Compositional Analysis of Modular Logic Programs [*]

Michael Codish
Department of Mathematics and Computer Science
Ben-Gurion University, Israel
codish@cs.bgu.ac.il

Saumya Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
debray@cs.arizona.edu

Roberto Giacobazzi
Dipartimento di Informatica
Universita di Pisa
I-56125 Pisa, Italy
giaco@di.unipi.it

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. 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.

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 specifications for a program can change while the program is being constructed, a modular program structure and a corresponding modular analysis can reduce the complexity of addressing such changes 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. We consider a semantics for 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

[*]A preliminary version of this paper appeared in *Proc. 1993 ACM Symposium on Principles of Programming Languages*. This research was supported in part by CEC DGIII EC-Israel collaborative activity, ISC-IL-90-PARFORCE

the entire program. Semantic treatments of modules in logic programs have been given by a number of authors (e.g. see [5, 29]), 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 by Bossi *et al.* [3]. The idea is to treat modules as programs in which undefined predicates are considered to be "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—i.e., iterated unfolding—as that used for describing the meaning of a module, leading to a conceptually simple and elegant treatment.

This semantics is attractive for abstract interpretation purposes, since it can be seen as a generalization of the approach of Falaschi *et al.* [17], which provides the semantic basis for abstract interpretations such as those discussed in [2, 6]. 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 might 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 open atoms. We propose two notions of composition: *symmetric composition* — which composes two program analyses; and *directed composition* — which uses the analysis of one module in the analysis of another. The termination of an analysis based on symmetric composition depends on the inter-module dependencies of the program. A decidable sufficient condition is provided to guarantee that sequences of atoms will be bounded in length. For directed composition, termination depends on the order in which the modules are analyzed.

The applicability of our approach depends both on the topological structure of program modules and on the algebraic structure of the abstract domain at the basis of an analysis. This paper illustrates the constraints underwhich compositional analysis can be obtained with ease. Of course there is a price to be paid for compositionality. One might have to merge modules to eliminate dependencies between modules and one might have to strengthen the abstract domain to enhance its capability to support compositional analysis. Similar properties are required by a domain to support goal-independent analyses as described in [10]. The technical proofs for our results can be found in [21].

2 Preliminaries

In the following we assume familiarity with the standard definitions and notation for logic programs [27], constraint logic programs [26] and abstract interpretation [15]. Throughout, we assume a fixed set of function symbols Σ, a fixed set of predicate symbols Π and a fixed denumerable set of variables **Var**. The non-ground term algebra over Σ and **Var** is denoted **Term**. Herbrand constraints are finite conjunctions of equations between terms in **Term**; the system of such constraints is denoted by $\mathcal{H}$. Constraints are (possibly infinite) disjunctions of Herbrand constraints. We often view a disjunction of Herbrand constraints as a set of Herbrand constraints. We let **false** denote the unsatisfiable constraint (e.g., the empty disjunction) and **true** the always satisfiable constraint (e.g., the empty conjunction). We write $\theta \leq \theta'$ if θ logically implies θ'. $\mathcal{C}$ denotes the set of disjunctions of Herbrand constraints over **Term** modulo logical equivalence. Thus $\mathcal{C}$ is a complete lattice ordered by $\leq$ with bottom element **false** and top element **true**.

The set of atoms constructed from predicate symbols in Π and terms from **Term** is denoted **Atom**. It is well known that "ordinary" logic programs can be characterized as constraint logic programs over the constraint system $\mathcal{H}$ [26], and in context of this paper it is technically convenient to view logic programs in this manner. A *goal* is of the form $\sigma \,[\!]\, \bar{\mathbf{a}}$, where σ is a Herbrand constraint and $\bar{\mathbf{a}}$ is a sequence of atoms. The empty sequence of atoms is denoted by **true**. We let $\langle \mathbf{a_i} \rangle_{\mathbf{i}=1}^{\mathbf{n}}$ denote the sequence $\mathbf{a}_1, \ldots, \mathbf{a_n}$. The concatenation of sequences $\bar{\mathbf{b}}_1$ and $\bar{\mathbf{b}}_2$ is written $\bar{\mathbf{b}}_1 :: \bar{\mathbf{b}}_2$. A (constrained) *Horn clause* is an object of the form $\mathbf{h} \leftarrow \sigma \,[\!]\, \bar{\mathbf{b}}$ where $\mathbf{h}$ is an atom, called the *head*, and $\sigma \parallel \bar{\mathbf{b}}$ is a goal, called the *body*. When $\sigma = \mathbf{true}$ we write $\mathbf{h} \leftarrow \bar{\mathbf{b}}$. A logic program (or module) is a finite set of clauses. The set of clauses constructed from elements of **Atom** is denoted **Clause**. We say that a clause $\mathbf{h} \leftarrow \sigma \parallel \bar{\mathbf{b}}$ is *normalized* if σ is in solved form and $\mathbf{h} \leftarrow \bar{\mathbf{b}}$ is linear, i.e., contains no multiple occurrences of any variable, and flat, i.e., contains no function symbols or constants. $\mathbf{vars}(\mathbf{t})$ denotes the set of variables occurring in a syntactic object $\mathbf{t}$.

A *substitution* ranging in **Sub** 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. Following tradition, the application of a substitution θ to an object $\mathbf{t}$ will be written $\mathbf{t}\theta$ rather than $\theta(\mathbf{t})$. The satisfiability of a Herbrand constraint σ, denoted $\mathcal{H} \models \sigma$, is determined by computing a solution (a set of equations in solved form or a *most general unifier*) of σ. We fix a partial function **mgu** which maps a pair of syntactic objects to an idempotent most general unifier of the objects. For a syntactic object $\mathbf{t}$ and constraint σ, $\sigma(\mathbf{t})$ denotes $\mathbf{t}\theta$ where $\theta = \mathbf{mgu}(\sigma)$. For a set of constraints Θ, $\Theta(\mathbf{t}) = \{\sigma(\mathbf{t}) | \sigma \in \Theta\}$. A *variable renaming* is a (possibly non-idempotent) substitution that is a bijection on **Var**. Two syntactic objects $\mathbf{t}_1$ and $\mathbf{t}_2$ are *equivalent up to renaming*, written $\mathbf{t}_1 \approx \mathbf{t}_2$, if $\mathbf{t}_1\rho = \mathbf{t}_2$ for some variable renaming ρ. The equivalence class of $\mathbf{t}$ under $\approx$ is denoted by $[\mathbf{t}]_\approx$. For a syntactic object $\mathbf{s}$ and a set $\mathbf{I}$ of equivalence classes under $\approx$, we denote by $\langle \mathbf{c}_1, \ldots, \mathbf{c_n} \rangle \ll_\mathbf{s} \mathbf{I}$ that $\mathbf{c}_1, \ldots, \mathbf{c_n}$ are representatives of elements of $\mathbf{I}$ renamed apart from $\mathbf{s}$ and from each other. In the discussion that follows, we will be concerned with sets of clauses modulo the following notion of equivalence between clauses: we say that $(\mathbf{h} \leftarrow \sigma \parallel \bar{\mathbf{b}}) \sim (\mathbf{h}' \leftarrow \sigma' \parallel \bar{\mathbf{b}}')$ iff $\sigma(\mathbf{h} \leftarrow \bar{\mathbf{b}}) \approx \sigma'(\mathbf{h}' \leftarrow \bar{\mathbf{b}}')$ and $\bar{\mathbf{b}}$ is a permutation of $\bar{\mathbf{b}}'$. The intention is identify clauses which differ syntactically yet specify the same implicational relationship between the clause body and its head. For example, a clause $\mathbf{p}([\mathbf{H}|\mathbf{L}]) \leftarrow \mathbf{q}(\mathbf{L})$ can also be written as $\mathbf{p}(\mathbf{X}) \leftarrow \mathbf{X} = [\mathbf{H}|\mathbf{L}] \parallel \mathbf{q}(\mathbf{L})$. For simplicity of exposition, we will abuse notation and assume that a clause represents its equivalence class under $\sim$ and write *Clause* rather than $[\mathbf{Clause}]_\sim$. The powerset of a set $\mathbf{X}$ is denoted by $\wp(\mathbf{X})$. An *interpretation* is any element in $\mathbf{Int} = \wp(\mathbf{Clause})$.

3 Bottom-Up Semantics for Composition

If $\mathbf{P}_1, \ldots, \mathbf{P_n}$ are logic program modules, then $\mathbf{P} = \cup_{\mathbf{i}=1}^{\mathbf{n}} \mathbf{P_i}$ is a *modular logic program*. Such a program is said to be *predicate disjoint* if no predicate is defined in more than one module. In the following, we assume that modular programs are predicate disjoint as in [19], unless otherwise specified. For a logic program (or module) $\mathbf{P}$, $\mathbf{open}(\mathbf{P})$ denotes the set of predicates that occur in the body of a clause in $\mathbf{P}$ but are not defined in $\mathbf{P}$. For any program $\mathbf{P}$, we denote by $\Phi_\mathbf{P}$ the set $\{\ [\mathbf{p}(\bar{\mathbf{x}}) \leftarrow \mathbf{p}(\bar{\mathbf{x}})]_\sim \mid \mathbf{p} \in \mathbf{open}(\mathbf{P})\ \}$.

The modular semantics we assume is an instance of the compositional bottom-up semantics of Bossi *et al.* [3]. We consider two notions of composition: the first, "symmetric composition", is that introduced in [3]. It is applicable to the analysis of modular logic programs for which we can guarantee that the size of clauses will be *bound* in the number of open predicates occurring in their bodies. The second, "directed composition", allows us to analyze a module while "plugging in" analyses for predicates defined in other modules that are lower in the program call graph.

The concrete semantics is formalized in terms of unfolding of clauses. The unfolding operator *unf* specifies the result of unfolding clauses from an interpretation $\mathbf{P}_1$ with clauses from an interpretation $\mathbf{P}_2$ (cf. [16]).

Definition 3.1 [unfolding]
The unfolding operator $\mathbf{unf} : \mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int}$ is defined as

$$\mathbf{unf}(\mathbf{P}_1, \mathbf{P}_2) = \left\{ \mathbf{h} \leftarrow \sigma' \parallel \bar{\mathbf{b}}_1 :: \cdots :: \bar{\mathbf{b}}_\mathbf{n} \;\middle|\; \begin{array}{l} \mathbf{c} = \ \mathbf{h} \leftarrow \sigma \parallel \mathbf{g}_1, \ldots, \mathbf{g_n} \in \mathbf{P}_1, \\ \langle \mathbf{h_i} \leftarrow \sigma_\mathbf{i} \parallel \bar{\mathbf{b}}_\mathbf{i} \rangle_{\mathbf{i}=1}^{\mathbf{n}} \ll_\mathbf{c} \mathbf{P}_2, \\ \sigma' = \sigma \ \wedge \ \bigwedge_{\mathbf{i}=1}^{\mathbf{n}} (\sigma_\mathbf{i} \wedge \{\mathbf{g_i} = \mathbf{h_i}\}), \\ \mathcal{H} \models \sigma' \end{array} \right\}$$

▮

A bottom-up semantics for open logic programs is defined in terms of "iterated unfolding" as follows:

Definition 3.2 [compositional fixpoint semantics [3]]
The semantics of a program $\mathbf{P}$ is given by the function $\mathcal{F} : \mathbf{Int} \to \mathbf{Int}$, defined as $\mathcal{F}(\mathbf{P}) = \mathbf{lfp}(\mathbf{T_P})$, where $\mathbf{T_P} : \mathbf{Int} \to \mathbf{Int}$ is defined as $\mathbf{T_P}(\mathbf{I}) = \mathbf{unf}(\mathbf{P}, \mathbf{I} \cup \Phi_{\mathbf{P}})$. ▮

Note that the above definition is a generalization of the "usual" fixpoint semantics for Horn logic programs. If $\mathbf{open}(\mathbf{P}) = \emptyset$ then the iterations of $\mathbf{T_P}$ starting from the empty interpretation are the same as with the (non-ground) immediate consequence operator of the s-semantics in Falaschi *et al.* [17].

Theorem 3.1 *[concrete composition [3]]*
Let $\mathbf{P}_1$ *and* $\mathbf{P}_2$ *be modules. Then*
(1) symmetric composition : $\mathcal{F}(\mathbf{P}_1 \cup \mathbf{P}_2) = \mathcal{F}(\mathcal{F}(\mathbf{P}_1) \cup \mathcal{F}(\mathbf{P}_2))$*; and*
(2) directed composition: $\mathcal{F}(\mathbf{P}_1 \cup \mathbf{P}_2) = \mathcal{F}(\mathbf{P}_1 \cup \mathcal{F}(\mathbf{P}_2))$.

4 Abstract Semantics and Composition

We assume the standard framework of abstract interpretation, as defined by Cousot and Cousot [15], in terms of Galois insertions. We let $(\mathbf{AInt}, \sqcup, \sqcap, \sqsubseteq)$ denote a complete lattice of *abstract interpretations*, where each abstract interpretation describes a set of clauses, and $(\mathbf{Int}, \alpha, \mathbf{AInt}, \gamma)$ is a Galois insertion. Note that programs and interpretations have the same structure — sets of clauses. Hence, the abstract semantics of a program $\mathbf{P}$ is $\mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}))$, where $\mathcal{F}^{\mathcal{A}} : \mathbf{AInt} \to \mathbf{AInt}$ is the semantic function. Abstract 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 $(\mathbf{Int}, \alpha, \mathbf{AInt}, \gamma)$ *be a Galois insertion and let* $\mathcal{F}^{\mathcal{A}} : \mathbf{AInt} \to \mathbf{AInt}$ *be a monotonic and safe approximation of* $\mathcal{F}$*, i.e. for every* $\mathbf{I} \in \mathbf{Int}$*,* $\alpha(\mathcal{F}(\mathbf{I})) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{I}))$*. Then, for any program modules* $\mathbf{P}_1, \mathbf{P}_2 \in \mathbf{Int}$*,*

1. $\alpha(\mathcal{F}(\mathbf{P}_1 \cup \mathbf{P}_2)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_1)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_2)))$. *[symmetric composition]*
2. $\alpha(\mathcal{F}(\mathbf{P}_1 \cup \mathbf{P}_2)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_1) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_2)))$. *[directed composition]*

Proof

$$\begin{array}{llcll}
(1) & \alpha(\mathcal{F}(\mathbf{P}_1 \cup \mathbf{P}_2)) & = & \alpha(\mathcal{F}(\mathcal{F}(\mathbf{P}_1) \cup \mathcal{F}(\mathbf{P}_2))) & [\ \text{Theorem 3.1}\ ] \\
 & & \sqsubseteq & \mathcal{F}^{\mathcal{A}}(\alpha(\mathcal{F}(\mathbf{P}_1) \cup \mathcal{F}(\mathbf{P}_2))) & [\ \text{safety}\ ] \\
 & & = & \mathcal{F}^{\mathcal{A}}(\alpha(\mathcal{F}(\mathbf{P}_1)) \sqcup \alpha(\mathcal{F}(\mathbf{P}_2))) & [\ \alpha \text{ is additive}\ ] \\
 & & \sqsubseteq & \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_1)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_2))) & [\ \text{safety and monotonicity}\ ]
\end{array}$$

The proof of (2) is similar. □

We now illustrate how a safe semantic function $\mathcal{F}^{\mathcal{A}} : \mathbf{AInt} \to \mathbf{AInt}$ can be induced from a domain $\mathcal{A}$ of abstract constraints (or substitutions). A general definition of abstract constraint system for abstract interpretation of possibly non Herbrand constraints can be found in [22].

Definition 4.1 [abstract constraints]
A domain of abstract constraints is a complete lattice $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ such that $(\mathcal{C}, \alpha_{\mathcal{A}}, \mathcal{A}, \gamma_{\mathcal{A}})$ is a Galois insertion and for any syntactic object $\mathbf{s}$ and sets of constraints $\Theta_1, \Theta_2 \in \mathcal{C}$, $\Theta_1(\mathbf{s}) \approx \Theta_2(\mathbf{s})$ implies $\gamma_{\mathcal{A}}\alpha_{\mathcal{A}}\Theta_1(\mathbf{s}) \approx \gamma_{\mathcal{A}}\alpha_{\mathcal{A}}\Theta_2(\mathbf{s})$. ▮

The condition of Definition 4.1 requires that $\gamma_{\mathcal{A}} \circ \alpha_{\mathcal{A}}$ preserves renamings. Similar conditions on abstract atoms and on abstract substitutions are assumed in [2, 6].

Example 4.1 The domain $\mathbf{Pos}$ was first proposed by Marriott and Søndergaard as a domain of abstract substitutions for groundness analysis [1]. The domain is formalized as a Galois insertion denoted $(\wp(\mathbf{Sub}), \alpha_{\mathbf{Pos}}, \mathbf{Pos}, \gamma_{\mathbf{Pos}})$ and consists of equivalence classes of positive propositional formulae, ordered by implication. The generalization of this domain to Herbrand constraints is straightforward. A truth assignment ξ satisfies a propositional formula $\mathbf{f}$, written $\xi \models \mathbf{f}$, if $\xi(\mathbf{f})$ is a tautology. For groundness analysis, a constraint $\theta \in \mathcal{C}$ is associated with a corresponding truth assignment $\mathbf{assign}_\theta$ that maps a variable $\mathbf{x}$ to *true* if and only if $\theta(\mathbf{x})$ is ground, and which is defined as $\lambda\mathbf{x}.vars(\theta(\mathbf{x})) = \emptyset$. The functions $\gamma_{\mathbf{Pos}} : \mathbf{Pos} \to \mathcal{C}$ and $\alpha_{\mathbf{Pos}} : \mathcal{C} \to \mathbf{Pos}$ are defined as follows (cf. [14]) $\gamma_{\mathbf{Pos}}(\mathbf{f}) = \left\{ \theta \,\middle|\, (\forall\theta') : \theta' \leq \theta \Rightarrow \mathbf{assign}_{\theta'} \models \mathbf{f} \right\}$ and $\alpha_{\mathbf{Pos}}(\Theta) = \wedge \left\{ \kappa \,\middle|\, \Theta \leq \gamma_{\mathbf{Pos}}(\kappa) \right\}$. □

Abstract clauses are similar to normalized (concrete) clauses except that an abstract constraint occurs instead of a concrete constraint.

Definition 4.2 [abstract clauses]
Let $\mathcal{A}$ be a domain of abstract constraints. An abstract clause over $\mathcal{A}$ is an object of the form $\mathbf{h} \leftarrow \kappa \parallel \bar{\mathbf{b}}$ such that $\kappa \in \mathcal{A}$ and $\mathbf{h} \leftarrow \mathbf{true} \parallel \bar{\mathbf{b}}$ is normalized. The set of abstract clauses over $\mathcal{A}$ is denoted $\mathbf{Clause}^{\mathcal{A}}$. ▮

The meaning of a set of abstract clauses over $\mathcal{A}$ is defined by a concretization mapping $\gamma : \wp(\mathbf{Clause}^{\mathcal{A}}) \to \wp(\mathbf{Clause})$ such that for any $\mathbf{I^a} \subseteq \mathbf{Clause}^{\mathcal{A}}$:

$$\gamma(\mathbf{I^a}) = \left\{ [\mathbf{h} \leftarrow \sigma \parallel \bar{\mathbf{b}}]_\sim \,\middle|\, \begin{array}{l} \mathbf{h} \leftarrow \kappa \parallel \bar{\mathbf{b}} \in \mathbf{I^a} \\ \sigma \in \gamma_{\mathcal{A}}(\kappa) \end{array} \right\}$$

A domain of abstract interpretations is induced from $(\mathcal{C}, \alpha_{\mathcal{A}}, \mathcal{A}, \gamma_{\mathcal{A}})$ by considering the equivalence relation $\mathbf{I}^{\mathbf{a}}_1 \cong \mathbf{I}^{\mathbf{a}}_2$ iff $\gamma(\mathbf{I}^{\mathbf{a}}_1) = \gamma(\mathbf{I}^{\mathbf{a}}_2)$, induced by γ on $\wp(\mathbf{Clause}^{\mathcal{A}})$. The intuition is that if $\{\mathbf{h} \leftarrow \kappa \parallel \bar{\mathbf{b}}\} \cong \{\mathbf{h} \leftarrow \kappa' \parallel \bar{\mathbf{b}}\}$ then κ and κ' describe the same set of concrete constraints, when restricted to $\mathbf{vars}(\mathbf{h} \leftarrow \bar{\mathbf{b}})$. Notice that the equivalence relation $\cong$ also provides variable hiding and equivalence up to renaming.

Definition 4.3 [abstract interpretations]
Let $\mathcal{A}$ be a domain of abstract constraints and $\mathbf{AInt}_{\mathcal{A}} = \wp(\mathbf{Clause}^{\mathcal{A}})/_{\cong}$. We lift γ to $\mathbf{AInt}_{\mathcal{A}} \to \mathbf{Int}$ in the standard way. Hence $\alpha : \mathbf{Int} \to \mathbf{AInt}_{\mathcal{A}}$ is defined as

$$\alpha(\mathbf{I}) = \left[\left\{ \mathbf{h} \leftarrow \alpha_{\mathcal{A}}(\sigma) \parallel \bar{\mathbf{b}} \,\middle|\, \begin{array}{l} \mathbf{c} \in \mathbf{I},\ \mathbf{h} \leftarrow \sigma \parallel \bar{\mathbf{b}} \sim \mathbf{c}, \\ \mathbf{h} \leftarrow \sigma \parallel \bar{\mathbf{b}} \text{ is normalized} \end{array} \right\} \right]_{\cong}$$

▮

In the following we will omit equivalence classes modulo $\cong$ when their use can be easily deduced from the context.

The running example considered throughout the paper is the modular logic program specifying the quick-sort relation depicted in Figure 1. The program consists of five modules and is illustrated in constraint form.

Example 4.2 Let $\mathcal{A} = \mathbf{Pos}$ and consider the module $\mathbf{P_{lg}}$ from Figure 1.

$$\alpha(\mathbf{P_{lg}}) = \left\{ \begin{array}{l} \mathbf{gt(x,y)} \leftarrow (\mathbf{x} \leftrightarrow \mathbf{x'}) \wedge \mathbf{y} \parallel \mathbf{num(x')}, \\ \mathbf{gt(x,y)} \leftarrow \mathbf{x} \leftrightarrow \mathbf{x'} \ \ \wedge \ \ \mathbf{y} \leftrightarrow \mathbf{y'} \parallel \mathbf{gt(x',y')}, \\ \mathbf{le(x,y)} \leftarrow \mathbf{x} \wedge (\mathbf{y} \leftrightarrow \mathbf{y'}) \parallel \mathbf{num(y')}, \\ \mathbf{le(x,y)} \leftarrow \mathbf{x} \leftrightarrow \mathbf{x'} \ \ \wedge \ \ \mathbf{y} \leftrightarrow \mathbf{y'} \parallel \mathbf{le(x',y')} \end{array} \right\}.$$

□

$$
\begin{array}{ll}
\mathbf{P_{qs}}: & \mathbf{qsort(X,Y) \leftarrow X = [\,] \ \ \wedge \ \ Y = [\,] \ [\!]\ true.} \\
& \mathbf{qsort(X,Y) \leftarrow X = [X_1|Xs] \ \ \wedge \ \ Ls' = [X_1|Bs] \ [\!]} \\
& \qquad \mathbf{split(X_1, Xs, L_1, L_2), \ \ qsort(L_1, Ls),} \\
& \qquad \mathbf{qsort(L_2, Bs), \ \ append(Ls, Ls', Y).} \\
\\
\mathbf{P_{app}}: & \mathbf{append(X,Y,Z) \leftarrow X = [\,] \ \ \wedge \ \ Y = Z \ [\!]\ true.} \\
& \mathbf{append(X,Y,Z) \leftarrow X = [X_1|Xs] \ \ \wedge \ \ Z = [X_1|Zs]) \ [\!]\ append(Xs, Y, Zs).} \\
\\
\mathbf{P_{sp}}: & \mathbf{split(X_1,X_2,X_3,X_4) \leftarrow X_2 = [\,] \ \ \wedge \ \ X_3 = [\,] \ \ \wedge \ \ X_4 = [\,] \ [\!]\ true.} \\
& \mathbf{split(X_1,X_2,X_3,X_4) \leftarrow X_2 = [Y|L] \ \ \wedge \ \ X_3 = [Y|L_1] \ \ \wedge \ \ X_4 = L_2 \ [\!]} \\
& \qquad \mathbf{gt(X_1, Y), \ \ split(X_1, L, L_1, L_2).} \\
& \mathbf{split(X_1,X_2,X_3,X_4) \leftarrow X_2 = [Y|L] \ \ \wedge \ \ X_3 = L_1 \ \ \wedge \ \ X_4 = [Y|L_2] \ [\!]} \\
& \qquad \mathbf{le(X_1, Y), \ \ split(X_1, L, L_1, L_2).}
\end{array}
$$

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

Figure 1: Modular quick-sort program.

Proposition 4.2
If $\mathcal{A}$ is a domain of abstract constraints and $\mathbf{AInt}_{\mathcal{A}}$ the induced domain of abstract interpretations, then $(\mathbf{AInt}_{\mathcal{A}}, \sqsubseteq)$ is a complete lattice where $\mathbf{I}^{\mathbf{a}}_1 \sqsubseteq \mathbf{I}^{\mathbf{a}}_2$ iff $\gamma(\mathbf{I}^{\mathbf{a}}_1) \subseteq \gamma(\mathbf{I}^{\mathbf{a}}_2)$ and $(\mathbf{Int}, \alpha, \mathbf{AInt}_{\mathcal{A}}, \gamma)$ is a Galois insertion.

We introduce an abstract unfolding operator which is defined in terms of an (monotonic, associative and commutative) operator $\otimes$ on abstract constraints which is required to approximate conjunction on concrete constraints. A domain of abstract constraints together with such an operator constitutes an *abstract constraint system*. The monotonicity requirement for $\otimes$ is natural from the perspective of abstract interpretation. The requirements for associativity and commutativity are stronger, and while they are motivated by a desire to approximate an associative and commutative concrete operation, namely, conjunction of constraints, it is possible that they may not be satisfied by all abstract domains. In this case, there is a tradeoff: one can either strengthen the domain to support compositionality, or one can manually construct larger modules so as to reduce the number of points where these requirements do not hold. In any case, there are relevant examples of domains for logic program analysis that fit in our framework, e.g. $\otimes$ as ACI unification of constraints for types and set-sharing analysis [11]. Termination is addressed in the standard way, i.e., by requiring that the domain of abstract interpretations $AInt_{\mathcal{A}}$ obtained from a set of abstract constraints $\mathcal{A}$ satisfies a given finiteness property, or via the use of widening operators, as discussed by Cousot and Cousot [15].

Definition 4.4 [abstract constraint system]
An abstract constraint system $\mathcal{A}$ is a $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ together with a monotonic, associative and commutative operator $\otimes : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ such that for every $\mathbf{c}_1, \mathbf{c}_2 \in \mathcal{C}$, $\alpha_{\mathcal{A}}(\mathbf{c}_1 \wedge \mathbf{c}_2) \ \sqsubseteq_{\mathcal{A}} \ \alpha_{\mathcal{A}}(\mathbf{c}_1) \otimes \alpha_{\mathcal{A}}(\mathbf{c}_2)$. We say that $\mathcal{A}$ is finitary if for any linear clause $\mathbf{h} \leftarrow \tilde{\mathbf{b}}$, the set $\{[\mathbf{h} \leftarrow \kappa \ [\!] \ \tilde{\mathbf{b}}]_{\cong} \mid \kappa \in \mathcal{A}\}$ does not contain infinite chains. ▮

Definition 4.5 [abstract unfolding]
Let $\mathcal{A}$ be an abstract constraint system. The corresponding abstract unfolding operator $\mathbf{unf}^{\mathcal{A}}$:

$\mathbf{AInt}_{\mathcal{A}} \times \mathbf{AInt}_{\mathcal{A}} \to \mathbf{AInt}_{\mathcal{A}}$ is defined as:

$$\mathbf{unf}^{\mathcal{A}}(\mathbf{P}^{\mathbf{a}}_1, \mathbf{P}^{\mathbf{a}}_2) = \left\{ \mathbf{h} \leftarrow \kappa' \,[\!]\, \bar{\mathbf{b}}_1 :: \cdots :: \bar{\mathbf{b}}_{\mathbf{n}} \;\middle|\; \begin{array}{l} \mathbf{c} = \mathbf{h} \leftarrow \kappa \,[\!]\, \mathbf{g}_1, \ldots, \mathbf{g}_{\mathbf{n}} \in \mathbf{P}^{\mathbf{a}}_1, \\ \langle \mathbf{h}_{\mathbf{i}} \leftarrow \kappa_{\mathbf{i}} \,[\!]\, \bar{\mathbf{b}}_{\mathbf{i}} \rangle_{\mathbf{i}=1}^{\mathbf{n}} \ll_{\mathbf{c}} \mathbf{P}^{\mathbf{a}}_2 \\ \kappa' = \kappa \otimes \bigotimes_{\mathbf{i}=1}^{\mathbf{n}} (\kappa_{\mathbf{i}} \otimes \alpha_{\mathcal{A}}(\mathbf{g}_{\mathbf{i}} = \mathbf{h}_{\mathbf{i}})) \end{array} \right\}$$

▮

The following lemma states that abstract unfolding satisfies the basic properties of concrete unfoldings (for generic properties of concrete unfoldings see [16]).

Lemma 4.3 *[associativity, continuity and additivity]*
For any abstract constraint system $\mathcal{A}$, $\mathbf{unf}^{\mathcal{A}}$ is associative, continuous in the second argument and additive in the first argument.

In the following we assume the obvious extensions of the notions of modular logic programs, open predicates, *etc.* for the abstract case. For an abstract program $\mathbf{P}^{\mathbf{a}}$, $\Phi_{\mathbf{P}^{\mathbf{a}}}$ is the natural extension of $\Phi_{\mathbf{P}}$. The abstract fixpoint semantics is defined in terms of abstract unfolding as follows.

Definition 4.6 [abstract fixpoint semantics]
Let $\mathbf{unf}^{\mathcal{A}}$ be the abstract unfolding operator for an abstract constraint system $\mathcal{A}$. Define $\mathcal{F}^{\mathcal{A}}$: $\mathbf{AInt}_{\mathcal{A}} \to \mathbf{AInt}_{\mathcal{A}}$ as $\mathcal{F}^{\mathcal{A}}(\mathbf{P}^{\mathbf{a}}) = \mathbf{lfp}(\mathbf{T}^{\mathcal{A}}_{\mathbf{P}^{\mathbf{a}}})$ where $\mathbf{T}^{\mathcal{A}}_{\mathbf{P}^{\mathbf{a}}} : \mathbf{AInt}_{\mathcal{A}} \to \mathbf{AInt}_{\mathcal{A}}$ is defined by $\mathbf{T}^{\mathcal{A}}_{\mathbf{P}^{\mathbf{a}}}(\mathbf{I}^{\mathbf{a}}) = \mathbf{unf}^{\mathcal{A}}(\mathbf{P}^{\mathbf{a}}, \mathbf{I}^{\mathbf{a}} \cup \Phi_{\mathbf{P}^{\mathbf{a}}})$.
▮

Proposition 4.4 *[correctness]*
Let $\mathcal{A}$ be an abstract constraint system. Then, $\mathbf{unf}^{\mathcal{A}}$ and $\mathcal{F}^{\mathcal{A}}$ are safe in the induced domain of abstract interpretations $(\mathbf{Int}, \alpha, \mathbf{AInt}_{\mathcal{A}}, \gamma)$. Namely, for every $\mathbf{I}_1, \mathbf{I}_2 \in \mathbf{Int}$: $\alpha(\mathbf{unf}(\mathbf{I}_1, \mathbf{I}_2)) \sqsubseteq \mathbf{unf}^{\mathcal{A}}(\alpha(\mathbf{I}_1), \alpha(\mathbf{I}_2))$ and $\alpha(\mathcal{F}(\mathbf{I}_1)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{I}_1))$.

5 Compositional Analysis

The following illustrates an application for compositional groundness analysis. It is straightforward to prove that **Pos** is finitary.

Example 5.1 Let $\mathcal{A} = \mathbf{Pos}$ and consider the modules $\mathbf{P}_{\mathbf{lg}}$ and $\mathbf{P}_{\mathbf{num}}$ from Figure 1.

$$\begin{array}{lcl} \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_{\mathbf{lg}})) & = & \left\{ \begin{array}{l} \mathbf{gt}(\mathbf{X}, \mathbf{Y}) \leftarrow (\mathbf{X} \leftrightarrow \mathbf{X}') \;\wedge\; \mathbf{Y} \,[\!]\, \mathbf{num}(\mathbf{X}') \\ \mathbf{le}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{X} \;\wedge\; (\mathbf{Y} \leftrightarrow \mathbf{Y}') \,[\!]\, \mathbf{num}(\mathbf{Y}') \end{array} \right\}; \\ \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_{\mathbf{num}})) & = & \{ \;\mathbf{num}(\mathbf{X}) \leftarrow \mathbf{X} \,[\!]\, \mathbf{true} \;\}. \end{array}$$

Abstract (symmetric) composition gives:

$$\mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_{\mathbf{lg}})) \cup \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}_{\mathbf{num}}))) = \left\{ \begin{array}{l} \mathbf{gt}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{X} \;\wedge\; \mathbf{Y} \,[\!]\, \mathbf{true} \\ \mathbf{le}(\mathbf{X}, \mathbf{Y}) \leftarrow \mathbf{X} \;\wedge\; \mathbf{Y} \,[\!]\, \mathbf{true} \\ \mathbf{num}(\mathbf{X}) \leftarrow \mathbf{X} \,[\!]\, \mathbf{true} \end{array} \right\}.$$

□

In the general case, an interesting technical problem arises as abstract unfolding may introduce arbitrarily large clauses so that analyses can no longer be guaranteed to terminate, even if the domain of abstract constraints is finitary. This necessitates a second (and orthogonal) abstraction to deal with unbounded clause bodies in the abstract semantics. Observe, for example, that the abstract unfoldings of the module $\mathbf{P}_{\mathbf{sp}}$ in Figure 1 introduce arbitrarily long abstract clauses.

This problem can be addressed in several ways. In [7], a notion of *star abstraction* adopted from [9] is applied to limit the length of clause bodies using a domain termed **Dep** for ground dependency analysis. The basic idea is to collapse the occurrences of calls to a predicate $\mathbf{p}$ in a clause body to one "canonical" call $\mathbf{p}_\star$. While this approach indeed restricts the size of the clauses which can be generated in the analysis, it also can result in a loss of precision in the unfolding process. Alternatively, in [18] it is shown that for finite domains a characterization of the compositional semantics can be obtained after a finite number of unfoldings without reaching a fixed point. However, in this approach the number of iterations and the size of the clauses that must be considered can be prohibitive. In this paper, we consider two cases for which the size of the abstract clauses generated during the analysis is bound. For the general case, our techniques can be combined with an additional layer of star abstraction to guarantee termination.

Bounded Symmetric Compositional Analysis.

We characterize a class of *bounded* program modules for which unfolding does not create clauses of unbound length. Consequently, if a program consists of bounded modules then symmetric composition can be applied in program 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: it suffices to avoid loops containing literals for open predicates. A convenient way to express this criterion is by way of a context free grammar.

Definition 5.1 [call grammar]
Let $\mathbf{P}$ be a module. Let $\mathbf{atoms(P)}$ and $\mathbf{open\_atoms(P)}$ denote the atoms and, respectively, the call to atoms in $\mathbf{open(P)}$, occurring in $\mathbf{P}$. The *call grammar* of $\mathbf{P}$ is the context-free grammar $\mathbf{G_P} = \langle \mathbf{N}, \mathbf{T}, \mathbf{Q}, \mathbf{S} \rangle$ defined as follows: the nonterminals are given by $\mathbf{N} = (\mathbf{atoms(P)} \setminus \mathbf{open\_atoms(P)}) \cup \{\mathbf{S}\}$, where $\mathbf{S}$ is a distinguished nonterminal — the start symbol of $\mathbf{G_P}$; the terminal symbols are given by $\mathbf{T} = \mathbf{open\_atoms(P)}$; and the productions $\mathbf{Q}$ are given by:

- For each $\mathbf{A} \in \mathbf{atoms(P)} \setminus \mathbf{open\_atoms(P)}$ there is a production $\mathbf{S} \longrightarrow \mathbf{A}$.
- For each $\mathbf{h} \leftarrow \sigma \,[\!]\, \mathbf{b}_1, \ldots, \mathbf{b_n} \in \mathbf{P}$ there is a production $\sigma(\mathbf{h}) \longrightarrow \sigma(\mathbf{b}_1) \cdots \sigma(\mathbf{b_n})$.
- For each pair of atoms $\mathbf{b} \in \mathbf{atoms(P)}$ from the body of a clause and $\mathbf{h} \in \mathbf{atoms(P)}$ from the head of a clause, such that $\mathbf{b}$ unifies with (a renaming of) $\mathbf{h}$ there is a production $\mathbf{b} \longrightarrow \mathbf{h}$.

▮

Example 5.2 Consider the following program, which computes the transitive closure of a binary relation $\mathbf{b}$ which is an open predicate in the definition:

$$tc(\mathbf{X},\mathbf{Y}) \leftarrow \mathbf{b}(\mathbf{X},\mathbf{Y}).$$
$$tc(\mathbf{U},\mathbf{V}) \leftarrow \mathbf{b}(\mathbf{U},\mathbf{W}), tc(\mathbf{W},\mathbf{V}).$$

The productions for the call grammar $\mathbf{G_P}$ of this program are given by

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

Observe that $\mathbf{L(G_P)}$ is not finite. □

Theorem 5.1
Let $\mathbf{P}$ be a module with call grammar $\mathbf{G_P}$. If the language $\mathbf{L(G_P)}$ of $\mathbf{G_P}$ is finite, then the number of atoms occurring in the clauses in $\mathcal{F}(\mathbf{P})$ is bounded.

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

Note that it is decidable whether the language of an arbitrary context-free grammar is finite [25]. Theorem 5.1 therefore gives a decidable sufficient condition for determining whether, for any given module $\mathbf{P}$, the clauses in $\mathcal{F}(\mathbf{P})$ are bounded. The following example illustrates the application of this approach.

Example 5.3 Consider the following program, which generates the list of prime numbers up to $\mathbf{N}$ for any given natural number $\mathbf{N}$, and assume that the **reverse** predicate is open and imported from a library:

```
primes(N, L) ←
    N < 2, L = [ ].
primes(N, L) ←
    N ≥ 2, intlist(N, L1),
    primes_1(L1, [2], L).

primes_1([ ], L0, L1) ←
    reverse(L0, L1).
primes_1([H|L], L0, L1) ←
    divisible(L0, H), primes_1(L, L0, L1).
primes_1([H|L], L0, L1) ←
    not_divisible(L0, H), primes_1(L, [H|L0], L1).
```

The program examines 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, the list is generated "backwards", and has to be reversed at the end. The corresponding context-free grammar has a finite language, and so it follows that unfolding this program does not produce clauses of unbounded size. □

Directed Compositional Analysis.

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. 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. In this case it is always possible to obtain a compositional evaluation for program analysis that does not involve clause structures of unbounded size. The abstract meanings of modules which are lower in the hierarchy can be used when evaluating the abstract meanings of the higher modules. In this way, if the hierarchy is "closed," i.e., for every module, each of its open predicates is defined in a module that is lower in the hierarchy, then unfoldings will always produce unit clauses. This corresponds to "plugging in" the analyses of the lower modules into the analyses of the higher modules, and can be useful in the bottom-up approach to program design. The following example illustrates a hierarchical groundness analysis, taking $\mathcal{A} = \mathbf{Pos}$.

Example 5.4 Consider the quick-sort program from Figure 1. The analysis for $\mathbf{P_{num}}$ and $\mathbf{P_{lg}}$ may be performed using a symmetrical composition as there is no problem of unbounded bodies. A hierarchical analysis starting with the lower modules will then consider $\mathbf{P_{app}}$:

$$\mathcal{F}(\mathbf{P_{app}}) = \{\ \mathbf{append}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) \leftarrow (\mathbf{x}_1 \wedge \mathbf{x}_2) \leftrightarrow \mathbf{x}_3 \ [\!]\ \mathbf{true}.\ \}$$

Next the analysis of $\mathbf{P_{lg}} \cup \mathbf{P_{num}}$ is plugged into $\mathbf{P_{sp}}$. Directed composition gives:

$$\mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P_{sp}}) \cup \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P_{lg}} \cup \mathbf{P_{num}}))) = \left\{ \begin{array}{l} \mathbf{split}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) \leftarrow \mathbf{x}_2 \wedge \mathbf{x}_3 \wedge \mathbf{x}_4 \ [\!]\ \mathbf{true} \\ \mathbf{gt}(\mathbf{x}, \mathbf{y}) \leftarrow \mathbf{x} \wedge \mathbf{y} \ [\!]\ \mathbf{true}, \\ \mathbf{le}(\mathbf{x}, \mathbf{y}) \leftarrow \mathbf{x} \wedge \mathbf{y} \ [\!]\ \mathbf{true}, \\ \mathbf{num}(\mathbf{x}) \leftarrow \mathbf{x} \ [\!]\ \mathbf{true}, \end{array} \right\}$$

Two additional applications of directed composition plug the analyses for **split** and **append** into the analysis of **qsort** giving as expected:

$$\{\mathbf{qsort}(\mathbf{x}_1, \mathbf{x}_2) \leftarrow \mathbf{x}_1 \leftrightarrow \mathbf{x}_2 \mathbin{[\![} \mathbf{true}\}.$$

□

A richer class of *semi-hierarchical* programs is defined to allow predicates which are undefined in all modules. To this end it is necessary to disallow certain combinations of recursion and calls to open predicates. 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. This can be formalized in terms of the condition for checking bounded modules.

Definition 5.2 [leveling, closure]
Let $\mathbf{P} = \cup_{\mathbf{i}=1}^{\mathbf{n}} \mathbf{P_i}$ be a modular logic program. A *leveling* of $\mathbf{P}$ is a partial order $\preceq$ on the modules of $\mathbf{P}$. The *closure* of a module $\mathbf{P_i} \in \mathbf{P}$ (with respect to a leveling $\preceq$) is the program :

$$\mathbf{closure}_{\preceq}(\mathbf{P_i}) = \bigcup_{\mathbf{P_j} \preceq \mathbf{P_i}} \mathbf{P_j}.$$

We say that $\mathbf{P}$ is *semi-hierarchical* if there exists a leveling $\preceq$ of $\mathbf{P}$ such that $\mathbf{closure}_{\preceq}(\mathbf{P_i})$ is bounded for $\mathbf{i} = 1..\mathbf{n}$. ▮

In particular, note that a program consisting of bounded modules is semi-hierarchical. The following example considers the public domain tokenizer for Prolog due to O'Keefe.

Example 5.5 Consider a program consisting of the following modules:

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

$\mathbf{P}_{util}$: Defines a set of user defined utilities, including the append program from Example 5.4. It contains no open predicates.

$\mathbf{P}_{sys}$: Defines a set of system defined I/O primitives. It contains no open predicates.

We include here part of $\mathbf{P}_{tok}$ (the predicate *read_tokens* is defined by a great many clauses having essentially the same structure, we include only a few of these to illustrate the point):

```
read_tokens(TokenList, Dictionary) ←
    read_tokens(32, Dict, ListOfTokens),
    append(Dict, [ ], Dict),
    Dictionary = Dict,
    TokenList = ListOfTokens.
read_tokens([atom(end_of_file)], [ ]).
read_tokens(41, Dict, [')'|Tokens]) ←
    get0(NextCh),
    read_tokens(NextCh, Dict, Tokens).
```

```
read_tokens(−1, _, _) ← fail.
read_tokens(Ch, Dict, Tokens) ←
    Ch =< 32,
    get0(NextCh),
    read_tokens(NextCh, Dict, Tokens).
read_tokens(40, Dict, ['('|Tokens]) ←
    get0(NextCh),
    read_tokens(NextCh, Dict, Tokens).
```

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

6 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 our presentation somewhat. 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 there is no real obstacle in providing a compositional analysis for programs which are not predicate disjoint, and in fact the semantic basis defined in [3] is not restricted to predicate disjoint 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. The main result in Theorem 4.1 extends easily for this case.

We illustrate the application of a compositional approach to provide a goal-independent analysis of the calls which arise in the execution of a given initial goal for a (closed) program **P**. The analysis is based on a variant of the well known Magic Set transformation [30] which characterizes the calls for a program **P** and initial goal **G** in terms of a transformed program $\mathbf{magic}(\mathbf{P};\mathbf{G})$. Following [8] we specify the transformation using a modular approach distinguishing between clauses which depend on the goal **G** and those which do not.

Let **P** be a (closed) program and $\mathbf{G} = \sigma \ [\![\ \mathbf{g}_1, \ldots \mathbf{g_k}$ an initial goal. The corresponding magic program is $\mathbf{magic}(\mathbf{P};\mathbf{G}) = \mathbf{P} \cup \mathbf{P}_{\mathcal{M}} \cup \mathbf{G}_{\mathcal{M}}$ where:

$$\mathbf{G}_{\mathcal{M}} = \left\{ \ \mathbf{call\_g_i} \leftarrow \sigma \ [\![\ \mathbf{g}_1, \ldots, \mathbf{g_{i-1}} \ \middle| \ 1 \leq \mathbf{i} \leq \mathbf{k} \ \right\}; \text{ and}$$

$$\mathbf{P}_{\mathcal{M}} = \left\{ \ \mathbf{call\_b_i} \leftarrow \sigma \ [\![\ \mathbf{call_h}, \mathbf{b}_1, \ldots, \mathbf{b_{i-1}} \ \middle| \ \begin{array}{l} \mathbf{h} \leftarrow \sigma \ [\![\ \mathbf{b}_1, \ldots, \mathbf{b_n} \in \mathbf{P}, \\ 1 \leq \mathbf{i} \leq \mathbf{n} \end{array} \right\}.$$

The program $\mathbf{magic}(\mathbf{P};\mathbf{G})$ has the property that if **p** is a call in a computation of **G** with **P** (assuming a left-to-right computation rule), then **call_p** is implied by $\mathbf{magic}(\mathbf{P};\mathbf{G})$. The magic program consists of three modules. The first is **P** itself; the second, $\mathbf{P}_{\mathcal{M}}$ is goal independent — its definition does not depend on the initial goal; and the third, $\mathbf{G}_{\mathcal{M}}$, depends on the particular choice of initial goal. Observe that the modules $\mathbf{P}_{\mathcal{M}}$ and $\mathbf{G}_{\mathcal{M}}$ are in general not predicate disjoint. In [8] the authors describe an implementation for goal independent analysis of call patterns. The basic idea is to evaluate the meaning

$$\mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{calls}(\mathbf{P};\mathbf{G})) = \mathcal{F}^{\mathcal{A}}(\alpha(\mathbf{P}) \cup \alpha(\mathbf{P}_{\mathcal{M}}) \cup \alpha(\mathbf{G}_{\mathcal{M}}))$$

of the abstract magic program as follows. The goal-independent part $\mathcal{F}^{\mathcal{A}}[\alpha(\mathbf{P}) \cup \alpha(\mathbf{P}_{\mathcal{M}})]$ is evaluated first by applying symmetric composition – which in this case is always guaranteed to generate bounded (in fact, binary) clauses. Then, when the initial goal is specified, the rest of the analysis is carried out using directed composition with the goal dependent part. This approach is shown to give call patterns for an initial goal in a highly efficient manner, once the goal independent phase of the analysis has been performed.

7 Related Work

Several other compositional semantics for logic programs have been proposed in the literature. These include the work of Brogi and Turini [4], and Gaifmann *et al.* [20]. In [4] the compositional semantics is provided by composing the $\mathbf{T_P}$ 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. Logical semantics for modules in logic programs have been proposed by a number of authors [5, 29]. These are typically based on various extensions to Horn logic: for example, Chen's treatment of modules [5] is based on second-order logic, while Miller's [29] uses implication goals in clause bodies. In [12] Comini *et al.* define a taxonomy of semantics that can be derived by abstracting **SLD** trees, and preserve properties like compositionality. In [23], the authors introduce an operation for functional combination of semantics, providing a systematic way to derive compositional semantics for logic programs. In these works, the semantics appear to be somewhat more complicated than that considered in [3], 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 incremental analysis of logic programs, where analysis can be carried out even if the program being analyzed is not available in its entirety, has been investigated by Hermenegildo *et al.* [24]. While the underlying motivation for this work resembles ours in many ways, the details differ substantially. In particular, the approach of Hermenegildo *et al.* involves re-analyzing (parts of) a program in response to changes to the program, while our approach involves first computing the abstract semantics for different modules and then composing these abstract semantics.

The problem of program analysis across module boundaries for imperative languages has been considered by a number of researchers: Cooper *et al.* [13] and Tichy *et al.* [32] 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 [31] 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. In particular, there is no notion of "composition of abstract semantics" and 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, if symmetric composition is applied, it is necessary to reanalyze only the modules that have actually changed: the effects of these changes are propagated by composition of abstract semantics.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: algebraic properties and efficient representation. In B. Le Charlier, editor, *Proceedings of the 1st Int. Static Analysis Symp. (SAS '94)*, volume 864 of *Lecture Notes in Computer Science*, pages 266–280. Springer-Verlag, Berlin, 1994.

[2] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[3] A. Bossi, M. Gabbrielli, G. Levi, and M.C. Meo. A compositional semantics for logic programs. *Theoretical Computer Science*, 122(1–2):3–47, 1994.

[4] A. Brogi and F. Turini. Fully abstract compositional semantics for an algebra of logic programs. *Theoretical Computer Science*, 149(2):201–229, 1995.

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

[6] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124(1):93–126, 1994.

[7] 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.

[8] M. Codish and B. Demoen. Analysing Logic Programs using "`Prop`"-ositional Logic Programs and a Magic Wand. *The Journal of Logic Programming*, 25(3):249-274, 1995.

[9] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems* 16(3):649-686, ACM Press, 1994.

[10] M. Codish, M. García de la Banda, M. Bruynooghe, and M. Hermenegildo. Exploiting goal independence in the analysis of logic programs. To appear in *The Journal of Logic Programming*.

[11] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. In *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, IEEE Press, pages 136-145, 1996.

[12] M. Comini and G. Levi and M.C. Meo. Compositionality of *SLD*-derivations and their abstractions. In J. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)*. The MIT Press, 1995.

[13] K.D. Cooper, K. Kennedy, and L. Torczon. Interprocedural Optimization: Eliminating Unecessary Recompilation. Technical Report Rice COMP TR85-7, Dept. of Computer Science, Rice University, Houston, 1985.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. Gabbrielli, R. Giacobazzi, and D. Montesi. Modular logic programs over finite domains. In D. Saccà, editor, *Proc. Eight Italian Conference on Logic Programming*, pages 663–678, 1993. (`URL: http://www.di.unipi.it/~giaco/papers.html`)

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

[20] 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.

[21] R. Giacobazzi. Semantic Aspects of Logic Program Analysis. *Ph.D. Thesis*, Dipartimento di Informatica, University of Pisa, TD-18/93, 1993.
(`URL: http://www.di.unipi.it/~giaco/papers.html`)

[22] R. Giacobazzi, S. K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191-248, 1995.

[23] R. Giacobazzi and F. Ranzato. Functional dependencies and Moore-set completions of abstract interpretations and semantics. In J. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)*, pages 321–335. The MIT Press, 1995.

[24] M. Hermenegildo, G. Puebla, K.Marriott, and P.J. Stuckey. Incremental Analysis of Logic Programs. In *Proc. Twelfth International Conference on Logic Programming*, pages 797–811.

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

[26] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

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

[28] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

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

[30] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 140–159. MIT Press, 1988.

[31] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. In *Proc. ACM SIGPLAN-90 Conference on Programming Language Design and Implementation*, pages 28–39. ACM, 1990.

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