Compositional Analysis of Modular Logic Programs

Michael Codish* Saumya K. Debray‡ Roberto Giacobazzi§

Abstract

This paper describes a semantic basis for a compositional approach to the analysis of logic programs. A logic program is viewed as consisting of a set of modules, each module defining a subset of the program's predicates. Analyses are constructed by considering abstract interpretations of a compositional semantics. The abstract meaning of a module corresponds to its analysis and composition of abstract meanings corresponds to composition of analyses. Such an approach is essential for large program development so that altering one module does not require re-analysis of the entire program. We claim that for a substantial class of programs, compositional analyses which are based on a notion of *abstract unfolding* provide the same precision as non-compositional analysis. A compositional analysis for ground dependencies is included to illustrate the approach. To the best of our knowledge this is the first account of a compositional framework for the analysis of logic programs.

*Department of Computer Science, KU Leuven, Belgium. codish@cs.kuleuven.ac.be.

‡Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA. debray@cs.arizona.edu. Supported in part by the National Science Foundation under grant number CCR-8901283.

§Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. giaco@di.unipi.it. Supported in part by the Esprit Basic Research Action 3012 - Compulog.

In proceedings of ACM POPL'93.

1 Introduction

It is widely acknowledged that as the size of a program increases, it becomes impractical to maintain it as a single monolithic structure. Instead, the program has to be broken up into a number of smaller units called *modules* that provide the desired functionality when combined. Modularity helps reduce the complexity of designing and proving correctness of programs. Modularity helps also in developing adaptable software. Since the program specifications can change while the program itself is being constructed, a modular structure of programs and a corresponding modular analysis can reduce the updating complexity both in program development and in program analysis. In contrast to this situation, however, current works on dataflow analysis of logic programs typically assume that the entire program is available for inspection at the time of analysis. Consequently, it is often not possible to apply existing dataflow analyses to large programs, either because the resource requirements are prohibitively high, or because not all program components are available when we wish to carry out the analysis. This is especially unfortunate because large programs are typically those that stand to benefit most from the results of good dataflow analysis.

In this paper, we give a formal account of how modular logic programs may be analyzed. The basic idea is more or less standard: we consider a semantics to modular programs, then study how such a semantics may be safely approximated and how the results of such approximations may be composed to yield flow analysis results for the entire program. We demonstrate this approach by giving a compositional ground dependencies analysis for modular logic programs. Semantic treatments of modules in logic programs have been given by a number of authors (see, for example, [7, 22]), typically based on nontrivial extensions to Horn clause logic that lead to complex semantics; it appears to us that the development of abstract interpretations based on such semantics is not entirely straightforward. The semantics we consider here as a basis for abstract interpretations is a simplification of that proposed in [4]. The essential idea is to treat modules as programs in which undefined predicates are considered *open*. The meaning of a module is given in terms of iterated unfoldings of the procedures defined in it, except that the open (i.e., imported) predicates are not unfolded—the result is to specify the meaning of a module in terms of structures that depend only on the meaning of the open predicates. It turns out that composition of modules is described using the same semantic function—namely, iterated unfolding—as that used for describing the meaning of a module, leading to a conceptually and mathematically simple and elegant treatment.

This semantics is attractive as a basis for abstraction as it resembles the semantics of [14] which provides the basis for abstract interpretation as described in [2] and [8]. The use of clauses as semantic objects leads to interesting technical complications for abstract interpretation, in that there are two independent dimensions along which we need finite descriptions in the abstract domain, namely, finite descriptions of sets of substitutions (the "usual" dimension), and also finite descriptions of unbounded sequences of atoms. This is not a matter of purely theoretical interest: in most Prolog systems currently available, e.g., BIM and Sicstus Prologs [3, 6], there are no *a priori* restrictions on the dependencies between the different modules in a program (a module A *depends on* a module B if a procedure defined in A calls a procedure defined in B), and it is entirely possible to have a set of modules mutually dependent on each other. As a practical matter, therefore, it is important to be able to deal with modules with arbitrary inter-module dependencies. Our general treatment—which we call *symmetric composition*—shows how these problems may be addressed. However, it may incur a loss in precision because of the need to approximate objects of unbounded size using descriptions of bounded size in a way that is not usually encountered in abstract interpretations. We identify a special case, where the dependencies between modules is *hierarchical*, where it can be guaranteed that there will be no need to sacrifice additional precision when dealing with what we call the *directed composition* of modules.

The rest of the paper is organized as follows: Section 2 presents briefly some preliminary definitions and notations. Section 3 describes the concrete semantics which is the basis for abstraction. Section 4 introduces the general compositional abstract semantics, while Section 5 describes a special case where abstractions are defined in terms of *abstract unfolding* and presents an example for ground dependencies analysis. Section 6 illustrates how our approach can be used for compositional analysis. Section 7 discusses how some restrictions on modules, assumed in earlier sections, can be relaxed. Section 8 discusses related work, and Section 9 concludes.

2 Preliminaries

In the following we assume familiarity with the standard definitions and notation for logic programs [20] and abstract interpretation [11, 12]. Throughout, we will assume a fixed set of function symbols Σ, a fixed set of predicate symbols Π and a fixed denumerable set of variables Var. With each function symbol $f \in \Sigma$ and predicate symbol $p \in \Pi$ is associated a unique natural number called its *arity*: a (predicate or function) symbol f with arity n is written f/n. The non-ground term algebra over Σ and Var is denoted $Term(\Sigma, Var)$ or $Term$ for short. The set of atoms constructed from predicate symbols in Π and terms from $Term$ is denoted $Atom(\Pi, \Sigma, Var)$ or $Atom$ for short. The powerset of a set X is denoted by $\wp(X)$. A *goal* $\bar{a}$ is a sequence of atoms, and is typically written $\langle a_1, \ldots, a_n \rangle$ or simply as $a_1, \ldots, a_n$. We sometimes view $\bar{a}$ as a set and write $b \in \bar{a}$. The empty sequence is denoted by $\langle\ \rangle$. The concatenation of goals $\bar{b}_1$ and $\bar{b}_2$ is denoted $\bar{b}_1 :: \bar{b}_2$. A *Horn clause* is an 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*. 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 $vars(t)$.

A *substitution* is a mapping from Var to $Term$ which acts as the identity almost everywhere: it extends to apply to any syntactic object in the usual way. The identity substitution is denoted ϵ. The set of idempotent substitutions is denoted Sub. Following tradition, the application of a substitution θ to an object t will be written $t\theta$ rather than $\theta(t)$. We fix a partial function mgu which maps a pair of syntactic objects to an idempotent most general unifier of the objects. A statement $\theta = mgu(s, t)$ implies that s and t are unifiable. The notation for mgu is extended as usual for sets of equations. We write $mgu(\langle a_1, \ldots, a_n \rangle, \langle b_1, \ldots, b_n \rangle)$ to denote the most general unifier of the set of equations $\{a_1 = b_1, \ldots, a_n = b_n\}$. Note that $mgu(\langle\ \rangle, \langle\ \rangle) = \epsilon$.

A *variable renaming* is a substitution that is a bijection on Var. Two syntactic objects t_1 and t_2 are *equivalent up to renaming*, written $t_1 \sim t_2$, if $t_1\rho = t_2$ for some variable renaming ρ. The equivalence class of t under $\sim$ is denoted by $[t]_\sim$. Given an equivalence class $\hat{t}$ of syntactic objects and a finite set of variables V, it is always possible to find a representative t of $\hat{t}$ (i.e. an object t such that $[t]_\sim = \hat{t}$) that contains no variables from V. For a syntactic object s and a set of equivalence classes of objects I, we denote by

$\langle c_1, \ldots, c_n \rangle \ll_s I$ that $c_1, \ldots, c_n$ are representatives of elements of I renamed apart from s and from each other, namely, that

1. $[c_i]_\sim \in I$;
2. $vars(c_i) \cap vars(s) = \emptyset$, $1 \leq i \leq n$; and
3. $i \neq j$ implies $vars(c_i) \cap vars(c_j) = \emptyset$, $1 \leq i, j \leq n$.

Note that the empty tuple is trivially renamed apart from any object s and (possibly empty) set I, i.e., $\langle\ \rangle \ll_s I$.

In the discussion that follows, we will be concerned with sets of clauses modulo renaming, i.e., subsets of $[Clause]_\sim$. For simplicity of exposition, we will abuse notation and assume that a clause represents its equivalence class and write *Clause* rather than $[Clause]_\sim$.

We focus on logic programs which are constructed from *predicate disjoint modules* (considered also in [15, 17]). If $P_1, \ldots, P_n$ are logic program modules, then $P = \bigcup_{i=1}^{n} P_i$ is a *modular logic program.* A modular logic program is *predicate disjoint* if the predicates defined in each module are disjoint from those defined in the others. For logic program (or module) P, $open(P)$ denotes the set of predicates that occur in the body of a clause in P but are not defined in P. For any program P, we denote by Φ_P the set $\left\{ [p(\bar{x}) \leftarrow p(\bar{x})]_\sim \;\middle|\; p \in open(P) \right\}$: as we will see, these tautologies play an important role in defining the semantics of P.

3 Bottom-up semantics for composition

This section presents the semantic basis for compositional abstract interpretations. The semantics is an instance of the compositional bottom-up semantics of Bossi *et al.* [4], specialized for the case of predicate disjoint modules. We propose two notions of composition which provide the basis for composition of analyses. The first, "symmetric composition", is that introduced in [4]. It is more general and applicable to the analysis of arbitrary modular logic programs. However the second, "directed composition", provides for potentially more precise analyses when programs have a hierarchical structure. It allows us to analyze a module while "plugging in" analyses for predicates defined in other modules that are lower in the hierarchy.

Definition 3.1 *An* interpretation *is any element in* $Int = \wp(Clause)$.

The concrete semantics is formalized in terms of unfolding of clauses. The unfolding operator *unf* specifies the result of unfolding clauses from an interpretation P_1 with clauses from an interpretation P_2.

Definition 3.2 *[unfolding] : The unfolding operator* $unf : Int \times Int \to Int$ *is defined as*

$$unf(P_1, P_2) = \left\{ \hat{c} \;\middle|\; \begin{array}{l} \hat{c} = [(h \leftarrow \bar{b}_1 :: \quad \cdots \quad :: \bar{b}_n)\sigma]_\sim, \\ [c]_\sim = [h \leftarrow g_1, \ldots, g_n]_\sim \in P_1,\ (n \geq 0), \\ \langle h_1 \leftarrow \bar{b}_1, \ldots, h_n \leftarrow \bar{b}_n \rangle \ll_c P_2, \\ \sigma = mgu(\langle g_1, \ldots, g_n \rangle, \langle h_1, \ldots, h_n \rangle) \end{array} \right\}.$$

Intuitively, the unfolding operator yields every possible way to unfold each literal in each clause of P_1 once using clauses in P_2. This operator is of interest as it can be applied to formalize both top-down and bottom-up semantics for logic programs [19]. The following formalizes a bottom-up semantics for open logic programs in terms of "iterated unfolding" — that is, repeatedly unfolding the clauses in a program until further unfolding produces no change:

Definition 3.3 *[fixpoint semantics [4]] : The fixpoint semantics of a program P is given by the function $\mathcal{F} : Int \to Int$, defined as $\mathcal{F}(P) = lfp(T_P^{\mathcal{C}})$, where $T_P^{\mathcal{C}} : Int \to Int$ is defined as $T_P^{\mathcal{C}}(I) = unf(P, I \cup \Phi_P)$.*

Several aspects of this definition demand explanation. First, compare with the standard T_P fixpoint semantics for Horn logic programs. The T_P operator infers (ground) facts from the clauses in the program given facts in an interpretation for the body goals. This process can also be viewed as a kind of unfolding where facts (i.e, unit clauses) are used to unfold program clauses. Once we consider modular programs, however, this approach is too simplistic because if we wish to give a compositional semantics then open predicates, i.e., those imported from other modules, have no definition available. The solution to this problem proposed in [4] is to unfold only those predicates which have a definition, so that the meaning of a program becomes dependent (as intuitively it should) on the meaning of its open predicates. Technically, this is accomplished by letting Φ_P to add tautological clauses for the open predicates and to unfold all predicates in the body of a clause. Notice that the unfolding of an undefined predicate with a tautological clause is basically a "no-op." Moreover, note that: If $open(P) = \emptyset$ then $\Phi_P = \emptyset$ and $T_P^{\mathcal{C}}(I) = unf(P, I)$. When I consists of unit clauses (facts), the fixpoint operator in this case gives precisely the generalized (non-ground) fixpoint semantics of [14].

Proposition 3.4 *[symmetric composition [4]] : Let P_1 and P_2 be modules, then $\mathcal{F}(P_1 \cup P_2) = \mathcal{F}(\mathcal{F}(P_1) \cup \mathcal{F}(P_2))$.*

Example 1 *Consider the logic program P (a portion of a quicksort program), consisting of the following two modules:*

$$
\begin{array}{ll}
P_{sp}: & split(X,[\,],[\,],[\,]). \\
 & split(X,[Y|L],[Y|L1],L2) \leftarrow \\
 & \quad gt(X,Y),\ split(X,L,L1,L2). \\
 & split(X,[Y|L],L1,[Y|L2]) \leftarrow \\
 & \quad le(X,Y),\ split(X,L,L1,L2).
\end{array}
$$

$$
\begin{array}{ll}
P_{lg}: & gt(s(0),0). \\
 & gt(s(X),s(Y)) \leftarrow gt(X,Y). \\
 & \\
 & le(0,0). \\
 & le(0,s(0)). \\
 & le(s(X),s(Y)) \leftarrow le(X,Y).
\end{array}
$$

The unfoldings of P_{sp} specify the possible ways of splitting a list of values into two lists of values; those "larger than" a given X and those "smaller than" X. In the module P_{sp} the domain of values and the interpretation of "larger" and "smaller" are open, since the predicates le/2 and gt/2 are open. Evaluation of $\mathcal{F}(P_{sp})$ proceeds as follows:[1]

1. $split(X,[\,],[\,],[\,]).$
2. $split(X,[Y_1],[Y_1],[\,]) \leftarrow gt(X,Y_1).$
   $split(X,[Y_1],[\,],[Y_1]) \leftarrow le(X,Y_1).$
3. $split(X,[Y_1,Y_2],[Y_1,Y_2],[\,]) \leftarrow gt(X,Y_1), gt(X,Y_2).$
   $split(X,[Y_1,Y_2],[Y_1],[Y_2]) \leftarrow gt(X,Y_1), le(X,Y_2).$
   $split(X,[Y_1,Y_2],[Y_2],[Y_1]) \leftarrow gt(X,Y_2), le(X,Y_1).$
   $split(X,[Y_1,Y_2],[\,],[Y_1,Y_2]) \leftarrow le(X,Y_1), le(X,Y_2).$

etc.

The unfoldings of P_{lg} specify the relations "less or equal" and "greater than" on integers. Since $open(P_{lg}) = \emptyset$, the result corresponds to the semantics of [14]: {gt(s(0),0), le(0,0), le(0,s(0)), gt(s(s(0)),s(0)), …}. The meaning of $P_{sp} \cup P_{lg}$ can be evaluated directly or by applying Proposition 3.4. In either case the result corresponds to the standard meaning as provided by the semantics of [14].

4 Abstract semantics and composition

We assume the standard framework of abstract interpretation as defined in [11] in terms of Galois insertions. We let $(AInt, \sqcup, \sqcap, \sqsubseteq)$ denote a complete lattice of *abstract models*, where each abstract model describes a set of clauses. $(Int, \alpha, AInt, \gamma)$ is a Galois insertion, i.e., $\alpha : Int \rightarrow AInt$ and $\gamma : AInt \rightarrow Int$ are monotonic mappings, and additionally, $\alpha(\gamma(I)) = I$ and $I' \sqsubseteq \gamma(\alpha(I'))$ for each $I' \in Int$ and $I \in AInt$.

The abstract semantics is a function $\mathcal{F}^{\mathcal{A}} : AInt \rightarrow AInt$ which assigns an abstract model to abstractions of programs. The abstract meaning of a program P is $\mathcal{F}^{\mathcal{A}}(\alpha(P))$. For now, we assume only that $\mathcal{F}^{\mathcal{A}}$ is *safe* with respect to the concrete semantics, namely, that for every $I \in Int$, $\alpha(\mathcal{F}(I)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\alpha(I))$.

Abstract (symmetric) composition is analogous to concrete composition. The following theorem states the correctness of applying abstract composition for program analyses based on any safe abstract semantics.

Theorem 4.1 *[correctness of abstract composition] Let $(Int, \alpha, AInt, \gamma)$ be a Galois insertion and let $\mathcal{F}^{\mathcal{A}} : AInt \rightarrow AInt$ be a monotonic and safe approximation of $\mathcal{F}$. Then, for any program modules $P_1, P_2 \in Int$, $\alpha(\mathcal{F}(P_1 \cup P_2)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P_1)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(P_2)))$.*

PROOF. Assume the premise of the theorem and let $P_1, P_2 \in Int$. Recall that α is continuous in any Galois insertion [11].

$$
\begin{array}{rcl}
\alpha(\mathcal{F}(P_1 \cup P_2)) & = & \alpha(\mathcal{F}(\mathcal{F}(P_1) \cup \mathcal{F}(P_2))) \\
 & & \qquad [\ \text{Proposition 3.4}\ ] \\
 & \sqsubseteq & \mathcal{F}^{\mathcal{A}}(\alpha(\mathcal{F}(P_1) \cup \mathcal{F}(P_2))) \\
 & & \qquad [\ \text{safety}\ ] \\
 & = & \mathcal{F}^{\mathcal{A}}(\alpha(\mathcal{F}(P_1)) \sqcup \alpha(\mathcal{F}(P_2))) \\
 & & \qquad [\ \alpha \text{ continuity}\ ] \\
 & \sqsubseteq & \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P_1)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(P_2))) \\
 & & \qquad [\ \text{safety and monotonicity}\ ]
\end{array}
$$
□

5 Compositional Analysis

In this section, we illustrate the ideas sketched in the previous section in a concrete way. We focus on abstract interpretations induced from a set of abstract substitutions *ASub*, and illustrate two examples of compositional analyses for detecting "ground dependencies"—which describe the manner in which the groundness of a variable in a clause depends on the groundness of other variables—induced from an appropriate domain of abstract substitutions. In the first example we introduce the set *VClause* of clauses in which all the terms are distinct variables. Abstract substitutions describe instances of these clauses providing a domain of abstract interpretations. This domain illustrates the technical problems that can arise in the analysis of general programs. In the second example, we apply an additional level of abstraction to

[1] We illustrate the clauses added by successive iterations of unfolding.

handle this problem. In the following let $(ASub, \sqsubseteq)$ be a complete lattice of abstract substitutions, let $(\wp(Sub), \alpha_S, ASub, \gamma_S)$ be a Galois insertion and let

$$VClause = \left\{ [c]_\sim \in Clause(\Pi, \emptyset, Var) \;\middle|\; \begin{array}{l}\text{no variable occurs}\\ \text{more than once in } c\end{array} \right\}.$$

Each element of $VClause$ is an equivalence class of clauses modulo renaming, and is syntactically represented as a clause. The idea is to associate each representative of $VClause$ with an abstract substitution that describes a set of its instances. This is accomplished in Definitions 5.2 and 5.3 (below) by formalizing abstract interpretations as mappings from $VClause$ to $ASub$. The ordering (likewise the join and the meet) on $VClause \to ASub$ is determined by the ordering on $ASub$. Namely, for $I_1^a, I_2^a \in VClause \to ASub$, $I_1^a \sqsubseteq I_2^a \Leftrightarrow \forall c.\ I_1^a(c) \sqsubseteq I_2^a(c)$. In the following it is convenient to view an abstract interpretation $I^a : VClause \to ASub$ as an equivalent binary relation:

$$\{ \langle c, \kappa \rangle \mid c \in VClause,\ \kappa = I^a(c) \}.$$

Example 2 *If $ASub = \wp(Sub)$ then the relation gt from Example 1 is described by*

$$\left\{ \begin{array}{l} \langle gt(x_1, x_2); \{x_1 \mapsto s(0), x_2 \mapsto 0\}\rangle, \\ \langle gt(x_1, x_2) \leftarrow gt(x_3, x_4); \{x_1 \mapsto s(x_3), x_2 \mapsto x_4\}\rangle \end{array} \right\}.$$

As an example of a domain of abstract substitutions, consider the domain Dep adopted from [9]:

Definition 5.1 *[dependency relation] : A relation R over a lattice X is additive iff $(x\ R\ x' \wedge y\ R\ y') \Rightarrow (x \sqcup y)\ R\ (x' \sqcup y')$. A dependency relation R is an additive equivalence relation (reflexive, symmetric and transitive) over $\wp(Var)$. We let Dep denote the complete lattice of dependency relations ordered by implication (containment).*

For notational convenience, we let an arbitrary relation represent the smallest dependency relation implying (i.e., containing) it. Furthermore, we let "$W_1 \leftrightarrow W_1', \ldots, W_n \leftrightarrow W_n'$" denote the relation $\{(W_1, W_1'), \ldots, (W_n, W_n')\}$ and drop set brackets when sets are singleton.

A dependency relation κ describes those substitutions θ satisfying the condition that for every $(V, W) \in \kappa$, the terms in $V\theta$ are ground iff the terms in $W\theta$ are ground. A particular case is when $V = \emptyset$ (or respectively $W = \emptyset$); in this case it means that the terms in $W\theta$ (or respectively $V\theta$) are definitely ground. The corresponding abstract interpretation is defined by the following: Let $\Theta \in \wp(Sub)$ and $\kappa \in Dep$. Define $\alpha_D : \wp(Sub) \to Dep$ and $\gamma_D : Dep \to \wp(Sub)$ by:

$$\alpha_D(\Theta) = \left\{ (V, W) \;\middle|\; \begin{array}{l} \forall \theta \in \Theta. \\ \quad vars(V\theta) = vars(W\theta) \end{array} \right\};$$

$$\gamma_D(\kappa) = \left\{ \theta \;\middle|\; \begin{array}{l} \forall (V, W) \in \kappa. \\ \quad vars(V\theta) = vars(W\theta) \end{array} \right\}.$$

Example 3 *Let $\theta = \{x \mapsto 0,\ y \mapsto 0\}$ and $\theta' = \{x \mapsto 0,\ y \mapsto s(0)\}$. Then $\alpha_D(\{\theta, \theta'\})$ is the smallest dependency relation which contains $\{x, y\} \leftrightarrow \emptyset$. Note that in our notation this is written simply as $\alpha_D(\{\theta, \theta'\}) = \{x, y\} \leftrightarrow \emptyset$.*

It is straightforward to prove that $(\wp(Sub), \alpha_D, Dep, \gamma_D)$ is a Galois insertion. We now describe how a domain of abstract interpretations is induced from $(\wp(Sub), \alpha_S, ASub, \gamma_S)$.

Definition 5.2 *[abstract interpretations I] : Define $\bar{\gamma} : (VClause \to ASub) \to Int$ and $\bar{\alpha} : Int \to (VClause \to ASub)$ by:*

$$\bar{\gamma}(I^a) = \left\{ [c\theta]_\sim \;\middle|\; \begin{array}{l} \langle c, \kappa \rangle \in I^a, \\ \theta \in \gamma_S(\kappa) \end{array} \right\}; \textit{ and}$$

$$\bar{\alpha}(I) = \left\{ \langle c, \kappa \rangle \;\middle|\; \begin{array}{l} c \in VClause, \\ \kappa = \alpha_S\{mgu(c, c') \mid c' \ll_c I\} \end{array} \right\}.$$

Example 4
Let $ASub = Dep$ and

$$I = \left\{ \begin{array}{l} le(0, 0), \\ le(0, s(0)), \\ le(s(X), s(Y)) \leftarrow le(X, Y) \end{array} \right\}.$$

Then,

$$\bar{\alpha}(I) = \left\{ \begin{array}{l} \langle le(x, y); \{x, y\} \leftrightarrow \emptyset\rangle, \\ \langle le(x, y) \leftarrow le(x', y'); x \leftrightarrow x', y \leftrightarrow y'\rangle \end{array} \right\}.$$

In the following we denote $AInt_D$ the domain of abstract interpretations where $ASub = Dep$.

Unfortunately $\bar{\alpha}$ and $\bar{\gamma}$ as defined above do not provide a Galois insertion as $\bar{\gamma}$ is not injective. This means that several distinct elements in $(VClause \to ASub)$ describe the same set of clauses. However this is easily fixed, as suggested in [12], by letting $\bar{\gamma}$ induce an equivalence relation $\equiv$ on $(VClause \to ASub)$:

Example 5
If $ASub = Dep$, $I_1^a = \{\langle le(x, y); \{x, y\} \leftrightarrow \emptyset\rangle\}$ and $I_2^a = \{\langle le(x, y); \{x, y, z\} \leftrightarrow \emptyset\rangle\}$. Then, $\bar{\gamma}(I_1^a) = \bar{\gamma}(I_2^a)$ and consists of all ground instances of $le(x, y)$.

Definition 5.3 *[abstract interpretations II] : Let $AInt = (VClause \to ASub)/_\equiv$ where $\equiv$ is the equivalence relation induced by $\bar{\gamma}$ on $(VClause \to ASub)$; i.e., $I_1^a \equiv I_2^a$ iff $\bar{\gamma}(I_1^a) = \bar{\gamma}(I_2^a)$. Define $\gamma : AInt \to Int$ and $\alpha : Int \to AInt$ by lifting $\bar{\gamma}$ and $\bar{\alpha}$ respectively: i.e., $\gamma([I^a]_\equiv) = \bar{\gamma}(I^a)$ and $\alpha(I) = [\bar{\alpha}(I)]_\equiv$.*

Notice that the equivalence relation $\equiv$ also provides variable hiding. Given a clause description $\langle c, \kappa \rangle$, where $c \in VClause$ and $\kappa \in ASub$, the relevant variables for the analysis are only those in $vars(c)$. The intuition is that if $\{\langle c, \kappa\rangle\} \equiv \{\langle c, \kappa'\rangle\}$, and $\kappa \neq \kappa'$ then κ and κ' describe the same set of abstract substitutions, when restricted to the variables in $vars(c)$. We do not require $ASub$ to be a finite height lattice (in fact Dep is not of finite height). However, we require that for every $c \in VClause$, the set $\{[\langle c, \kappa\rangle]_\equiv \mid \kappa \in ASub\}$ is finite. It is not difficult to show that $(Int, \alpha, AInt, \gamma)$ is a Galois insertion.

A safe abstract semantics can now be defined in terms of *abstract unfolding* which is in turn defined in terms of an abstract unification function $mgu^{\mathcal{A}} : (Atom^* \times ASub) \times (Atom \times ASub)^* \to ASub$ which is assumed to satisfy the (safety) condition that if $mgu^{\mathcal{A}}(\langle \bar{a}; \kappa_0\rangle, \langle\langle b_1; \kappa_1\rangle, \ldots, \langle b_n; \kappa_n\rangle\rangle) = \kappa$, $\theta_i \in \gamma_S(\kappa_i)$ $(0 \leq i \leq n)$, and $mgu(\bar{a}\theta_0, \langle b_1\theta_1, \ldots, b_n\theta_n\rangle) = \theta$, then $\theta \in \gamma(\kappa)$.

Example 6 *The following is a safe abstract unification function for Dep similar to that introduced in [9].*

$$mgu_D^{\mathcal{A}}(\langle \bar{a}; \kappa_0\rangle, \langle\langle b_1; \kappa_1\rangle, \ldots, \langle b_n; \kappa_n\rangle\rangle) = \bigcup_{i=0}^{n} \kappa_i \cup \left\{ \ (\{x\}, vars(t)) \ \middle| \ x \mapsto t \in mgu(\bar{a}, \bar{b}) \ \right\}.$$

where $\bar{b} = \langle b_1, \ldots, b_n\rangle$. *For instance:*

$$mgu_D^{\mathcal{A}}(\langle gt(x', y'); \emptyset\rangle, \langle gt(x'', y''); x'' \leftrightarrow \emptyset, y'' \leftrightarrow \emptyset\rangle) = x' \leftrightarrow x'' \leftrightarrow \emptyset, y' \leftrightarrow y'' \leftrightarrow \emptyset$$

indicating that x', y', x'', *and* y'' *are all ground.*

Definition 5.4 *[abstract unfolding]*[2] *: The abstract unfolding operator* $unf^{\mathcal{A}} : AInt \times AInt \to AInt$ *is defined as:*

$$unf^{\mathcal{A}}(P_1, P_2) = \bigsqcup \left\{ \bar{c} \ \middle| \ \begin{array}{l} \bar{c} = \langle h \leftarrow \bar{b}_1 :: \quad \cdots \quad :: \bar{b}_n; \hat{\kappa}\rangle, \\ c = \langle h \leftarrow g_1, \ldots, g_n; \kappa\rangle \in P_1, \\ \langle\langle h_1 \leftarrow \bar{b}_1; \kappa_1\rangle, \ldots, \\ \qquad\qquad \langle h_n \leftarrow \bar{b}_n; \kappa_n\rangle\rangle \ll_c P_2, \\ \hat{\kappa} = mgu^{\mathcal{A}} \left(\begin{array}{l} \langle\langle g_1, \ldots, g_n\rangle, \kappa\rangle, \\ \langle\langle h_1; \kappa_1\rangle, \ldots, \langle h_n; \kappa_n\rangle\rangle \end{array} \right) \end{array} \right\}.$$

The abstract fixpoint semantics is now defined in terms of abstract unfolding by:

Definition 5.5 *[abstract fixpoint semantics] : Define* $\mathcal{F}^{\mathcal{A}} : AInt \to AInt$ *as* $\mathcal{F}^{\mathcal{A}}(P^a) = lfp(T^{\mathcal{A}}_{Pa})$ *where* $T^{\mathcal{A}}_{Pa} : AInt \to AInt$ *is defined by*

$$T^{\mathcal{A}}_{Pa}(I^a) = unf^{\mathcal{A}}(P^a, I^a \sqcup \Phi'_{Pa})$$

and Φ'_{Pa} *is the natural extension of* Φ_P *for abstract programs and has the property that* $\Phi'_{Pa} = \alpha(\Phi_P)$.

Proposition 5.6 *If* $mgu^{\mathcal{A}}$ *is a safe abstract unification function, then* $unf^{\mathcal{A}}$ *is a safe abstract unfolding and* $\mathcal{F}^{\mathcal{A}}$ *is a safe abstract semantics, namely, for every* $I_1, I_2 \in Int$, *(i)* $\alpha(unf(I_1, I_2)) \sqsubseteq unf^{\mathcal{A}}(\alpha(I_1), \alpha(I_2))$; *and (ii)* $\alpha(\mathcal{F}(I_1)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\alpha(I_1))$.

Theorem 4.1 can now be applied to justify compositional analyses. However, it is interesting to note that when $AInt$ is induced from $ASub$ we can prove a stronger result which implies that composition does not introduce additional loss of precision, i.e., that $\mathcal{F}^{\mathcal{A}}(I_1^a \sqcup I_2^a) = \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(I_1^a) \sqcup \mathcal{F}^{\mathcal{A}}(I_2^a))$ The proof is similar to that of Proposition 3.4. It relies on the observation that (abstract) unfolding is an associative binary operator that is left-distributive over the composition of programs, and that $\mathcal{F}^{\mathcal{A}}$ is idempotent.

Symmetric Composition: Analysis of General Modules

As mentioned earlier, existing Prolog implementations allow arbitrary inter-module dependencies. An interesting technical problem arises in this case: abstract unfolding may introduce arbitrarily large clauses so that analyses can no longer be guaranteed to terminate. This necessitates a second (and orthogonal) abstraction to deal with unbounded clause bodies in the abstract semantics. One proposal to deal with abstract domains containing infinite chains is to use some kind of widening/narrowing approach to restrict the analysis to a finite subspace of the entire domain [13]. Here we consider a somewhat simpler solution that can be formalized in the standard framework of abstract interpretations by restricting $AInt$ to be a finite height lattice. We apply a further level of abstraction to provide finitary descriptions of (sets of) arbitrarily large abstract clauses. A domain $VClause_\star \subseteq VClause$ in which clauses are restricted to have bodies containing at most one occurrence of a predicate symbol is introduced. In this case, since Π may be assumed to be finite, the new abstract domain, denoted $AInt^\star$, becomes finite. This abstraction, called *star abstraction* and originally introduced in [9], provides an appropriate framework to develop compositional analyses. We demonstrate this for the case of ground dependencies analysis. The basic idea is to collapse all occurrences of the same predicate in a body to one "canonical" atom, representing any possible sequence of atoms with that predicate symbol. The collapsing of a sequence $p(\bar{x}_1), \ldots, p(\bar{x}_m)$ of atoms is a pair $\langle p(\bar{x}); \kappa\rangle$ where $\kappa \in Dep$ captures the intuition that each argument x_j of $p(\bar{x})$ represents the set of the j^{th} arguments in $p(\bar{x}_1), \ldots, p(\bar{x}_m)$. The following definition formalizes this as an abstraction function:

[2] Elements of $ASub$ are, in general, infinite objects. To formally "rename apart" objects of $AInt$ it is necessary to assume that every element of $ASub$ can be represented by a finite object, which is not unreasonable.

Definition 5.7 *[star abstraction] :* $\alpha_\star : AInt_D \rightarrow AInt_D^\star$ *is defined by:*

$$\alpha_\star(I^a) =$$
$$\sqcup \left\{ \bar{c} \;\middle|\; \begin{array}{l} \bar{c} = (h \leftarrow b_1 :: \ldots :: b_m ;\ \bigcup_{i=0}^{m} \kappa_i), \\ \langle h \leftarrow \bar{b}, \kappa_0 \rangle \in I^a, \\ \langle \langle b_1 ; \kappa_1 \rangle, \ldots, \langle b_m ; \kappa_m \rangle \rangle \ll_h collapse(h \leftarrow \bar{b}) \end{array} \right\}$$

where $collapse : VClause \rightarrow [\wp(Atom \times Dep)]_\sim$ *is defined by:*

$$collapse(h \leftarrow \bar{b}) =$$
$$\left\{ \hat{c} \;\middle|\; \begin{array}{l} \hat{c} = [\langle p(\bar{x}); x_1 \leftrightarrow X_1, \ldots, x_n \leftrightarrow X_n \rangle]_\sim, \\ p/n \in \Pi,\ \bar{x} = \{x_1, \ldots, x_n\}, \\ \bar{x} \cap vars(h \leftarrow \bar{b}) = \emptyset,\ 1 \leq i \leq n, \\ X_i = \{y_i | p(y_1, \ldots, y_n) \in \bar{b}\} \neq \emptyset \end{array} \right\}$$

Example 7
Let $I^a \in AInt_D$ *be*

$$\left\{ \left\langle \begin{array}{c} split(x_1, x_2, x_3, x_4) \leftarrow gt(x_5, x_6), gt(x_7, x_8); \\ x_1 \leftrightarrow \{x_5, x_7\}, x_2 \leftrightarrow x_3 \leftrightarrow \{x_6, x_8\} \end{array} \right\rangle \right\}.$$

Then,

$$\alpha_\star(I^a) = \left\{ \left\langle \begin{array}{c} split(y_1, y_2, y_3, y_4) \leftarrow gt(y_5, y_6); \\ y_1 \leftrightarrow y_5, y_2 \leftrightarrow y_3 \leftrightarrow y_6 \end{array} \right\rangle \right\}.$$

Notice that as a result of star abstraction, multiple occurrences of $gt(\ldots)$ *goals have been "collapsed" into a single occurrence.*

We do not fully formalize here the star abstraction as a Galois insertion (and hence as an abstract interpretation in our framework). However, we observe that $\alpha_\star$ is a complete join-morphism which implies that an adjoint concretization mapping $\gamma_\star : AInt_D^\star \rightarrow AInt_D$ does exist and is determined by $\gamma_\star(I) = \cup \{ I' \mid \alpha_\star(I') = I \}$. Since the composition of Galois insertions is also a Galois insertion [12], $(Int, \alpha_\star \circ \alpha, AInt_D^\star, \gamma \circ \gamma_\star)$ provides a suitable basis for abstract interpretation. It is the second level of (star) abstraction which guarantees termination of compositional ground dependencies analysis for arbitrary logic program (modules) by taking for $unf^{\mathcal{A}}$ in Definition 5.5 the function $unf^\star : AInt_D^\star \times AInt_D^\star \rightarrow AInt_D^\star$ defined by

$$unf^\star(I_1^a, I_2^a) = \alpha_\star(unf^{\mathcal{A}}(I_1^a, I_2^a)).$$

The corresponding function $\mathcal{F}^{\mathcal{A}} : AInt_D^\star \rightarrow AInt_D^\star$ is denoted by $\mathcal{F}^\star$. The resulting abstract semantics can also be thought of as being produced by an abstract unfolding operator $unf^{\mathcal{A}}$ that never produces multiple literals with the same predicate symbol in an unfolded clause body, instead collapsing them into a single canonical literal—that is, where all the "action" takes place during abstract unfolding rather than in a separate abstraction step.

We illustrate (the result of) a compositional ground dependency analysis for the *split* relation defined in Example 1. The modules P_{sp} and P_{lg} are analyzed independently and then the results are composed.

Example 8 *Recall the program* $P_{sp} \cup P_{lg}$ *from Example 1. The abstract meanings of the two modules* P_{sp} *and* P_{lg} *are given by* $\mathcal{F}^\star(P_{sp})$ *and* $\mathcal{F}^\star(P_{lg})$ *respectively:*

$$\mathcal{F}^\star(P_{sp}) =$$
$$\left\{ \begin{array}{l} \langle split(x_1, x_2, x_3, x_4); x_2 \leftrightarrow x_3 \leftrightarrow x_4 \leftrightarrow \emptyset \rangle, \\ \langle split(x_1, x_2, x_3, x_4) \leftarrow gt(x_5, x_6); \\ \qquad x_2 \leftrightarrow x_3 \leftrightarrow x_6, x_1 \leftrightarrow x_5, x_4 \leftrightarrow \emptyset \rangle, \\ \langle split(x_1, x_2, x_3, x_4) \leftarrow le(x_5, x_6); \\ \qquad x_2 \leftrightarrow x_4 \leftrightarrow x_6, x_1 \leftrightarrow x_5, x_3 \leftrightarrow \emptyset \rangle, \\ \langle split(x_1, x_2, x_3, x_4) \leftarrow gt(x_5, x_6), le(x_7, x_8); \\ \qquad x_2 \leftrightarrow \{x_6, x_8\}, x_3 \leftrightarrow x_6, x_4 \leftrightarrow x_8, \\ \qquad\qquad x_1 \leftrightarrow x_5 \leftrightarrow x_7 \rangle \end{array} \right\}.$$

$$\mathcal{F}^\star(P_{lg}) = \left\{ \begin{array}{l} \langle gt(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset \rangle, \\ \langle le(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset \rangle \end{array} \right\}.$$

The abstract meaning of the composition of the two modules is then obtained as

$$\mathcal{F}^\star(\mathcal{F}^\star(P_{sp}) \sqcup \mathcal{F}^\star(P_{lg})) =$$
$$\left\{ \begin{array}{l} \langle split(x_1, x_2, x_3, x_4); x_2 \leftrightarrow x_3 \leftrightarrow x_4 \leftrightarrow \emptyset \rangle, \\ \langle gt(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset \rangle, \\ \langle le(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset \rangle \end{array} \right\}.$$

Intuitively, this is what we expect: using the abstract semantics of the module P_{lg}*, we have inferred that the second, third, and fourth arguments of the predicate split must be ground. This is in fact the best we can do — the first argument of split may in fact not be ground, given the first clause defining this predicate (see Example 1).*

Directed Composition: Analysis of Hierarchical Programs

As discussed above, a compositional analysis that first analyzes different modules in isolation, then composes the resulting analyses, may have to deal with the possibility of unbounded clause bodies during analysis, typically by sacrificing some precision to gain termination. However, a common program design technique is to structure different modules in a hierarchical way, so that components of a program are defined and understood in terms of previously defined components. If the modules in a program are structured hierarchically, it is possible to take advantage of this fact and obtain a compositional evaluation of the program that does not involve clause structures of unbounded size.

The underlying idea is quite straightforward. Consider the program of Example 1, where P_{lg} is lower in the hierarchy of modules than P_{sp}. We can use the (abstract) meaning of P_{lg} to evaluate the (abstract) meaning of P, i.e., considering unfoldings of $P' = P_{sp} \cup \mathcal{F}(P_{lg})$. While P' is an infinite program, it will have a finite abstraction. Furthermore, as $open(P') = \emptyset$, unfoldings will produce only unit clauses, i.e., clauses with empty bodies. When viewed as a program analysis, this corresponds to "plugging" the analysis of P_{lg} into the analysis of P_{sp} instead of composing the respective analyses. While a "pure" composition is preferable, the latter may provide more precise results as it requires less abstraction, and simpler abstract models for the program.

Proposition 5.8 *[directed composition]*
For any two (predicate disjoint) modules P_1 and P_2, $\mathcal{F}(P_1 \cup P_2) = \mathcal{F}(P_1 \cup \mathcal{F}(P_2))$.

Proof. Consider any two modules P_1 and P_2. We have

$$\begin{array}{rcl} \mathcal{F}(P_1 \cup \mathcal{F}(P_2)) & = & \mathcal{F}(\mathcal{F}(P_1) \cup \mathcal{F}(\mathcal{F}(P_2))) \\ & & \quad [\ \text{by Proposition 3.4}\ ] \\ & = & \mathcal{F}(\mathcal{F}(P_1) \cup \mathcal{F}(P_2)) \\ & & \quad [\ \mathcal{F}\ \text{is idempotent}\ ] \\ & = & \mathcal{F}(P_1 \cup P_2) \\ & & \quad [\ \text{by Proposition 3.4}\ ] \end{array}$$

□

The following result shows that a "bottom-up" composition of modules is sound:

Theorem 5.9 *Let $(Int, \alpha, AInt, \gamma)$ be a Galois insertion and let $\mathcal{F}^{\mathcal{A}} : AInt \rightarrow AInt$ be a monotonic and safe approximation of $\mathcal{F}$. Then, for any program modules $P_1, P_2 \in Int$, we have $\alpha(\mathcal{F}(P_1 \cup P_2)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\alpha(P_1) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(P_2)))$.*

The following example illustrates a hierarchical dependency analysis (i.e. taking $ASub = Dep$).

Example 9 *Consider the logic program defining the qs relation for a quicksort program, where split is defined in Example 1:*

$$\begin{array}{ll} P_{qs}: & qs([\,],[\,]). \\ & qs([X|Xs], Ys) \leftarrow \\ & \quad split(X, Xs, L1, L2),\ qs(L1, Ls), \\ & \quad qs(L2, Bs),\ append(Ls, [X|Bs], Ys). \end{array}$$

$$\begin{array}{ll} P_{app}: & append([\,], X, X). \\ & append([X|W], Y, [X|Z]) \leftarrow append(W, Y, Z). \end{array}$$

The analysis starts from the module P_{app}. The abstract meaning of append is obtained as:
$lfp(T^{\mathcal{A}}_{\alpha(P_{app})}) =$

$$\left\{ \begin{array}{l} \langle append(x_1, x_2, x_3); \\ \qquad [x_1 \leftrightarrow \emptyset, x_2 \leftrightarrow x_3] \sqcup [\{x_1, x_2\} \leftrightarrow x_3] \rangle \end{array} \right\}.$$

Notice that the intersection (the lub on Dep) of (the smallest dependency relation containing) $\{x_1, x_2\} \leftrightarrow x_3$ and (the smallest dependency relation containing) $x_1 \leftrightarrow \emptyset, x_2 \leftrightarrow x_3$ is (the smallest dependency relation containing) $\{x_1, x_2\} \leftrightarrow x_3$. So $\mathcal{F}^{\mathcal{A}}(\alpha(P_{app})) = \{\langle append(x_1, x_2, x_3); \{x_1, x_2\} \leftrightarrow x_3\rangle\}$. The abstract meaning of P_{lg} (from Example 1) is

$\alpha(P_{lg}) =$

$$\left\{ \begin{array}{l} \langle gt(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset\rangle, \\ \langle gt(x, y) \leftarrow gt(x', y'); x \leftrightarrow x', y \leftrightarrow y'\rangle, \\ \langle le(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset\rangle, \\ \langle le(x, y) \leftarrow le(x', y'); x \leftrightarrow x', y \leftrightarrow y'\rangle \end{array} \right\}.$$

$$\mathcal{F}^{\mathcal{A}}(\alpha(P_{lg})) = \left\{ \begin{array}{l} \langle gt(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset\rangle, \\ \langle le(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset\rangle \end{array} \right\}.$$

To approximate the meaning of $P_{sp} \cup P_{lg}$ we apply (abstract) directed composition:

$\mathcal{F}^{\mathcal{A}}(\alpha(P_{sp}) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(P_{lg}))) =$

$$\left\{ \begin{array}{l} \langle split(x_1, x_2, x_3, x_4); x_2 \leftrightarrow x_3 \leftrightarrow x_4 \leftrightarrow \emptyset\rangle, \\ \langle gt(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset\rangle, \\ \langle le(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset\rangle \end{array} \right\}$$

Observe that the result is the same as evaluation of $\mathcal{F}^{\mathcal{A}}\alpha(P_{sp} \cup P_{lg})$. An additional application of (abstract) directed composition provides the following approximation of the meaning of $P_{qs} \cup P_{app} \cup P_{sp} \cup P_{lg}$:

$$\left\{ \begin{array}{l} \langle qs(x_1, x_2); x_1 \leftrightarrow x_2\rangle, \\ \langle append(x_1, x_2, x_3); \{x_1, x_2\} \leftrightarrow x_3\rangle, \\ \langle gt(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset\rangle, \\ \langle le(x, y); x \leftrightarrow \emptyset, y \leftrightarrow \emptyset\rangle, \\ \langle split(x_1, x_2, x_3, x_4); x_2 \leftrightarrow x_3 \leftrightarrow x_4 \leftrightarrow \emptyset\rangle \end{array} \right\}$$

Abstract Composition: Precision vs. Termination

The first part of this section demonstrates that, in general, the (symmetric) composition of program analyses may require an additional layer of abstraction implying a potential loss of precision. The second part illustrates that a weaker form of (directed) composition can be applied to analyze programs with a hierarchical structure. In this case analyses are potentially more precise. However, this approach is limited to programs with a hierarchical structure and in particular to *closed* programs (i.e., programs in which every predicate is defined in some module); moreover, the composition *is* weaker and in particular, a module cannot be analyzed until all "lower" modules are available and have been analyzed.

In the following we provide some syntactic characterizations which strengthen both of the above approaches to compositional program analysis. The first characterization identifies a class of *bounded* program modules. Unfolding clauses in such modules does not create clauses of unbound length. Consequently, if a program consists of bounded modules then a single layer of abstraction is sufficient for symmetric composition of analyses. The basic idea is to detect the absence of loops in the program's call graph which might cause a problem. Note that not all loops create unbounded unfoldings. A convenient way to express this criterion is by way of a context free grammar.

The second characterization identifies a class of *semi-hierarchical* programs which can be analyzed using one layer of abstraction with directed composition. This class is richer than the class of hierarchical programs assumed above. To be more general, and in particular to allow predicates which are undefined in all modules, it is necessary to disallow certain combinations of recursion and calls to open predicates. Our approach draws on the notion of *stratification* (introduced in [1] to support a safe use of negation), identifying those programs where only negated relations whose meaning is fixed beforehand are allowed. The basic idea is that modules which call open predicates may be allowed in the hierarchy as long as there exists a bound on the number of their occurrences in unfoldings. A syntactic condition is defined in terms of the condition for checking bounded modules.

The following formalizes the call graph of a program in terms of a context free grammar.

Definition 5.10 *[call grammar] : Let P be a module. Let $atoms(P)$ and $open\_atoms(P)$ denote the atoms and, respectively, the open atoms (i.e., any atom whose predicate symbol is in $open(P)$), occurring in P. The* call grammar *of P is the context-free grammar $G_P = \langle N, T, Q, S\rangle$ defined as follows: the set of nonterminals is given by $N = (atoms(P) \setminus open\_atoms(P)) \cup \{S\}$, where S is a distinguished nonterminal that is the start symbol of G_P; the set of terminal symbols is given by $T = open\_atoms(P)$; and the set of productions Q is given by the following:*

- *For each $A \in atoms(P) \setminus open\_atoms(P)$ there is a production*

  $$S \longrightarrow A.$$

- *For each clause 'h :– $b_1, \ldots, b_n$' in P there is a production*

  $$h \longrightarrow b_1 \cdots b_n.$$

- *For each pair of atoms $\langle b, h\rangle \in atoms(P) \times atoms(P)$ such that b occurs in the body of a clause, h is the head of a clause and b unifies with (a renaming of) h there is a production*

  $$b \longrightarrow h.$$

Example 10 *Consider the following program, which computes the transitive closure of a binary relation b:*

$$\begin{array}{l} tc(X, Y) \leftarrow b(X, Y). \\ tc(U, V) \leftarrow b(U, W), tc(W, V). \end{array}$$

Assume that the only open predicate in this program is b. The call grammar for this program is $G = \langle N, T, Q, S\rangle$, where:

$N = \{S, tc(X, Y), tc(U, V), tc(W, V)\}$;

$T = \{b(X, Y), b(U, W)\}$;

and whose productions are given by

$$\begin{array}{l} S \longrightarrow tc(X, Y) \mid tc(U, V) \mid tc(W, V) \\ tc(X, Y) \longrightarrow b(X, Y) \\ tc(U, V) \longrightarrow b(U, W)\ tc(W, V) \\ tc(W, V) \longrightarrow tc(X, Y) \mid tc(U, V) \end{array}$$

The structure of this grammar becomes more obvious if we rename the grammar symbols as follows:

$$\begin{array}{l} tc(X, Y) \mapsto A, tc(U, V) \mapsto B, tc(W, V) \mapsto C \\ b(X, Y) \mapsto \mathsf{a}, b(U, W) \mapsto \mathsf{b} \end{array}$$

The productions of the grammar then become:

$$\begin{array}{l} S \longrightarrow A \mid B \mid C \\ A \longrightarrow \mathsf{a} \\ B \longrightarrow \mathsf{b} C \\ C \longrightarrow A \mid B \end{array}$$

Observe that $L(G) = \left\{ \mathsf{b}^n \mathsf{a} \;\middle|\; n \geq 0 \right\}$ is not finite.

Theorem 5.11 *Let P be a module with call grammar G_P. If the language $L(G_P)$ of G_P is finite, then the number of atoms occurring in the clauses in $\mathcal{F}(P)$ is bounded.*

Proof. (outline)
Given a program P, let the *rank* of a clause c in $\mathcal{F}(P)$ be the smallest number of unfolding steps necessary to obtain c from P. It can be shown that for any program P, for every clause $c \in \mathcal{F}(P)$ there is a string w in $L(G_P)$ such that the number of atoms in the body of c is equal to the length of w: the proof is by induction on the rank of c. Now suppose that $L(G_P)$ is finite. Let N be the length of the longest string in $L(G_P)$, then no clause in $\mathcal{F}(P)$ can have more than N atoms in its body. The theorem follows. □

Note that it is decidable whether the language of an arbitrary context-free grammar is finite [18]. Theorem 5.11 therefore gives a decidable sufficient condition for determining whether, for any given module P, the clauses in $\mathcal{F}(P)$ are bounded. The following example illustrates the application of this approach.

Example 11 *Consider the following program, which generates the list of prime numbers up to N for any given natural number N:*

$$
\begin{array}{l}
primes(N, L) \leftarrow \\
\quad N < 2, L = [\,]. \\
primes(N, L) \leftarrow \\
\quad N \geq 2, intlist(N, L1), primes\_1(L1, [2], L). \\
\\
primes\_1([\,], L0, L1) \leftarrow \\
\quad reverse(L0, L1). \\
primes\_1([H|L], L0, L1) \leftarrow \\
\quad divisible(L0, H), primes\_1(L, L0, L1). \\
primes\_1([H|L], L0, L1) \leftarrow \\
\quad not\_divisible(L0, H), primes\_1(L, [H|L0], L1).
\end{array}
$$

We omit the definitions of intlist/2, divisible/2, not_divisible/2. The idea of this program is to examine a list of numbers, checking each number to see if it is divisible by any of the primes found up to that point—if it is not, it is added to the list of primes found, and the process continues with the remaining numbers. However, because of the way primes are added to the list as they are found, the list is generated "backwards", and has to be reversed at the end.

Now suppose that the only open predicate in this program is reverse/2, which is imported from a library. The corresponding context-free grammar has a finite language, since the only nonterminals that derive a nonempty string are primes and primes_1, each of which derive only the symbol 'reverse(L0, L1)'. It follows from this that unfolding this program does not produce clauses of unbounded size.

Before introducing the class of semi-hierarchical programs we need the following notation:

Definition 5.12 *[leveling, closure] : Let $P = \bigcup_{i=1}^{n} P_i$ be a modular logic program. A* leveling *of P is a partial order $\preceq$ on the modules of P. The* closure *of a module $P_i \in P$ (with respect to a leveling $\preceq$) is the program :*

$$closure_{\preceq}(P_i) = \bigcup_{P_j \preceq P_i} P_j.$$

Definition 5.13 *[semi-hierarchical programs] : Let $P = \bigcup_{i=1}^{n} P_i$ be a modular logic program. We say that P is* semi-hierarchical *if there exists a leveling $\preceq$ of P such that $closure_{\preceq}(P_i)$ is bounded for $i = 1..n$.*

In particular, note that a program consisting of bounded modules is semi-hierarchical, since the empty partial order serves as an appropriate leveling for such a program. The following example considers the public domain tokenizer for Prolog written by Richard O'Keefe.

Example 12 *Consider a program consisting of the following modules:*

P_{tok} *: Defines a tokenizer for Prolog. The open predicates of this module are append, defined in P_{util}, and I/O primitives defined in P_{sys}.*

P_{util} *: Defines a set of user defined utilities, including the append program from Example 9. It contains no open predicates.*

P_{sys} *: Defines a set of system defined I/O primitives. It contains no open predicates.*

We include here part of P_{tok}:

$$
\begin{array}{l}
read\_tokens(TokenList, Dictionary) \leftarrow \\
\quad read\_tokens(32, Dict, ListOfTokens), \\
\quad append(Dict, [\,], Dict), \\
\quad Dictionary = Dict, \\
\quad TokenList = ListOfTokens. \\
read\_tokens([atom(end\_of\_file)], [\,]). \\
\\
read\_tokens(-1, \_, \_) \leftarrow fail. \\
read\_tokens(Ch, Dict, Tokens) \leftarrow \\
\quad Ch =< 32, \\
\quad get0(NextCh), \\
\quad read\_tokens(NextCh, Dict, Tokens). \\
read\_tokens(40, Dict, ['('|Tokens]) \leftarrow \\
\quad get0(NextCh), \\
\quad read\_tokens(NextCh, Dict, Tokens). \\
read\_tokens(41, Dict, [')'|Tokens]) \leftarrow \\
\quad get0(NextCh), \\
\quad read\_tokens(NextCh, Dict, Tokens).
\end{array}
$$

The program $P_{tok} \cup P_{util} \cup P_{sys}$ is hierarchical: P_{tok} is "above" the modules P_{util} and P_{sys}. While the program $P = P_{tok} \cup P_{sys}$ is not hierarchical, it is semi-hierarchical. Hence P can be analyzed without considering the meaning of append.

6 Reusing Analyses

The goal of this work has been to develop a formal technique for the compositional abstract interpretation of modular logic programs. With such an approach, if some modules in a program change during development, it is necessary to reanalyze only those modules that have changed: the abstract semantics computed for the other modules can be reused without any problems, and the new abstract semantics for the program computed simply by composing them with the (new) abstract semantics computed for the modules that have changed. (Contrast this to the work of [10, 24], where it is necessary to reanalyze not only the modules that have changed, but (potentially) also any module that depends on a changed module.) In this section, we illustrate this reuse of abstract semantics with an example.

Example 13 *Consider again the program of Example 1: suppose the module P_{lg} is changed to use a different formulation of the predicates gt and le:*

$$
\begin{array}{l}
gt(s(X), X).\\
gt(s(X), Y) \leftarrow gt(X, Y).\\
gt(s(X), s(Y)) \leftarrow gt(X, Y).\\
\\
le(X, X).\\
le(X, Y) \leftarrow gt(Y, X).
\end{array}
$$

Let the changed module be denoted by P'_{lg}. Its abstract semantics, using the same abstract domain as in the previous examples, is given by
$\mathcal{F}^{\star}(P'_{lg}) =$

$$\{\langle le(x_1, x_2); x_1 \leftrightarrow x_2\rangle, \langle gt(x_1, x_2); x_1 \leftrightarrow x_2\rangle\}.$$

The new abstract semantics for split can now be obtained without reanalysis, by simply composing the (previously computed) abstract semantics of split with the (new) abstract semantics for gt and le:
$\mathcal{F}^{\star}(\mathcal{F}^{\star}(P_{sp}) \sqcup \mathcal{F}^{\star}(P'_{lg})) =$

$$\left\{ \begin{array}{l} \langle split(x_1, x_2, x_3, x_4); x_2 \leftrightarrow x_3 \leftrightarrow x_4\rangle, \\ \langle gt(x_1, x_2); x_1 \leftrightarrow x_2\rangle, \\ \langle le(x_1, x_2); x_1 \leftrightarrow x_2\rangle \end{array} \right\}.$$

It can be seen from this that the change to the definitions of the predicates gt and le leads to a slightly different abstract meaning for the predicate split: whereas in Example 8 it was inferred that each of the second, third and fourth arguments of split was definitely ground, we now infer that these three arguments are either all ground, or are all nonground. On examining the program P'_{lg}, it is apparent that this is, in fact, what should be inferred.

7 More General Composition

The main focus of this paper has been on the compositional analysis of predicate disjoint modules. This choice is motivated by the fact that module based implementations of logic programming languages typically provide this functionality. Moreover from a technical point of view, the assumption that modules are predicate disjoint simplifies somewhat our presentation. For example, we do not need to introduce "import" declarations to the syntax since only predicates which are not defined in a module may be open.

However, it is worth noting that from the analysis point of view there is no real obstacle in providing for compositional analysis of programs which are not predicate disjoint. Moreover, although most implementations do not support such modules, the possibility of spreading the definitions of a predicate in different modules is useful, for example, in distributed deductive databases. This allows different modules to represent different views of the knowledge about a predicate.

To substantiate our claim, we note that the compositional semantics defined in [4] (which is the basis for our framework) is not restricted to predicate disjoint modules. Instead, each module is conceptually accompanied by a declaration of its open predicates. The concrete fixpoint semantics is defined as before, by allowing tautological clauses in Φ_P for each open predicate. Moreover, Proposition 3.4 holds for arbitrary modules while Theorem 4.1 and Proposition 5.8 extend with no difficulty.

The following example illustrates the feasibility of applying compositional analysis to programs which contain modules which are not predicate disjoint.

Example 14 *Consider a program consisting of the following modules:*

P_{int} : *Defining the evaluation of arithmetic expressions over the integers and including definitions for predicates integer/1, plus/3, times/3, and eval/2. The clauses for eval/2 include:*

$$
\begin{array}{l}
eval(A + B, Y) \leftarrow\\
\quad eval(A, A'), eval(B, B'),\\
\quad plus(A', B', Y).\\
eval(A * B, Y) \leftarrow\\
\quad eval(A, A'), eval(B, B'),\\
\quad times(A', B', Y).\\
eval(X, X) \leftarrow integer(X).
\end{array}
$$

P_{real} : *Defining the evaluation of arithmetic expressions over the reals and including definitions for predicates real/1, sinus/2 and eval/2. The clauses for eval/2 include:*

$$
\begin{array}{l}
eval(sin(A), Y) \leftarrow\\
\quad eval(A, A'), sinus(A', Y).\\
eval(X, X) \leftarrow real(X).
\end{array}
$$

The predicate eval/2 is assumed to be partially defined and hence open in both P_{int} and P_{real}. Consider a ground dependency analysis of these modules. For P_{int} we could expect the following result of an analysis:

$\mathcal{F}^\star(P_{int}) =$

$$\left\{ \begin{array}{l} \langle integer(x); x \leftrightarrow \emptyset\rangle, \\ \langle plus(x,y,z); \{x,y\} \leftrightarrow z\rangle, \\ \langle times(x,y,z); \{x,y\} \leftrightarrow z\rangle, \\ \langle eval(x,y); \{x,y\} \leftrightarrow \emptyset\rangle, \\ \langle eval(x,y) \leftarrow eval(x',y'); x \leftrightarrow x', y \leftrightarrow y'\rangle \end{array} \right\}.$$

The last tuple derives from the fact that eval/2 is open and hence has an added tautological clause. Likewise, the anticipated result of an analysis for P_{real} is:

$\mathcal{F}^\star(P_{real}) =$

$$\left\{ \begin{array}{l} \langle real(x); x \leftrightarrow \emptyset\rangle, \\ \langle sinus(x,y); x \leftrightarrow y\rangle, \\ \langle eval(x,y); \{x,y\} \leftrightarrow \emptyset\rangle, \\ \langle eval(x,y) \leftarrow eval(x',y'); x \leftrightarrow x', y \leftrightarrow y'\rangle \end{array} \right\}.$$

Now consider the analysis for $P = P_{int} \cup P_{real}$. There are two possible views:

1. *If eval/2 is assumed closed in P then we should consider unfoldings of $\mathcal{F}^\star(P_{int})$ and $\mathcal{F}^\star(P_{real})$ giving*

$$\left\{ \begin{array}{l} \langle integer(x); x \leftrightarrow \emptyset\rangle, \\ \langle plus(x,y,z); \{x,y\} \leftrightarrow z\rangle, \\ \langle times(x,y,z); \{x,y\} \leftrightarrow z\rangle, \\ \langle real(x); x \leftrightarrow \emptyset\rangle, \\ \langle sinus(x,y); x \leftrightarrow y\rangle, \\ \langle eval(x,y); \{x,y\} \leftrightarrow \emptyset\rangle \end{array} \right\}.$$

2. *On the other hand if eval/2 is assumed open then we should consider unfoldings of $\mathcal{F}^\star(P_{int})$ and $\mathcal{F}^\star(P_{real})$ together with the tautological clause $eval(x,y) \leftarrow eval(x,y)$ giving:*

$$\left\{ \begin{array}{l} \langle integer(x); x \leftrightarrow \emptyset\rangle, \\ \langle plus(x,y,z); \{x,y\} \leftrightarrow z\rangle, \\ \langle times(x,y,z); \{x,y\} \leftrightarrow z\rangle, \\ \langle real(x); x \leftrightarrow \emptyset\rangle, \\ \langle sinus(x,y); x \leftrightarrow y\rangle, \\ \langle eval(x,y); \{x,y\} \leftrightarrow \emptyset\rangle, \\ \langle eval(x,y) \leftarrow eval(x',y'); x \leftrightarrow x', y \leftrightarrow y'\rangle \end{array} \right\}.$$

8 Related Work

Several compositional semantics for logic programs have been proposed in the literature. These include Mancarella *et al.* [21], Gaifmann *et al.* [16] and Bossi *et al.* [4]. In [21] the compositional semantics is provided by composing the T_P functions associated with program modules. Gaifmann *et al.* propose to adopt clauses as semantic objects in order to characterize partial computations (from the head to the body) and to enable different notions of composition. Bossi *et al.* also consider clauses as semantic objects. They propose a bottom-up approach providing a semantics that resembles the non-ground T_P operator of [14]. Logical semantics for modules in logic programs have been proposed by a number of authors [7, 22]. These are typically based on various extensions to Horn logic: for example, Chen's treatment of modules [7] is based on second-order logic, while Miller's [22] uses implication goals in clause bodies. In either case, the semantics appears to be somewhat more complicated than that considered in [4], and we conjecture that a formal treatment of abstract interpretation based on such semantics would require considerably more machinery than that given here.

The problem of program analysis across module boundaries for imperative languages has been considered by a number of researchers: Cooper *et al.* [10] and Tichy *et al.* [24] are concerned primarily with low-level details of maintaining information to allow a compiler to determine whether a change to one program unit necessitates the recompilation of another, separately-compiled, unit, while Santhanam and Odnert [23] consider register allocation across module boundaries. While the motivation for their work is related to ours, the treatment is significantly different in that no attempt is made to give a formal semantic account of the problem or the proposed solutions. These authors have no notion of "composition of abstract semantics" analogous to ours; because of this, if the dataflow characteristics of a module in a program changes, it is necessary to reanalyze other modules that depend on it—in the worst case, this can lead to reanalysis of every module in the program. By contrast, in our approach it is necessary to reanalyze only the modules that have actually changed: the effects of these changes are propagated by composition of abstract semantics.

9 Conclusions

We have described a compositional approach to the abstract interpretation of modular logic programs. In the proposed framework the analysis of a program can be derived by composing the analyses of its constituent modules. For a substantial class of *hierarchical* programs, composition does not entail a sacrifice of precision.

In addition to reducing the conceptual complexity of large programs and enabling the analysis of programs developed by teams, we expect that our framework will prove useful in developing new applications which focus on the analysis of the interaction between modules.

Finally, this paper has focused on the abstraction of a bottom-up compositional semantics. However the approach taken is of general interest. In particular the ideas of considering hierarchical programs and star abstraction are applicable also for the development of top-down frameworks for compositional analysis.

Acknowledgements: The stimulating discussions with Maurizio Gabbrielli and Giorgio Levi and the comments of Gerda Janssens are gratefully acknowledged.

References

[1] K. R. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Morgan Kaufmann, Los Altos, Ca., 1988.

[2] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. Technical Report TR 12/91, Dipartimento di Informatica, Università di Pisa, 1991. To appear in *ACM Transactions on Programming Languages and Systems.*

[3] BIM_Prolog reference manual. B.I.M. B – 3078, Everberg, Belgium.

[4] 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*, pp. 570–580, 1992.

[5] M. Bruynooghe, G. Janssens, B. Demoen, and A. Callebaut. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proc. Fourth IEEE Int'l Symp. on Logic Programming*, pp. 192–204. IEEE Comp. Soc. Press, 1987.

[6] M. Carlsson and J. Widen. *SICStus Prolog Users Manual.* SICS, Sweden, 1988.

[7] W. Chen. A Theory of Modules Based on Second-Order Logic. In *Proc. Fourth IEEE Int'l Symp. on Logic Programming*, pp. 24–33. IEEE Comp. Soc. Press, 1987.

[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, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pp. 331– 345. The MIT Press, Cambridge, Mass., 1991.

[10] K.D. Cooper, K. Kennedy, and L. Torczon. Interprocedural Optimization: Eliminating Unecessary Recompilation. In *Proc. SIGPLAN '86 Symp. on Compiler Construction*, pp. 58–67, 1986.

[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*, pp. 238–252, 1977.

[12] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pp. 269–282, 1979.

[13] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proc. of PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, Berlin, 1992.

[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] H. Gaifman, M. J. Maher, and E. Y. Shapiro. Reactive Behavior Semantics for Concurrent Constraint Logic Programs. In E. Lusk and R. Overbeck, editors, *Proc. North American Conf. on Logic Programming'89*, pp. 553–572. The MIT Press, Cambridge, Mass., 1989.

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

[17] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro. Fully abstract denotational semantics for Concurrent Prolog. In *Proc. Third IEEE Symp. on Logic In Computer Science*, pp. 320–335. IEEE Computer Society Press, 1988.

[18] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[19] 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*, pp. 1649–1665. The MIT Press, Cambridge, Mass., 1988.

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

[21] P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pp. 1006–1023. The MIT Press, Cambridge, Mass., 1988.

[22] D. Miller. A Theory of Modules for Logic Programming. In *Proceedings IEEE Symposium on Logic Programming*, pp. 106–114, 1986.

[23] V. Santhanam and D. Odnert, "Register Allocation across Procedure and Module Boundaries", *Proc. ACM SIGPLAN-90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 28–39.

[24] W.F. Tichy and M.C. Baker. Smart Recompilation. In *Proc. Twelfth ACM Symp. on Principles of Programming Languages*, pp. 236–244. ACM, 1985.