_c I, \\ \sigma = mgu(\langle a_1, \ldots, a_n \rangle, \langle h_1, \ldots, h_n \rangle) \end{array} \right\}$$

The bottom-up concrete semantics of a program P is determined by $lfp(U_P)$, and it is computed by $U_P^{\omega}(\varnothing)$ since U_P is monotonic and continuous

The concrete collecting semantics for the top-down analysis is defined in terms of an

operator D_{P_Q}. In the collecting semantics, the operator collects, from the bottom-up collecting semantics, a set of clauses relevant to the given query Q, where a clause $h \leftarrow a_1, \ldots, a_n \in lfp(U_P)$ is said to be *relevant* to a query Q if

1. h is unifiable with an atom in Q, or
2. h is unifiable with a body atom of a clause relevant to Q.

Definition 6 Let P_Q *be a logic program with a query* Q. *Consider a function body which an interpretation* I *to the set of the body atoms of clauses in* I. *Then* $D_{P_Q}\ p(lfp(U_P)) \rightarrow p(lfp(U_P))$ *is defined by*

$$D_{P_Q}(I) = \left\{ [\ c\ ]\ \sim \;\middle|\; \begin{array}{l} [\ c\ ]\ \sim = [\ h \leftarrow \overline{b}\ ]\ \sim \in lfp(U_P), \\ mgu(h, a) \neq fail \text{ for some } [\ a\ ]\ \sim \\ \in body(Q \cup I) \end{array} \right\}$$

The top-down concrete semantics of P_Q is determined by $lfp(D_{P_Q})$ and it is computed by $D^{\omega}_{P_Q}(\emptyset)$ since D_{P_Q} is monotonic and continuous.

We assume the standard framework of abstract interpretation in terms of a Galois insertion. In the followin, let $(ASub, \sqsubseteq)$ be a complete lattice of abstract substitutions, such that $(p(Sub), \alpha_S, ASub, \gamma_S)$ be a Galois insertion. We will construct a domain *AInt* of abstract interpretations, which is a complete lattice of *abstract interpretations*, such that each abstract interpretation describes a set of clauses, and $(Int, \alpha, AInt, \gamma)$ be a Galois insertion.

Let $LC = Clause(\Pi, \Sigma, Var)/\sim$ be the set of flat clauses, i.e syntactic oojects with no function and constant, where no variable occurs more than once. We onstruct an abstract domain *AInt* of abstract interpretations by associating each representative of LC with an abstract substitution that describes a set of its instances, as in [7]. An *abstract clause* is a pair in $LC \times ASub$, and an abstract domain is $AInt = p(LC \times ASub)/_{\equiv}$ where $\equiv$ is the equivalence relation induced by a relation $\preceq$ on $p(LC \times ASub)$: $I_1^A \preceq I_1^A$ *iff* $\gamma(I_1^A) \subseteq \gamma(I_2^A)$.

Definition 7 Let $(p(Sub), \alpha_S, ASub, \gamma_S)$, *be a domain of abstract substitutions. The induced domain of abstract interpretation* $(Int, \alpha, AInt, \gamma)$ *is constructed as follows:*

1. *The function* $\gamma: p(LC \times ASub) \rightarrow Int$ *is defined by* $\gamma(I^A) = \{ [\ c\theta\ ]\ \sim \mid \langle c, k \rangle \in I^A, \theta \in \gamma_s(k) \}$
2. *The function* α: $Int \rightarrow p(LC \times ASub)$ *is defined by* $\alpha(I) = \{ \langle c, k \rangle \mid c \in LC, k = \alpha_s(\{ mgu(c, c') \mid c' \ll_c I \})$
3. $AInt = p(LC \times ASub)/_{\equiv}$ *where* $\equiv$ *is the equivalence relation induced by a relation* $\preceq$ *on* $p(LC \times ASub)$: $I_1^A \preceq I_1^A$ *iff* $\gamma(I_1^A) \subseteq \gamma(I_2^A)$, *and the corresponding partial order on AInt is denoted by* $\sqsubseteq$.
4. *The function* γ *is lifted to* $AInt \rightarrow Int$ *by taking* $\gamma([\ I^A\ ]) = \gamma(I^A)$, *and the function* α *is lifted to* $Int \rightarrow AInt$ *by taking* $\alpha(I) = [\ \alpha(I)\ ]$.

If $(p(Sub), \alpha_S, ASub, \gamma_S)$ is a Galois insertion, then so is the induced domain of abstract interpretation $(Int, \alpha, AInt, \gamma)$. This can be proved in a similar way to [6].

Definition 8 *An* abstract interpretation *is any element in AInt.*

We now consider the two-phase abstract interpretation framework. For the bottom-up analysis, we construct an abstract interpretation $((Int, \subseteq), U_P, (AInt, \sqsubseteq), U_P^A, \alpha, \gamma)$.

Definition 9 ([7]) *An abstract unification function* $mgu^A : (Atom^* \times ASub) \times (Atom \times ASub)^* \rightarrow ASub \cup \{fail\}$ *is said to be* correct *if it satisfies the condition that if*

(1) $k = mgu^A(\langle\langle a_1, \ldots, a_n \rangle, k_0 \rangle, \langle\langle H_1; k_1 \rangle, \ldots, \langle h_n, k_n \rangle\rangle), \theta_i \in \gamma_S(k_i), (1 \le i \le n)$ and

(2) $\theta = mgu(\langle a_1, \ldots, a_n \rangle \theta_0, \langle h_1\theta_1, \ldots, h_n\theta_n \rangle)$ *then* $\theta \in \gamma_S(k)$.

In the sequel, we assume that mgu^A is *correct* and *monotonic*. The bottom-up abstract analysis is formalized in terms of an abstract operator U_P^A.

Definition 10 Let P be a logic program. Let U_P^A : $AInt \rightarrow AInt$ be such that

$$U_P^A(I^A)=\sqcup\left\{\langle c;k\rangle \;\middle|\; \begin{array}{l}\langle c=h\leftarrow a_1,\ldots,a_n;k_0\rangle\\ \langle\langle h_1\leftarrow \overline{b_1},k_1\rangle,\ldots,\langle h_n\leftarrow \overline{b_n},k_n\rangle\rangle \ll_c I^A\\ k=mgu^A(\langle\langle a_1,\ldots,a_n\rangle;k_0\rangle,\langle\langle h_1\rangle,k_1\rangle,\\ \quad\ldots,\langle h_n;k_n\rangle\rangle)\end{array}\right\}$$

U_P^A can be easily shown to be monotonic. The bottom-up abstract analysis determines the least fixpoint $lfp(U_P^A)$. If the abstract domain is finite, then the least fixpoint is computed in finite time, i.e., $lfp(U_P^A)=(U_P^A)^n(\perp)$ for some finte n, since U_P^A is monotonic. Moreover, it is also proved in a similar context that the least fixpoint is computed also in finite time when *ASub* is finite [6] The correctness of the bottomup analysis is a consequence of the correct abstract unification function mgu^A and the Galois insertion $(Int,\ \alpha,\ AInt,\ \gamma)$

Theorem 3 *For any logic program P,* $lfp(U_P)\subseteq\gamma(lfp(U_P^A))$.

For the top-down analysis, we construct an abstract interpretation

$$((\wp(lfp(U_P)),\subseteq),D_{P_Q},(\wp(lfp(U_P^A)),\subseteq),D_{P_Q}^A,\alpha,\gamma).$$

The top-down abstract analysis collects, from the result of bottom-up abstract analysis, the set of the abstract clauses *relevant* to the given query Q. An abstract clause $< h \leftarrow a_1, \ldots, a_n ; k > \in lfp(U_P^A)$ is said to be *relevant* to a given query Q if

1. $\langle h,k\rangle$ is abstractly unifiable ($mgu^A \neq fail$) with an atom in Q, or
2. $\langle h,k\rangle$ is abstractly unifiable ($mgu^A \neq fail$) with a body atom of an abstract clause relevant to Q.

The top-down abstract analysis is formalized in terms of an abstract operator D_P^A

Definition 11 *Let P_Q be a program with a query Q. Consider a function body which maps an abstract interpretation I^A to the set of the body atoms of abstract clauses in I^A. Then $D_{P_Q}^A : \wp(lfp(U_P^A))\rightarrow\wp(lfp(U_P^A))$ is defined by*

$$D_{P_Q}^A(I^A)=\sqcup\left\{\langle c^A\rangle \;\middle|\; \begin{array}{l}c^A=\langle h\leftarrow\overline{b},k\rangle\\ mgu^A(\langle h;k\rangle,\langle a,\hat{k}\rangle)\neq fail\ for\ some\\ \langle a,\hat{k}\rangle\in body(\alpha(Q)\cup I^A)\end{array}\right\}$$

$D_{P_Q}^A$ can be easily shown to be monotonic. The top-down abstract analysis determines the least fixpoint $lfp(D_P^A)$. Since the abstract domain $\wp(lfp(U_P^A))$ is finite, and $D_{P_Q}^A$ is monotonic, then the least fixpoint is computed in finite time, i.e., $lfp(D_P^A)=(D_P^A)^n(\perp)$ for some finite n. The correctness of the top-down analysis is a consequence of the correct abstract unification function mgu^A and the Galois insertion $(\wp(lfp(U_P)),\ \alpha\ ,\wp(lfp(U_P^A)),\gamma)$.

Theorem 4 *For any logic program P and query Q,* $lfp(D_{P_Q})\subseteq\gamma(lfp(D_{P_Q}^A))$.

3.2 An Instance of Two-phase Framework

We now give an instance of the framework by considering the depth k abstraction as an abstraction function. The set of depth k *abstract clauses* is defined by $C^A=\{\alpha(c)|c\in C\}$ where for any clause $h\leftarrow a_1,\ldots,a_n$, $\alpha(h\leftarrow a_1,\ldots,a_n)$ is $[\ \hat{\alpha}(h)\leftarrow\hat{\alpha}(a_1),\ldots,\hat{\alpha}(a_n)]$, where $\hat{\alpha}(a)$ maps an atom into its depth k abstract atom. An depth k abstract clause is just denoted by $c^A=h^A\leftarrow a_1^A,\ldots,a_n^A$, since it can be easily transformed into an abstract clause in $LC\times ASub$. For the bottom-up analysis, we define an abstract interpretation $((\wp(C),\subseteq),U_P,(\wp(C^A),\subseteq),U_P^A,\alpha,\gamma)$ based on the depth k abstraction It is clear that the abstract domain for the bottom-up analysis constitutes a Galois insertion $((\wp(C),\subseteq),\alpha,(\wp(C^A),\subseteq),\gamma)$ where the abstraction function α is lifted on an interpretation (a set of clauses), and the concretization function is defined by $\gamma(I^A)=\{c\mid c^A\in I^A,\alpha(c)=c^A\}$. The domain of abstract substitutions based on the depth k abstraction is defined in Section 2.3 as $(Sub,\ \alpha_\phi,$ ASub, $\gamma_\phi)$ and the abstract unification function based on the depth k abstraction is defined as :

$$mgu^A((a_1,\ldots,a_n),(a_1^A,\ldots,a_n^A))=\alpha_\phi(\vartheta),$$
$$where\ \vartheta=mgu(a_1,\ldots,a_n),(a_1^A,\ldots,a_n^A)$$

It can be easily shown that mgu^A is an *correct*

and *monotonic* abstract unification function. The bottom-up abstract analysis based on the depth k abstraction is the least fixed point $lfp(U_P^A)$ which provides an depth k approximation of the success patterns of each (program) clause. The bottom-up abstract analysis can be computed in finite time because 1. the operator U_P^A is monotonic as the abstract unification function mgu^A is monotonic, and 2 the abstract domain $\wp(C^A)$ is finite. The correctness of the bottom-up analysis, i.e. $\gamma(U_P^A(I^A)) \supseteq U_P(\gamma(I^A))$, is a consequence of the correct abstract unification function mgu^A and the underlying Galois insertion.

Example 2 *When we consider the depth 2 abstraction, $lfp(U_P^A)$ for the program in Example 1 is*

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

The bottom-up analysis computes an depth k approximation of the success patterns of clauses 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 abstract analysis collects a subset of $lfp(U_P^A)$, which consists of the depth k abstract success patterns of program clauses relevant to the query. The top-down abstract analysis determines the least fixpoint $lfp(D_{P_Q}^A)$ which contains all the depth k abstract clauses in $lfp(U_P^A)$ which are relevant to the query. The top-down abstract analysis is finite, because

1. the operator D_P^A is monotonic as the abstract unification function mgu^A is monotonic, and
2. the abstract domain $\wp(lfp(U_P^A))$ is finite.

The correctness of the top-down phase, i.e $\gamma(D_{P_Q}^A(I^A)) \supseteq D_{P_Q}(\gamma(I^A))$, is a consequence of the correct abstract unification function mgu^A and the underlying Galois insertion.

Example 3 *Consider a query of the form p(a,Z).*

The result of the top-dosn analysis for the program P in Example 2 is as follows

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

4. Applications

There have been several researches for optimizing execution of logic programs based on the framework.

First, an optimization technique for bottom-up execution of logic programs is presented in [2,3], in which the bottom-up execution based on system graphs is improved using abstract filters which are construced from the result of an analysis based on the framework. Unrelevant derivations to the given query are eliminated during bottom-up execution by the power of abstract filters and the total number of derivations in the naive execution, the static filtering , and the abstract filtering are compared by experiments. The experiment uses the depth k abstract domain.

Second, the two-phase abstract analysis can be applied to improve the linear execution as in Prolog [2,17]. The linear execution model can be improved. 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. In particular it allows usto 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 the query 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. If the subgoal passes the call check and succeeds, the exit check is tested for the subgoal with the answer substitution being applied. The total number of calls in the naive linear execution, and the improved linear execution are compared by simulation in [17].

Third, the two-phase abstract analysis can be easily applied to improving other top-down (parallel) execution models such as [5,8,11,12,20]. As an example, we consider them *AND/OR Process Model* [8] 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. Similar to the new linear execution model, AND processes eliminate unnecessary child OR processes by testing the call check condition whenever an OR process is created and testing the exit check condition whenever the OR process succeeds.

5. Related Works

There are several related works for analyzing call patterns of logic programs. A framework for abstract interpretation is also provided based on OLDT resolution in [16], and Kanamori uses depth k abstraction in [15] in which depth k success patterns are computed based on OLDT resolution [20]. Gabbrielli and Giacobazzi provides an abstract bottom-up semantics for goal independent analysis of call patterns [14]. The abstract bottom-up semantics gives the call patternsfor every possible query, so the call patterns for any goal can be collectedfrom the abstract semantics. Codish, Dams and Yardeni characterize call patterns by a transformational approach in [6]. A program and and a goal G are transformed by a *magic set*-like transformation such that every call pattern of G in P is a success pattern. Call patternscan hen be obtained from the bottom-up denotation of thetransformed program. This provides the call patterns for a given goal.The above approaches approximate call patterns from a given query, while the proposed approach approximate success patterns relevant toa give query.

Even though some two-phase analysis can be designed with considering only abstract atoms as in [10], incorporating abstract clauses in abstract interpretation allows more intelligence than abstract atoms when they are applied to improve execution models. If we consider some two-phase analysis which considers abstract atoms,the improved execution model has to check a call with all abstract atomswhich 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 the corresponding abstract clauses as in Theorem 5. 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.

6. Conclusion

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. As well as the proposed analysis framework can be applied to othercompile-time optimizations, it can be easily applied to any safe approximation such as ground dependencies, sharing, type graph etc. This information can also be used to improve the efficiency of top-down executions, following the same model described before.

References

[1] 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*, vol. 15, no 1, pp 133--181, 1993.

[2] Chang, B.-M., Efficient Bottom-up Execution of Logic Programs using Compile-time Analysis. PhD dissertation, Dept of Computer Science, KAIST, Feb. 1994.

[3] Chang, B.-M., Choe, K.-M., and Han., T. Efficient

bottom-up execution of logic programs using abstract interpretation, *Information Processing Letters*, Vol. 47, Sep. 1993, pp.149-157.

[4] Chang, J.-H., Despain, A. M., and DeGroot, D. 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.

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

[7] Codish, M., Debray, S., and Giacobazzi, R. Compositional analysis of modular logic programs. In *Proceedings of the 20-th ACM Symposium on Principles of Programming Languages*, ACM, New York, 1993, pp451--464.

[8] Conery, J. The AND/OR process model for parallel interpretation of logic programs. PhD thesis, Univ. of California, Irvine, 1983.

[9] Cousot, P., and Cousot., R. 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*, ACM, New York, 1977, pp. 238--252.

[10] Cousot, P., and Cousot., R. Abstract interpretation and application to logic programs, *Journal of Logic Programming*, Vol 13, 1992, pp.103--179.

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

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

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

[14] Gabbrielli, M. and Giacobazzi, R. Goal independency and call patters in the analysis of logic programs In *Proceedings of the 1994 ACM Symposium on Applied Computing*, 1994, pp. 394--399.

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

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

[17] Kim, M., An implementation and analysis of logic program evaluators usingtwo-phase abstract interpretation, MS Thesis, Department of Computer Science, KAIST, Feb. 1995.

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

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

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

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

창 병 모

1988년 서울대학교 컴퓨터공학과 졸업(학사). 1990년 한국과학기술원 전산학과에서 공학석사 학위 취득. 1994년 한국과학기술원 전산학과에서 공학박사 학위 취득. 1994년 ~ 1995년 한국전자통신연구소 박사후 연수 연구원. 1995년 ~ 현재 숙명여자대학교 전산학과 조교수. 관심분야는 컴파일러 구성론(정적 분석, 코드 최적화), 논리 프로그래밍, 연역 데이타베이스