Modular Logic Programs over Finite Domains

Maurizio Gabbrielli Roberto Giacobazzi Danilo Montesi

CWI,
P.O. Box 4079,
1009 AB Amsterdam,
The Netherlands.
gabbri@cwi.nl

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
Italy
(giaco,montesi)@di.unipi.it

Abstract

In this paper we study the properties of a compositional semantics for logic programs and its applications to modular analysis and logic-based databases, focusing on programs defined on a finite domain (i.e. on a function free finite signature). By this restriction we obtain a finite characterization of the compositional semantics which has the same correctness and compositionality properties of the original (fixpoint-based) definition, even if the latter could introduce infinite denotations also for finite domains programs. This property is particularly suitable for semantics-based program analysis, since it allows to identify a class of abstract domains for which the same finite characterization of the (abstract) compositional semantics holds. By combining this result with an existing framework for the modular analysis of logic programs, we can obtain a compositional analysis without requiring an additional level of abstraction. The finiteness of our characterization can also be useful for the modular construction and the uniform compilation of Datalog programs.

1 Introduction

Module systems and data abstraction are powerful methods for managing the complexity of large programs. Recently, a new semantics for modular logic programs has been introduced in [5]. The essential idea is to enhance the semantics proposed in [14] by allowing clauses in the semantic domain. The meaning of a module is given by the least fixpoint $\mathcal{F}_\Omega(P)$ of a suitable operator T_P^Ω. This provides a semantics which is compositional wrt union of modules, still maintaining the simplicity of the standard T_P-based semantics introduced by van Emden and Kowalski. From the point of view of applications, a drawback of such a fixpoint semantics is that since clauses are used as semantics objects, $\mathcal{F}_\Omega(P)$ can be an infinite set even if considering programs defined over a *finite domain* (i.e. a function-free finite signature).

In this paper we notice that this restriction allows to obtain a *finite* characterization of the compositional semantics of a program. This is obtained by taking as semantics of P the result of a finite number of iterations of the operator T_P^Ω, without reaching the fixpoint. While this would not be correct in general, we show that for finite domain programs the resulting denotation has the same compositionality and correctness properties of the fixpoint semantics. This is particularly suitable for semantics-based program analysis and logic-based databases. In fact, when considering abstract interpretation based on $\mathcal{F}_\Omega(P)$, there are two independent dimensions along which we need finite descriptions in the abstract domain. The first one is the usual finite abstraction on the (possibly) infinite set of descriptions of the concrete computations which depends on the particular property analyzed. For the case of logic programs, concrete observables can be expressed by (infinite sets of) substitutions and their abstraction is provided by the standard approaches to logic program analysis ([8, 3, 6, 12]). The second one is a finite description of sequences of atoms. Indeed, even if the set of abstract substitutions is finite, the abstract version of the compositional fixpoint semantics may introduce arbitrary large clauses in the abstract semantics, so that the analysis may not terminate. In [9] this problem has been tackled by introducing an additional layer of abstraction which is obtained by applying fixpont acceleration techniques such as the so called $\star$-*abstraction*. This is applied to provide finitary descriptions for arbitrary large clauses, thus introducing a further approximation which make the analysis less precise. While this is needed to handle abstractions over generic (possibly infinite) abstract domains, we will show that there exists a wide class of abstract domains for which a finite description of the semantics is obtainable without a further level of abstraction. This corresponds precisely to consider compositional semantics over finite domains.

The finite characterization of the compositional semantics can be useful also for Datalog programs. A Datalog program ([7]) is a particular logic program over a function-free signature which consists of two components: a set of Horn clauses (the *intensional database*) which are used to derive new facts and the *extensional database* storing the information in terms of ground unit clauses. Compilation techniques are often used to optimize the *query answering process*. Since there are no uniform compilation methods, usually the intensional database is partitioned into syntactic classes of clauses (e.g. linear, regular, etc.) and each class is compiled by using a specific technique. The compositional semantics $\mathcal{F}_\Omega(P)$ provides a way to translate a recursive program into a recursion-free one. Our results give a finite characterization of this process for Datalog programs (on a finite domain), thus providing a *uniform* modular compilation technique for intensional databases. The mentioned compositionality furnishes also the semantic base to develop deductive databases in an incremental and modular way.

The remaining of this paper is organized as follows. After some preliminaries, in the next section we introduce the compositional semantics which we are referring to. Section 3 shows the formal results that set up our finite characterization for the semantics and the related correctness and compositionality properties. In section 4 we first apply these results to static program analysis, by identifying a suitable class of abstract domains which allows the applicability of the previous results to the abstract semantics. Then we consider the case of Datalog programs.

2 Preliminaries

In the following we assume familiarity with the standard definitions and notation for logic programs. The standard reference works by Apt [1] and Lloyd [22] provide the necessary background material.

Throughout we denote by $(\Pi, \Sigma, \mathit{Var})$ a first-order language defined over a signature Σ (a set of function symbols with their arity), a set of predicate symbols Π and a denumerable set of variables Var. The non-ground term algebra over Σ and Var is denoted by $\mathit{Term}(\Sigma, \mathit{Var})$

or *Term* for short. The set of atoms constructed from predicate symbols in Π and terms from *Term* is denoted by $\mathcal{A}^{\Pi}(\Sigma, \mathit{Var})$ or $\mathcal{A}^{\Pi}$ for short. A goal is a sequence of atoms. The *length* of a sequence b is denoted by $|b|$. A clause is an object of the form $h :- B$ where h is an atom, called the *head*, and B is a goal, called the *body*. A clause with an empty body is a *unit clause*. The set of clauses constructed from elements of $\mathcal{A}^{\Pi}(\Sigma, \mathit{Var})$ is denoted by $\mathcal{C}(\Pi, \Sigma, \mathit{Var})$ or $\mathcal{C}$ for short. We will abuse by considering unit clauses as atoms. A logic program P is a finite set of clauses. A *substitution* ϑ is a mapping from *Var* to *Term* which acts as the identity almost everywhere, i.e. such that its domain $dom(\vartheta) = \{x \in \mathit{Var} \mid \vartheta(x) \neq x\}$ is finite. It extends to apply to any syntactic object (e.g. terms, atoms, clauses, *etc.*) in the usual way. The set $range(\vartheta)$ is defined as $range(\vartheta) = \cup_{x \in dom(\vartheta)} var(\vartheta(x))$. We will use the set theoretic notation $\left\{ x \mapsto t \,\middle|\, x \in dom(\theta),\ \theta(x) = t \right\}$ to represent θ. Given a syntactic object s, we denote by $\vartheta_{|s}$ the restriction of ϑ to the variables of s and by $S\theta$ the application of ϑ to s. Composition of substitutions ρ and σ is defined as usual and denoted by $\rho\sigma$. The set of substitutions is denoted by *Sub*. We fix a partial function *mgu* which maps a pair of syntactic objects to an idempotent most general unifier of the objects, if such exists. Thus, a statement $\theta = mgu(s, t)$ implies that s and t are unifiable. A *variable renaming* is a substitution that is a bijection on *Var*. Syntactic objects t_1 and t_2 are *equivalent up to renaming*, denoted by $t_1 \sim t_2$, if, for some variable renaming ρ, $t_1\rho = t_2$. Moreover we denote by $pred(t)$ the set of predicate symbols occurring in the syntactic object t. The set of variables that occur in a syntactic object t is denoted by $var(t)$. If G is a goal, $G \overset{\vartheta}{\leadsto}_P \Box$ denotes a SLD refutation of G in the program P with computed answer ϑ. A *parallel derivation* for a goal G in a program P is obtained by simultaneously replacing each atom in G with the body of clauses in P, provided that G and the corresponding clause heads are unifiable. The notion of ordinal powers of a function f on a complete lattice L is defined as usual, namely $f \uparrow 0 = \bot$, $f \uparrow (n+1) = f(f \uparrow n)$ and $f \uparrow \omega = \sqcup_{n<\omega} f \uparrow n$ where $\bot$ is the bottom and $\sqcup$ is the *lub* of the lattice. The notations $\tilde{t}$, $\tilde{x}$ will be used to denote tuples of terms and variables respectively, while $\tilde{B}$ denotes a (possibly empty) conjunction of atoms. Moreover, we denote the power set of X by $\wp(X)$. Finally, to specify function parameters, we often make use of Church's lambda notation.

2.1 Semantics for Composition

In this section we will introduce a semantics which is compositional wrt the union of (logic) programs. Compositionality has to do with a (syntactic) program composition operator $\circ$, and holds when the semantics of the compound construct $C_1 \circ C_2$ is defined by (semantically) composing the semantics of the constituents C_1 and C_2. In the case of logic programs, the problematic syntactic construct is the *union* of clauses. Indeed, it is easy to see that the usual semantics based on atoms, such as the least Herbrand model and the *s-semantics* [14] are not compositional wrt $\cup$. For example, for $P = \{p(X) :- q(X),\ r(a)\}$ and $Q = \{q(a)\}$, the least Herbrand model $M(P \cup Q) = \{p(a), q(a), r(a)\}$ of the union cannot be obtained from $M(P) = \{r(a)\}$ and $M(Q) = \{q(a)\}$.

This kind of compositionality is a desirable property since it allows an incremental and modular definition of programs and of their semantics, and, as shown in [9], it provides a semantic base for modular program analysis. In the case of logic languages, a typical partially defined program could be a program where the intensional definitions are completely known while extensional definitions are only partially known and can be further specified by adding new clauses. Partially defined programs have been called Ω-*open* programs. An Ω-open program (open program for short) is a logic program P together with a set Ω of predicate symbols. A predicate symbol occurring in Ω is considered to be only partially defined in P and can be further specified by composing P with other programs. As specified by the following definition, the composition of open programs is simply union of clauses modified in order to take into account the "interface" described by the set Ω.

Definition 2.1 (Ω-union) *[5] Let P_1 be an Ω_1-program and P_2 be an Ω_2-program. If $\Omega \subseteq \Omega_1 \cup \Omega_2$ and $(pred(P_1) \cap pred(P_2)) \subseteq (\Omega_1 \cap \Omega_2)$ then $P_1 \cup_\Omega P_2$ is the Ω-open program $P_1 \cup P_2$. Otherwise $P_1 \cup_\Omega P_2$ is not defined.*

A semantics which is compositional wrt the $\cup_\Omega$ operator is obtained in [5] by using a domain based on clauses. This semantics is formalized in terms of unfolding of clauses. The unfolding operator *unf* specifies the result of unfolding clauses from a program P_1 with clauses from a program P_2.

Definition 2.2 (Unfolding)
The unfolding operator $unf : \wp(C) \times \wp(C) \rightarrow \wp(C)$ is defined as

$$unf_P(Q) = \left\{ \hat{c} \;\middle|\; \begin{array}{l} \hat{c} = (h : -\tilde{b}_1, \quad \cdots \quad, \tilde{b}_n)\sigma, \\ c = h : -g_1, \ldots, g_n \in P, \ (n \geq 0), \ h_1 : -\tilde{b}_1, \ldots, h_n : -\tilde{b}_n \in Q \\ \textit{renamed apart and } \sigma = mgu(\langle g_1, \ldots, g_n \rangle, \langle h_1, \ldots, h_n \rangle) \end{array} \right\}.$$

With abuse of notation, in the following we denote by $\sim$ also the equivalence on clauses defined by $H : -A_1, \ldots, A_n \sim K : -B_1, \ldots, B_n$ iff there exists a permutation $\{i_1, \ldots, i_n\}$ such that $H : -A_1, \ldots, A_n$ and $K : -B_{i_1}, \ldots, B_{i_n}$ are variant. Moreover $\mathcal{C}$ is also the set of the $\sim$ equivalence classes of clauses (on the given signature) and, given a set of predicates Ω, we define $\mathcal{C}^\Omega = \{H : -\tilde{B} \in \mathcal{C} \mid pred(\tilde{B}) \subseteq \Omega\}$ and $\mathcal{A}^\Omega = \{A \mid A \in \mathcal{A}^\Pi \text{ and } pred(A) \in \Omega\}$.

Definition 2.3 (Interpretations) *[5]*
An interpretation *is any element in $\wp(\mathcal{C})$.*

Remark
To simplify the notation we will denote both a clause c and its $\sim$ equivalence class by c and we will consider a program P both as a set of clauses and as an interpretation. We assume that, when considering a syntactic element of an $\sim$ equivalence class, such an element is implicitly renamed apart, i.e. it does not share any variable with any other expression in the given context. Moreover, given any syntactic operator f on sets of clauses, f will denote also the operator on sets of $\sim$ equivalence classes obtained in the obvious way. This is correct since all the syntactic operators use clauses which are renamed apart.

Note that, according to previous remark, *unf* denotes also a binary (semantic) operator on interpretations. The unfolding operator is of interest as it can be applied to formalize both top-down and bottom-up semantics for logic programs [21]. Below we show the bottom-up construction of the semantics for the open-programs. For a set of predicate symbols Ω we denote by id_Ω the set $\{p(\tilde{x}) : -p(\tilde{x}) \in \mathcal{C} \mid p \in \Omega\}$.

Definition 2.4 (Fixpoint semantics [5])
The fixpoint semantics of an Ω-open program P is given by the function $\mathcal{F}_\Omega : \wp(\mathcal{C}) \rightarrow \wp(\mathcal{C})$, defined as $\mathcal{F}_\Omega(P) = lfp(T_P^\Omega)$, where $T_P^\Omega : \wp(\mathcal{C}) \rightarrow \wp(\mathcal{C})$ is defined as $T_P^\Omega(I) = unf_P(I \cup id_\Omega)$.

By T_P^Ω continuity we have $lfp(T_P^\Omega) = T_P^\Omega \uparrow \omega$. In order to obtain a compositional semantics where open predicates in Ω can be defined from outside P, the T_P^Ω operator generates interpretations which contain clauses. Technically, this is accomplished by letting id_Ω to add tautological clauses for the open predicates and by unfolding all predicates in the body of a clause. Notice that the unfolding of a predicate with a tautological clause is basically a "no-op" and that the clauses in $\mathcal{F}_\Omega(P)$ have bodies with only predicates in Ω. Moreover, note that for $\Omega = \emptyset$, since $id_\emptyset$ is empty, $T^\emptyset_P(I) = unf_P(I)$. Therefore when I consists of unit clauses (facts), $T^\emptyset_P$ is exactly the (non-ground) immediate consequence operator whose least fixpoint $\mathcal{F}_\emptyset(P)$ is the s-semantics of P ([14]).

Considering *computed answers* as "observable" and $\cup_\Omega$ as program composition operator, we obtain an "observational congruence" on programs which can be defined as follows.

Definition 2.5 *Let P_1, P_2 be Ω-open programs. Let G be a goal and Q be a program such that $P_i \cup_\Omega Q$ is defined, for $i = 1, 2$. Then we define $P_1 \approx_\Omega P_2$ iff for any G, Q we have that $G \overset{\vartheta}{\leadsto}_{P_1 \cup_\Omega Q} \Box$ iff $G \overset{\vartheta'}{\leadsto}_{P_2 \cup_\Omega Q} \Box$ and $G\vartheta \sim G\vartheta'$.*

$\mathcal{F}_\Omega$ is proved to be compositional wrt $\cup_\Omega$ and correct wrt $\approx_\Omega$. In other words $\mathcal{F}_\Omega(P)$ contains the information necessary to describe the behaviour of P in terms of computed answers, compositionally wrt union of programs. Moreover, each successful derivation for the goal G in $P \cup_\Omega Q$, where $Q \subseteq \mathcal{A}^\Omega$, can equivalently be performed by using parallel derivation steps in $\mathcal{F}_\Omega(P \cup_\Omega Q)$. The following are the formal results.

Theorem 2.6 (Compositionality [5])
Let P_1 be an Ω_1-open program and P_2 be an Ω_2-open program such that, for a set of predicates Ω, $P_1 \cup_\Omega P_2$ is defined. Then $\mathcal{F}_\Omega(P_1 \cup_\Omega P_2) = \mathcal{F}_\Omega(\mathcal{F}_{\Omega_1}(P_1) \cup_\Omega \mathcal{F}_{\Omega_2}(P_2))$.

Theorem 2.7 (Correctness [5])
Let P, P' be Ω-open programs. Then

1. *$P \approx_\Omega \mathcal{F}_\Omega(P)$.*
2. *if $\mathcal{F}_\Omega(P) = \mathcal{F}_\Omega(P')$ then $P \approx_\Omega P'$*

Moreover for $Q \subseteq \mathcal{A}^\Omega$ and for any goal $G = A_1, \ldots, A_n$, $G \overset{\vartheta}{\leadsto}_{P \cup_\Omega Q} \Box$ iff there exists $B_1, \ldots, B_n \in T^\emptyset_{\mathcal{F}_\Omega(P)}(Q) \cup Q$ such that $\gamma = mgu((A_1, \ldots, A_n)(B_1, \ldots, B_n))$ and $\gamma_{|G} = \vartheta_{|G}$.

For the particular case of $\Omega = \emptyset$ we have the following version of previous theorem which shows the full abstraction of the s-semantics.

Theorem 2.8 *[14] Let P_1, P_2 be programs. Then $P_1 \approx_\emptyset P_2$ iff $\mathcal{F}_\emptyset(P_1) = \mathcal{F}_\emptyset(P_2)$*

Finally note that, as shown by the following theorem (which generalize a theorem given in [17] for the case $\Omega = \Pi$), the composition $\cup_\Omega$ of two Ω-open programs induce the same equivalence $\approx_\Omega$ on programs as the union of an Ω-open program with set of atoms whose predicates are in Ω.

Theorem 2.9 *[16] Let P_1, P_2 be programs. Then $P_1 \not\approx_\Omega P_2$ iff there exists a set of atoms $Q \subseteq \mathcal{A}^\Omega$ such that $P_1 \cup Q \not\approx_\emptyset P_2 \cup Q$.*

3 Compositional Semantics over Finite Domains

In this section we will always consider programs and goals defined on a *finite signature* Σ which contains only constant symbols. We will equivalently refer to this assumption by saying that we consider programs on a *finite domain.*

Given an Ω-open program P, we define $P_n = T_P^{\Omega} \uparrow n$. P_n can be considered a kind of *partially* correct semantics of P. Indeed, when considering a signature Σ containing a function symbol (or infinite constant symbols), it is always possible to find a program P such that, for any finite n, $P \not\approx_{\Omega} P_n$. This is not the case if we consider a finite domain. In fact with this assumption we can obtain an $n^* < \omega$ such that the finite denotation $P_{n^*} = T_P^{\Omega} \uparrow n^*$ has the properties stated for $\mathcal{F}_{\Omega}$ in Theorem 2.7. Even if P_{n^*} is not a fixpoint of T_P^{Ω} (wrt set inclusion), P_{n^*} can be considered the *finite* compositional semantics of the program. Note that even for finite domains $\mathcal{F}_{\Omega}(P)$ may be infinite. For example, for the Ω-program $P = \{p, p : -p, q\}$, with $\Omega = \{q\}$, we obtain the infinite semantics $\mathcal{F}_{\Omega}(P) = \{p : -q,\ p : -q, q,\ p : -q, q, q \ldots\}$. In the following we show that P_{n^*} is the first interpretation in the chain $\{P_n\}_{n<\omega}$ such that, for any $Q \subseteq \mathcal{A}^{\Omega}$, $(T^{\emptyset}_{P_{n^*}} + id)(Q) = (T^{\emptyset}_{P_{n^*+1}} + id)(Q)$. This property is shared by all of the partial interpretations P_n in the chain, for $n \geq n^*$, including $lub(\{P_n\}_{n<\omega})$ (which is precisely the $\mathcal{F}_{\Omega}(P)$ semantics). This implies (essentially because of theorem 2.9) that we have obtained the same compositional behaviour as P. In order to formally prove these statements, let us first define the notion of T-stable partial semantics P_n, and let show that there exists a T-stable P_n for any program P.

Definition 3.1 *Let P be an Ω-open program. We say that P_n is T-stable iff for any $Q \subseteq \mathcal{A}^{\Omega}$, $(T^{\emptyset}_{P_n} + id)(Q) = (T^{\emptyset}_{P_{n+1}} + id)(Q)$.*

Proposition 3.2
Let P be an Ω-open program. There exists $n < \omega$ such that P_n is T-stable.

Lemma 3.3 shows that, as one would expect, when ascending in the chain $\{P_n\}_{n<\omega}$, T-stability is always preserved. The proof can be carried out easily by using the definition $T_P^{\Omega}(Q) = unf_P(Q \cup Id_{\Omega})$ and by the associativity of the unfolding.

Lemma 3.3
Let P be an Ω-open program and let $Q \subseteq \mathcal{A}^{\Omega}$. If $(T^{\emptyset}_{P_n} + id)(Q) = (T^{\emptyset}_{P_{n+1}} + id)(Q)$ then for any $k \geq n$ and for any $Q \subseteq \mathcal{A}^{\Omega}$, $(T^{\emptyset}_{P_n} + id)(Q) = (T^{\emptyset}_{P_k} + id)(Q)$.

Remark
Note that when considering programs on a finite domain, the set of (variance) equivalence classes of atoms $\mathcal{A}^{\Pi}$ is finite. Then for any program P and for any $Q \subseteq \mathcal{A}^{\Pi}$, $\mathcal{F}_{\emptyset}(P \cup Q)$ is a finite set. By theorem 2.9, $P_1 \approx_{\Omega} P_2$ iff for any $Q \subseteq \mathcal{A}^{\Omega} \subseteq \mathcal{A}$, $P_1 \cup Q \approx_{\emptyset} P_2 \cup Q$ iff (by Theorem 2.8) for any $Q \subseteq \mathcal{A}^{\Omega}$, $\mathcal{F}_{\emptyset}(P_1 \cup Q) = \mathcal{F}_{\emptyset}(P_2 \cup Q)$. Therefore for any Ω and any P_1, P_2 under previous assumption it is decidable if $P_1 \approx_{\Omega} P_2$.
Moreover, for any program P and for any n, P_n, $(T^{\emptyset}_P + id)(Q)$ and $(T^{\emptyset}_{P_n} + id)(Q)$ are finite sets. Therefore T-stability is decidable.

Let us now show the desired result for T-stable interpretations.

Proposition 3.4
Let P be an Ω-open program. Then

1. *if P_n is T-stable then $P \approx_{\Omega} P_n$. The converse does not hold,*

2. P_n *is* T*-stable iff for any goal* $G = A_1, \ldots, A_m$ *and for any set of atoms* $Q \subseteq \mathcal{A}^\Omega$*:*

$$G \overset{\vartheta}{\leadsto}_{P \cup Q} \Box \quad \Leftrightarrow \quad \textit{there exist } B_1, \ldots, B_m \in T^{\emptyset}_{P_n}(Q) \cup Q \textit{ such that } \vartheta_{|G} = \gamma_{|G}, \quad \textit{where } \gamma = mgu((A_1, \ldots, A_m)(B_1, \ldots, B_m)).$$

Example 3.1 Let us consider a signature $\Sigma = \{a, b\}$, a set of open predicates $\Omega = \{p, q\}$ and a program $P = \{p(X, Y) : -p(X, Y), q(X, Y)\}$. Then $P_1 \approx_\Omega P$ since, $T^\Omega_P \uparrow 1 = P_1 = P$. However P_1 is not T-stable since, for $Q = \{q(a, Y), q(X, b), p(X, Y)\}$, $T^{\emptyset}_{P_1}(Q) \neq T^{\emptyset}_{P_2}(Q)$. Indeed $p(a, b) \notin T^{\emptyset}_{P_1}(Q)$ and $p(a, b) \in T^{\emptyset}_{P_1}(Q)$ because $p(X, Y) : -p(X, Y), q(X, Y), q(X, Y) \in P_2 = T^\Omega_P(P_1)$. $\Box$

The previous results show that there exists a T-stable denotation P_n for program P, and that each T-stable denotation P_n satisfies the properties of $\mathcal{F}_\Omega(P)$, i.e. the compositional behaviour of a program P in terms of computed answers is completely specified by P_n. Indeed, further applications of the T^Ω operator for a T-stable interpretation may add new clauses, but these clauses do not add any possibility of computing new answers when composed with other program modules. We can now give the following definition of the finite compositional semantics P_{n^*}. It is clear from the definition and from proposition 3.4 that P_{n^*} is finite and has the required correctness and compositionality properties. In the next section we will show how P_{n^*} can be used to develop a modular analysis.

Definition 3.5 *Let* P *be an* Ω*-open program. Let* n^* *be the least index such that* P_{n^*} *is* T*-stable. Then* P_{n^*} *is the* finite *open semantics of the program* P*.*

We notice that the test for T-stability can be perfomed incrementally. Indeed for any $Q \subseteq \mathcal{A}^\Omega$, clearly $T^{\emptyset}_{P_{n+1}}(Q) = T^{\emptyset}_{P_n}(Q)$ iff $T^{\emptyset}_{\Delta\, n}(Q) \subseteq T^{\emptyset}_{P_n}(Q)$ where $\Delta\, 0 = \emptyset$ and $\Delta\, n = P_{n+1} \setminus P_n$ for $n \geq 1$. Moreover, we do not need to consider the whole $\wp(\mathcal{A}^\Omega)$ space for any P_n to decide T-stability. In fact, while computing the (T^Ω_P) fixpoint we can reduce the number of the elements in $\wp(\mathcal{A}^\Omega)$ to be checked by observing that, for a given $Q \in \wp(\mathcal{A}^\Omega)$, if $(T^{\emptyset}_{P_{n+1}} + id)(Q) = (T^{\emptyset}_{P_n} + id)(Q)$ then for any $k \geq n$ (by Lemma 3.3) $(T^{\emptyset}_{P_{k+1}}(Q) + id) = (T^{\emptyset}_{P_k} + id)(Q)$. A simple algorithm to compute the compositional T-stable semantics for a program module P can be defined as in Table 1. The procedure is terminating because the set D decreases during the computation (a straightforward consequence of Proposition 3.2).

For $\Omega \neq \Pi$, there exists P and n such that $P \not\approx_\Omega P_n$. For example, if $P = \{q(x), p(X) : -q(X)\}$ and $\Omega = \emptyset$, then $P \not\approx_\emptyset P_n$ for $n = 0, 1$ and $P \approx_\emptyset P_2$. Moreover, as shown in Example 3.1, in general P_{n^*} is not the first interpretation in the chain $\{P_n\}_{n<\omega}$ which is $\approx_\Omega$ equivalent to P.

We can alternatively consider as finite compositional semantics for P the denotation $P_{\overline{n}}$ where $\overline{n}$ is the least n such that $P \approx_\Omega P_n$. By Proposition 3.4 $P_{\overline{n}}$ is a subset of P_{n^*} and it is easy to see that $P_{\overline{n}}$ has the same properties of P_{n^*}, with the only difference that a derivation in $P_{\overline{n}} \cup Q$ may require more than two (parallel) derivation steps. For some applications in data-flow analysis, where it is more important to maintain a small dimension of the semantics than to have an efficient analysis, the semantics $P_{\overline{n}}$ can provide better results than P_{n^*}. It is worth noting that, as shown by the following result, there exists an interesting class of programs for which n^* and $\overline{n}$ coincide, namely *predicate disjoint programs*. We say that the predicate p is defined in the program P if P contain a clause such that p appear in the head.

Definition 3.6 *An* Ω*-open program* P *is* predicate disjoint *if any predicate symbol defined in* P *does not belong to* Ω*.*

Input: A Ω-open program P.
Output: The T-stable semantics Pn for P.

begin
$\{n = 0\}$
$D := \wp(\mathcal{A}^{\Omega})$;
$Pn := \emptyset$;
$\Delta\, n := T_P^{\Omega}(\emptyset)$;
while $D \neq \emptyset$ and $\Delta\, n \neq \emptyset$ **do**
$\Big\{ Pn = T_P^{\Omega} \uparrow n \ \wedge\ \Delta\, n = T_P^{\Omega} \uparrow (n+1) \setminus Pn \ \wedge\ \Big((T_{Pn}^{\emptyset} + id)(X) \neq (T_{T_P^{\Omega}(Pn)}^{\emptyset} + id)(X) \Rightarrow X \in D \Big) \Big\}$
for each $X \in D$ **do**
if $(T_{\Delta\, n}^{\emptyset} + id)(X) \subseteq (T_{Pn}^{\emptyset} + id)(X)$ **then** $D := D \setminus X$ **fi**;
$Pn := Pn \cup \Delta\, n$;
$\Delta\, n := T_P^{\Omega}(Pn) \setminus Pn$
$\{n := n + 1\}$
endw $\{Pn = T_P^{\Omega} \uparrow n$ is a T-stable interpretation in $\{P_n\}_{n<\omega} \ \wedge\ \Delta\, n = P_{n+1} \setminus P_n\}$
end

Table 1: An algorithm for T-stable semantics.

According to previous definition, if we consider a $\cup_{\Omega}$ composition of *predicate disjoint* modules, predicates defined in each module are disjoint from those defined in the others. From a practical point of view the notion of *predicate disjoint* module is relevant since often module based implementations of logic programming languages assume this kind of separation. A similar restriction is also considered in [17] and in [9]. A typical example of *predicate disjoint* program is provided by a Datalog program, where intensional and extensional predicates are disjoint, and where the extensional predicates are considered partially defined (see Section 4.2). Let us now prove the mentioned equality.

Proposition 3.7
Let P be an Ω-open predicate disjoint program. Let $\overline{n}$ be the least n such that $P_n \approx_{\Omega} P$ and let n^* be defined as in Definition 3.5. Then $n^* = \overline{n}$.

4 Applications

In the following we describe two applications of the previous results in the context of compositional data-flow analysis and logic-based databases.

4.1 Compositional Analysis over Finite Domains

It is well known that modularity helps in reducing the complexity of designing and proving correctness of programs and provides an easy approach to develop adaptable software. However, most of the already existing frameworks for analysis ([8, 3, 6, 12]) require the entire program to be available at the time of analysis. This approach is often unpractical for large programs or for software developed by teams, either because the resource requirements are prohibitively high, or because not all program components are available at the same time for analysis.

The relevance of compositionality in logic program analysis has been firstly addressed in [9] by introducing a framework which is based on the compositional semantics $\mathcal{F}_\Omega$ and which formalizes the analysis in terms of the standard theory of abstract interpretation ([11]). The abstract meaning of a module corresponds to its analysis and composition of abstract semantics corresponds to composition of analyses. In the following we will briefly describe this framework. Then, using the results of previous section, we will show how for a wide class of abstract domains the analysis can be performed by avoiding a further level of abstraction on clauses which was introduced in [9] in order to treat any generic (possibly infinite) abstract domain.

An abstract semantics typically associate programs with entities which capture the essence of "what they do" while abstracting away from the *concrete* details related with its execution model. This abstraction should provide a finite characterization of the (concrete) semantics, which is useful for static data-flow analysis of programs. Let $(\Pi, \Sigma, \mathit{Var})$ be a first-order language. To handle compositionality, the abstract semantic objects are descriptions of clauses. They are defined by describing sets of substitutions in $\wp(\mathit{Sub})$ by (abstract) descriptions of substitutions in a lattice ASub. We denote by $\sqcup_S$ the corresponding *lub*. As the standard theory of abstract interpretation is formalized in terms *Galois insertions* [11], specifying the relation between abstract and concrete semantics objects, we assume $(\wp(\mathit{Sub}), \alpha_S, \gamma_S, \mathit{ASub})$ be a Galois insertion[1]. *Abstract clauses* are pairs $\mathcal{LC} \times \mathit{ASub}$, where $\mathcal{LC} \subseteq \mathcal{C}(\Pi, \emptyset, \mathit{Var})$ is the set of *flat clauses*, i.e. where no variable occurs twice. Abstract interpretations are sets of abstract clauses. Let I^a be an abstract interpretation. The meaning of I^a is specified by the following concretization function γ:

$$\gamma(I^a) = \left\{ c\theta \;\middle|\; \begin{array}{l} \langle c; \kappa \rangle \in I^a \\ \theta \in \gamma_S(\kappa) \end{array} \right\}.$$

Several abstract objects may represent the same concrete interpretation (i.e. γ may not be injective). This problem is solved in [9] by defining an equivalence relation $\sim_\gamma$ on $\wp(\mathcal{LC} \times \mathit{ASub})$ such that $I_1^a \sim_\gamma I_2^a$ iff $\gamma(I_1^a) = \gamma(I_2^a)$. Notice that the equivalence $\sim_\gamma$ provides variable restriction and equivalence up to renaming.

Let $\mathit{AInt} = \wp(\mathcal{LC} \times \mathit{ASub})/_{\sim_\gamma}$ be a complete lattice with $\sqcup$ as *lub*. This approach induces a Galois insertion between concrete interpretations $\wp(\mathcal{C})$ and abstract interpretations AInt: $(\wp(\mathcal{C}), \alpha, \gamma, \mathit{AInt})$ (α is induced by γ in the usual way).

As in the concrete case, the abstract fixpoint operator is based on iterating *abstract unfolding*. Let $P^a, Q^a \in \mathit{AInt}$, and $\mathit{unf}^a : \mathit{AInt} \times \mathit{AInt} \to \mathit{AInt}$ such that

$$\mathit{unf}^a_{P^a}(Q^a) = \sqcup \left\{ \bar{c} \;\middle|\; \begin{array}{l} \bar{c} = \langle h :- \tilde{b}_1 :: \quad \cdots \quad :: \tilde{b}_n; \hat{\kappa} \rangle, \\ c = \langle h :- g_1, \ldots, g_n; \kappa \rangle \in P^a, \\ \langle h_1 :- \tilde{b}_1; \kappa_1 \rangle, \ldots, \langle h_n :- \tilde{b}_n; \kappa_n \rangle \in Q^a \text{ renamed apart}, \\ \hat{\kappa} = \mathit{mgu}^a\left(\langle \langle g_1, \ldots, g_n \rangle; \kappa \rangle, \langle \langle h_1, \ldots, h_n \rangle; \kappa_1 \ldots \kappa_n \rangle \right) \end{array} \right\}.$$

where mgu^a represent the *abstract unification* and, for any $\vartheta \in \gamma_S(\kappa_1)$ and $\sigma \in \gamma_S(\kappa_2)$, if $\mathit{dom}(\vartheta) \cap \mathit{dom}(\sigma) = \emptyset$ and $\mathit{range}(\sigma) \cap \mathit{range}(\vartheta) = \emptyset$, then $\vartheta\sigma \in \gamma_S(\kappa_1\kappa_2)$.

Given a program (module) P let us define $P^a = \alpha(P)$ and $\mathcal{T}_P^\Omega : \mathit{AInt} \to \mathit{AInt}$ such that $\mathcal{T}_{P^a}^\Omega(I^a) = \mathit{unf}^a_{P^a}(I^a \cup \alpha(\mathit{id}_\Omega))$. By unf^a definition, $\mathcal{T}_{P^a}^\Omega$ is continuous. Moreover, by assuming the correctness of the abstract unfolding (i.e. $\mathit{unf}_P(Q) \subseteq \gamma(\mathit{unf}^a_{\alpha(P)}(\alpha(Q)))$ it follows that for each logic program P and interpretation I:

$$T_P^\Omega(I) \subseteq \gamma(\mathcal{T}_{\alpha(P)}^\Omega(\alpha(I))).$$

[1] i.e. α_S and γ_S are monotonic functions, $\alpha_S \circ \gamma_S = \mathit{id}$ and $\gamma_S \circ \alpha_S \supseteq \mathit{id}$ [11].

Based on this observations, the following *abstract composition* theorem guarantees the correctness of the analysis. We denote $\sqcup_\Omega$ the abstract version of $\cup_\Omega$.

Theorem 4.1 (Correctness [9])
Let P_1 and P_2 be Ω_1 and Ω_2-open programs respectively and $\Omega \subseteq \Pi$ such that $P_1 \cup_\Omega P_2$ is defined. Let also $(\wp(\mathcal{C}), \alpha, \gamma, AInt)$ be a Galois insertion such that

$$unf_P(Q) \subseteq \gamma(unf^a_{\alpha(P)}(\alpha(Q))).$$

Then $\mathcal{F}_\Omega(P_1 \cup_\Omega P_2) \subseteq \gamma(\mathcal{F}^a_\Omega(\mathcal{F}^a_{\Omega_1}(\alpha(P_1)) \sqcup_\Omega \mathcal{F}^a_{\Omega_2}(\alpha(P_2))))$.

The correctness specifies the meaning of the approximation introduced to let the analysis be effective. In this case, the concrete meaning of the composition of two programs is described by the composition of abstract meaning of the two programs, i.e. everything which is computed is described.

While the standard correctness property for abstract unfolding is based on the correctness of the abstract unification mgu^a, several technical properties of the abstract unification are required to allow the abstract unfolding to correctly mimic the concrete one. In particular, if $\mathcal{K} \subseteq ASub$, $\kappa, \kappa_1, \kappa_2, \kappa_3 \in ASub$, and $\bar{a}, \bar{b}, \bar{c}, \bar{d}$ be renamed apart tuples of flat atoms such that $|\bar{a}| = |\bar{c}|$ and $|\bar{b}| = |\bar{d}|$, we assume:

- $mgu^a(\langle \bar{a}; \sqcup_S \mathcal{K}\rangle, \langle \bar{c}; \kappa\rangle) = \sqcup_S \left\{ mgu^a(\langle \bar{a}; \kappa'\rangle, \langle \bar{c}; \kappa\rangle) \,\middle|\, \kappa' \in \mathcal{K} \right\}$ *additivity*
- $mgu^a(\langle \bar{a}; \kappa_1\rangle, \langle \bar{c}; \kappa_2\rangle) = mgu^a(\langle \bar{c}; \kappa_2\rangle, \langle \bar{a}; \kappa_1\rangle)$ *commutativity*
- $mgu^a(\langle \bar{b}; mgu^a(\langle \bar{a}; \kappa_1\rangle, \langle \bar{c}; \kappa_2\rangle)\rangle, \langle \bar{d}; \kappa_3\rangle) = mgu^a(\langle \bar{a}; \kappa_1\rangle, \langle \bar{c}; mgu^a(\langle \bar{b}; \kappa_2\rangle, \langle \bar{d}; \kappa_3\rangle)\rangle)$

  associativity

To extend the results of the previous section to the abstract semantics for composition, we need an associative and additive abstract unfolding operator. As shown by the following lemma, the associativity of the abstract unification implies the associativity of the abstract unfolding. This property allow us to repeat essentially the same proofs of previous section also for the abstract semantics.

Lemma 4.2
If mgu^a is associative and commutative then unf^a is associative.

To provide convergence in solving semantic fixpoint equations, the abstract domain is usually required to satisfy the *ascending chain condition.* Most of the existing abstract domains for substitutions are studied to satisfy this condition (e.g. *Prop* [10] [2], *affine relations* [20] [3] and *Sharing* [19] which is discussed later in the paper). Even if *ASub* satisfies this

---

[2]The abstract domain *Prop* provides a concise representation for abstract substitutions able to describe ground dependency relations among their arguments. A propositional formula with connectives $\{\leftrightarrow, \vee, \wedge\}$ is associated with each substitution. For example, the formula x represents a substitution ϑ such that $\vartheta(x)$ is ground, while $x \leftrightarrow y \wedge z$ represents a substitution ϑ such that $var(\vartheta(x)) = var(\vartheta(y)) \cup var(\vartheta(z))$, i.e. x is ground iff both y and z are ground.

[3]The domain of *affine relations* has been introduced to derive linear size relations among variables in a program. This analysis is useful for compile-time overflow and array bound checking. Let V be a finite set of variables and $\delta : Term \rightarrow \omega$ be a norm. In the logic programming case an abstract substitution is an *affine subspace* of $\Re^{|V|}$, i.e. a point, line, plane *etc.* possibly not including the origin. An abstract substitution κ represents any concrete substitution ϑ such that $dom(\vartheta) = V$ and the vector $\langle \delta(\vartheta(x))\rangle_{x \in V}$ satisfies κ. This abstract domain satisfies the ascending chain condition because in any properly ascending chain of affine subspaces $\kappa_1 \subseteq \kappa_2 \subseteq ...$ the subspace κ_i must have a dimension of at least one greater than κ_{i-1}, and $|V| < \omega$.

condition, the induced abstract domain of interpretations does not guarantee termination, as it is always possible to build (by unfolding) clauses with arbitrary long bodies. Thus, due to the base semantics construction, an additional layer of abstraction or the use of *extrapolation techniques* like *widening/narrowing* ([11]) is required. The solution adopted in [9] is a further abstraction performed by using these techniques (i.e. $\star$-abstraction). Abstract clauses are allowed to contain at most one occurrence of the same predicate symbol. Multiple occurrences of atoms for the same predicate in a body are then "compressed" to a single atom. In general, such a compression introduces a further approximation which makes the analysis less precise.

In view of the previous results, we can identify a wide class of abstract domains for which finiteness of the semantics, and hence termination, can be guaranteed independently on the finiteness of $AInt$. Therefore, using these domains the (compositional) analysis can be performed without the mentioned further loss of precision. We classify these domains as *compositionally tractable* abstract domains. From the abstract interpretation viewpoint, they play the same role as *finite domains* in (concrete) logic programming.

Definition 4.3
The abstract domain of interpretations $AInt$ is compositionally tractable iff for each flat clause c, the set $\left\{ [\{\langle c;\kappa\rangle\}]_{\sim_\gamma} \;\middle|\; \kappa \in ASub \right\}$ is finite.

As in the concrete case, we define $P^a_n = \mathcal{T}^{\Omega}_{P^a} \uparrow n$, for $P^a \in AInt$. Moreover, we extend the notion of T-stability from concrete to abstract interpretations in the obvious way. Notice that, by correctness: $\gamma(\alpha(P_n)) \subseteq \gamma(\alpha(P)_n)$. The proof of the following result is analogous to that of Proposition 3.2, since by mgu^a additivity $\lambda I.unf^a_I(P^a)$ is additive.

Proposition 4.4
Let P be an Ω-open program and $AInt$ be a compositionally tractable abstract domain. There exists $n < \omega$ such that $\alpha(P)_n$ is T-stable.

A compositional abstract interpretation can be provided by a finite iteration of $\mathcal{T}$. We call this interpretation a *T-stable abstract interpretation.* A characterization of n^* can be obtained by mimicing the results of the previous section (Lemma 3.3 and Proposition 3.7). The algorithm in Table 1 provides a compositional abstract interpretation on a compositionally tractable domain for any Ω open program (module). The followin theorem states the correctness of a T-stable abstract interpretation wrt computed answers.

Theorem 4.5
Let $G = A_1, ..., A_m$ be a goal, $Q \subseteq \mathcal{A}^{\Omega}$ and P be a Ω-open program and $\alpha(P)_n$ be T-stable. If $G \overset{\vartheta}{\leadsto}_{P\cup Q} \Box$ then there exist $B_1, ..., B_m \in \gamma\left(\mathcal{T}^{\emptyset}_{\alpha(P)_n}(\alpha(Q)) \sqcup \alpha(Q)\right)$ renamed apart such that $\beta = mgu((A_1, \ldots, A_m)(B_1, \ldots, B_m))$ and $\vartheta_{|G} = \beta_{|G}$.

Several abstract domains for logic program are compositionally tractable. As an example of a domain of abstract substitutions, in the following we consider the domain *Sharing*, proposed by Jacobs and Langen in [19]. *Sharing* has been introduced to identify aliasing between variables with a great deal of accuracy. Let σ be a (concrete) substitution: x and y *share* the variable w if $w \in var(\sigma(x)) \cap var(\sigma(y))$. The sharing information description is based on the notion of *sharing group*. The sharing group of a variable w (denoted by $sg(\sigma, w)$) is the set of variables in the domain of σ that share w [19]. A suitable representation for aliasing is provided by defining $Sharing = \wp(\wp(Var))$. Intuitively, if $S \in Sharing$, each of the $X \in S$ represents a sharing group for a given variable name. For example, the substitution

$$\sigma = \{x_1 \mapsto [A \mid L], x_2 \mapsto M, x_3 \mapsto [A \mid N], x_4 \mapsto L, x_5 \mapsto M, x_6 \mapsto N\}$$

is represented by the abstract substitution: $\kappa = \{\{x_1, x_4\}, \{x_1, x_3\}, \{x_2, x_5\}, \{x_3, x_6\}\}$, where $\{x_1, x_4\}$ is the sharing group for L in σ. Groundness and independence can also be derived from *Sharing*. Let $S \in \mathit{Sharing}$; a variable is ground in S if it does not appear in any sharing group of S. Two variables x and y are *independent* iff they do not share a common variable. By the definition, x and y are independent in S iff they do not appear together in the same sharing group. Ground dependences can also be captured: in $\{\{x_1, x_3\}, \{x_2, x_3\}\}$ x_3 is ground iff both x_2 and x_3 are ground.

Proposition 4.6
Sharing is compositionally tractable.

The same holds for the abstract domain *Prop* ([10]) and for depth-k abstractions ([3]). On the other hand, the abstract domain of *affine relations* ([20]) is not compositionally tractable because for $n < \omega$, there are infinitely many affine subspaces in $\Re^n$.

Because of the use of flat clauses as semantic objects, abstract unification is much simpler than the one defined in [19]. We follow Jacobs and Langen in the definition of mgu^a. Let $S \in \mathit{Sharing}$, $x \in \mathit{Var}$ and $in(x, S) = \left\{ X \in S \,\middle|\, x \in X \right\}$. The *closure under union* of S, denoted by S^* is the smallest superset of S such that $X \in S^* \;\wedge\; Y \in S^* \;\Rightarrow X \cup Y \in S^*$. The abstract unification requires an auxiliary binary operator $\oplus$ on *Sharing*, such that $S \oplus T = \left\{ X \cup Y \,\middle|\, X \in S, Y \in T \right\}$ for $S, T \in \mathit{Sharing}$. Let $p(x_1, ..., x_n)$ and $p(x_1', ..., x_n')$ be (flat) atoms, and $\kappa, \kappa' \in \mathit{Sharing}$:

$$mgu^a(\langle p(x_1, .., x_n); \kappa\rangle, \langle p(x_1', .., x_n'); \kappa'\rangle) = unify(x_1, .., x_n; x_1', .., x_n'; \kappa \cup \kappa') \quad \text{where}$$

$$unify(x_1, ..., x_n; x_1', ..., x_n'; S) = \begin{cases} (S \setminus in(x_1, S) \setminus in(x_1', S)) \cup \\ \qquad\qquad (in(x_1, S) \oplus in(x_1', S))^* & \text{if } n = 1 \\ \\ unify(x_2, ..., x_n; x_2', ..., x_n'; unify(x_1; x_1'; S)) & \text{if } n > 1 \end{cases}$$

The correctness of mgu^a is a consequence of the correctness condition proved in [19]. The same holds for additivity, commutativity and associativity of mgu^a. The relevance of *Sharing* in data-flow analysis of logic programs has been considered in [23] to exploit *independent AND-parallelism*. Other important optimizations can be obtained from the ground dependency information. In the following we consider an example for compositional sharing analysis defined in terms of T-stable abstract interpretations. It shows a practical case where, once a module has been changed, the semantics of the composition may be affected.

Example 4.1 Consider an "insertion sort" routine defined in terms of three modules:

P_{sort}
```
sort([],[]).
sort([X|Xs],Ys) :- sort(Xs,Zs),insert(X,Zs,Ys).
```

P_{ins}
```
insert(X,[],X).
insert(X,[Y|Ys],[Y|Zs]) :- gt(X,Y),insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Zs]) :- le(X,Y).
```

P_{gl}
```
gt(s(X), X).                      le(X, X).
gt(s(X), Y) :- gt(X, Y).          le(X, s(Y)) :- le(X,Y).
gt(s(X), s(Y)) :- gt(X, Y).       le(s(X), s(Y)) :- le(X, Y).
```

Assume `insert` be open in P_{sort}. Then we obtain:

$I^a_{sort_1}$: $\langle sort(x_1,x_2) :- ; \emptyset\rangle$

$I^a_{sort_2}$: $\langle sort(x_1,x_2) :- insert(x_5,x_6,x_7); \{\{x_1,x_5\},\{x_2,x_7\}\}\rangle$

$$I^a_{sort_3}: \left\langle \begin{array}{r} sort(x_1,x_2) :- insert(x'_5,x'_6,x'_7), insert(x_5,x_6,x_7); \\ \{\{x_1,x_5\},\{x_2,x_7\},\{x_6,x'_7\}\} \end{array} \right\rangle$$

$\vdots$

Assume also `le` and `gt` be open in P_{ins}. Thus we have:

$I^a_{ins_1}$: $\langle insert(x_1,x_2,x_3) :- ; \{\{x_1,x_3\}\}\rangle$

$$\left\langle \begin{array}{c} insert(x_1,x_2,x_3) :- le(x_4,x_5); \\ \{\{x_1,x_4,x_3\},\{x_2,x_3,x_5\},\{x_2,x_3\}\} \end{array} \right\rangle$$

$I^a_{ins_2}$: $\left\langle\ insert(x_1,x_2,x_3) :- gt(x_4,x_5); \{\{x_2,x_3,x_5\},\{x_1,x_4,x_3\}\}\ \right\rangle$

$$\left\langle \begin{array}{c} insert(x_1,x_2,x_3) :- gt(x_4,x_5), le(x'_4,x'_5); \\ \{\{x_1,x_4,x'_4\},\{x_2,x_3,x'_5\},\{x_2,x_3\},\{x_2,x_3,x_5\}\} \end{array} \right\rangle$$

$\vdots$

P_{gl} is closed (i.e. completely defined) and has the following abstract semantics $\alpha(P_{gl})_1$:

$$\langle gt(x_1,x_2) :- ; \{\{x_1,x_2\}\}\rangle \quad \langle le(x_1,x_2) :- ; \{\{x_1,x_2\}\}\rangle$$

The T-stable abstract semantics for `insert` is obtained in 2 steps and hence is $\alpha(P_{ins})_2 = I^a_{ins_1} \cup I^a_{ins_2}$. Notice that the test for T-stability of $\alpha(P_{ins})_2$ can be performed by unfolding $\alpha(P_{ins})_3 \backslash \alpha(P_{ins})_2$ with any possible sharing-pattern for `gt` and `le`. The abstract interpretation for $P_{ins} \cup P_{gl}$ is given by composing the interpretations for each single module:

$$\left\{ \begin{array}{l} \langle gt(x_1,x_2) :- ; \{\{x_1,x_2\}\}\rangle \quad \langle le(x_1,x_2) :- ; \{\{x_1,x_2\}\}\rangle \\ \langle insert(x_1,x_2,x_3) :- ; \{\{x_1,x_3\}\}\rangle \\ \langle insert(x_1,x_2,x_3) :- ; \{\{x_1,x_2,x_3\},\{x_2,x_3\}\}\rangle \\ \langle insert(x_1,x_2,x_3) :- ; \{\{x_1,x_2,x_3\}\}\rangle \end{array} \right\}.$$

A different definition for the predicates `gt` and `le`, like the following:

P'_{gl}
```
gt(s(0), 0).                        le(0, 0).
gt(s(X), Y) :- gt(X, Y).            le(X, s(Y)) :- le(X,Y).
gt(s(X), s(Y)) :- gt(X, Y).         le(s(X), s(Y)) :- le(X, Y).
```

provides a different meaning for $P_{ins} \cup P'_{gl}$ wrt the sharing information:

$$\left\{ \begin{array}{l} \langle gt(x_1,x_2) :- ; \emptyset\rangle \quad \langle le(x_1,x_2) :- ; \emptyset\rangle \\ \langle insert(x_1,x_2,x_3) :- ; \{\{x_1,x_3\}\}\rangle \\ \langle insert(x_1,x_2,x_3) :- ; \{\{x_2,x_3\}\}\rangle \\ \langle insert(x_1,x_2,x_3) :- ; \emptyset\rangle \end{array} \right\}.$$

We do not need to recompile the analysis for P_{ins} and P_{sort} once P_{gl} has been changed. The abstract meaning of P_{sort} does not result affected by modifying P_{gl}. For P_{sort}, in both the cases, we obtain: $\{\langle sort(x_1,x_2) :- ; \emptyset\rangle\ ,\ \langle sort(x_1,x_2) :- ; \{\{x_1,x_2\}\}\rangle\}$. $\square$

4.2 Databases

A Datalog program is a function-free logic program consisting of two components [7]: the extensional database (*EDB*) and the intensional database (*IDB*). The set Π of predicate symbols is partitioned into Π_{EDB} and Π_{IDB}. *EDB* uses only predicate symbols of Π_{EDB} and defines *extensional* relations. Clauses of *IDB* are built using symbols in Π_{IDB} for the head and symbols in Π for the body, and define *intensional* relations. A logical query is a goal. Two basic methods of logical query processing can be identified. Using the *interpretation method*, a query is directly evaluated against the set of clauses (*IDB*) and the base relations (*EDB*). The query interpreter receives as input a set of base relations, a set of clauses, a query, and returns a result. This method does not, in essence, distinguish between the *data extension*, in the form of base relations, and the *data intension*, in the form of a set of clauses. On the other side, the *compilation method* makes a distinction between the data intension and the data extension by considering the intensional database as a *mapping* between intensional relations and extensional one. It divides query processing into the following three phases:

1. A *database translation phase*, which receives as input the set of clauses of the *IDB*, and transform it into a recursion-free one.

2. A *query translation phase*, which receive as input the translated database, and a query, possibly involving one or more intensional relations, and produces the equivalent query that involves only extensional relations.

3. An *execution phase* which receives as input the translated query, evaluates this query against the stored data and returns the results.

The *standard fixpoint semantics* for logic programs ([1, 22]) as introduced by van Emden and Kowalski, provides a bottom-up execution model which is strongly related to the interpretation view. Several evaluation strategies have been developed to improve the naive strategies based on the immediate consequence operator [2]. However, interpretation methods are very inefficient and therefore most systems approach the problem of query processing by means of compilation. The main problem for compilation methods is the treatment of recursive clauses. Several specialized techniques have been proposed for specific classes of clauses (such as linear, regular etc.) and there is not a general compilation framework which can treat all the types of rules. The standard fixpoint semantics in this case is not useful since it does not distinguish between the data intension and the data extension.

The compositional fixpoint semantics introduced in Definition 2.4 allows to consider the data extension and data intension as separate modules. The set of extensional predicates Π_{EDB} can be seen as the set of open predicate symbols Ω. *IDB* and *EDB* are then two predicate disjoint Π_{EDB}-open programs. Intensional relations are hypothetically defined in $\mathcal{F}_{\Pi_{EDB}}$ by means of non recursive clauses. For Datalog programs defined over finite domains, according to the results of Section 3, we can then obtain a finite semantics (the T-stable IDB_n) with the mentioned correctness and compositionality features. This provides the basis for an *uniform technique* to translate the intensional database *IDB* into an equivalent one which is recursion-free (and which maps intensional relations into extensional one). The translation is obtained essentially by the algorithm in Table 1. Note that, according to proposition 3.4 the translated *IDB* allows to answer to any query (with intensional relations) using at most two (parallel) derivation steps.

Example 4.2
Consider $\Sigma = \{\texttt{henry}, \texttt{peter}, \texttt{john}\}$, $\Omega = \{\texttt{parent}\}$,
$EDB = \{\texttt{parent(henry,peter)}$, $\texttt{parent(peter,john)}$ and

$$\begin{array}{ll} IDB = \{ & \texttt{ancestor(X,Y)} :- \ \texttt{parent(X,Y)}. \\ & \texttt{ancestor(X,Z)} :- \ \texttt{parent(X,Y)}, \ \texttt{ancestor(Y,Z)}. \ \} \end{array}$$

Then

$$\begin{array}{ll} T^{\Omega}_{IDB} \uparrow \omega = \{ & ancestor(X,Y) :- parent(X,Y), \\ & ancestor(X,Z) :- parent(X,Y), parent(Y,Z), \\ & \vdots \\ & ancestor(X,Z) :- parent(X,Y_1), \ldots, parent(Y_t,Z), \\ & \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \end{array}$$

However, since the domain is finite, we can obtain the following finite semantics $T^{\Omega}_{IDB} \uparrow 2$ such that $T^{\Omega}_{IDB} \uparrow 2 \approx_{\Omega} IDB$. The query $\texttt{ancestor(henry,john)}$ is translated by means of $T^{\Omega}_{IDB} \uparrow 2$ into the extensional query $\texttt{parent(henry,Y),parent(Y,john)}$ which is evaluated against EDB.

$$\begin{array}{ll} T^{\Omega}_{IDB} \uparrow 2 = \{ & ancestor(X,Y) :- parent(X,Y), \\ & ancestor(X,Z) :- parent(X,Y), parent(Y,Z) \quad \} \end{array}$$

□

Because of compositionality, the mentioned translation technique can be carried out in an incremental and modular way, with obvious advantages. Moreover, the (finite) compositional semantics can be used to support modular construction of Datalog programs where also the *IDB* is partitioned in several modules. As illustrated by the previous examples, each time a new module is added, we do not need to recompute the semantics of (and we do not need to recompile) the whole system but we can compose the semantic of the new module with that of the old one. To the best of our knowledge, the only existing approach to modular databases is in [15], but it does not support a modular bottom-up execution model such as the one considered above.

References

[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] F. Bancilhon and R. Ramakrishnan. Performance Evaluation of Data Intensive Logic Programs. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 439–519. Morgan-Kaufmann, 1987.

[3] 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*, 15(1):133–181, 1993.

[4] E. Bertino, M. Martelli, and D. Montesi. An Incremental Semantics for CLP(AD). In A. Marchetti, Spaccamela, P. Mentrasti, and M. Venturini Zilli, editors, *Proc. Fourth Italian Conference on Theoretical Computer Science*, pages 53–67. World Scientific, 1992.

[5] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. Contributions to the Semantics of Open Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 570–580, 1992.

[6] M. Bruynooghe. A Practical Framework for the Abstract Interpretations of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[7] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog - And Never Dared to Ask. *IEEE Tran. on Knowledge and Data Eng.*, 1(1):146–164, March 1989.

[8] M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. Technical report, Dept. of Computer Science, The Weizmann Institute, Rehovot, 1990. To appear in Theoretical Computer Science.

[9] M. Codish, S. K. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Proc. Twentieth Annual ACM Symp. on Principles of Programming Languages*, pages 451–464. ACM Press, 1993.

[10] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.

[11] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[12] S. K. Debray. Efficient Dataflow Analysis of Logic Programs. *Journal of the ACM*, 39(4):949–984, 1992.

[13] F. Denis and J.-P. Delahaye. Unfolding, Procedural and Fixpoint Semantics of Logic Programs. In C. Choffrut and M. Jantzen, editors, *STACS 91*, volume 480 of *Lecture Notes in Computer Science*, pages 511–522. Springer-Verlag, Berlin, 1991.

[14] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[15] B. Freitag. A Deductive Database Language Supporting Modules. In *Proc. Second Int'l Computer Science Conference*, 1992. To appear.

[16] M. Gabbrielli, G. Levi, and M. C. Meo. Observational Equivalences for Logic Programs. In K. Apt, editor, *Proc. Joint Int'l Conf. and Symposium on Logic Programming*, pages 131–145. The MIT Press, Cambridge, Mass., 1992.

[17] H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 134–142. ACM, 1989.

[18] H. Gaifman and E. Shapiro. Proof theory and semantics of logic programs. In *Proc. Fourth IEEE Symp. on Logic In Computer Science*, pages 50–62. IEEE Computer Society Press, 1989.

[19] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992.

[20] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[21] G. Levi. Models, Unfolding Rules and Fixpoint Semantics. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1649–1665. The MIT Press, Cambridge, Mass., 1988.

[22] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

[23] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):315–347, 1992.