Abductive Analysis of Modular Logic Programs

ROBERTO GIACOBAZZI, *Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy.*
E-mail: giaco@di.unipi.it

Abstract

We introduce a practical method for abductive analysis of modular logic programs. This is obtained by reversing the deduction process, which is usually applied in static-dataflow analysis of logic programs, on generic, possibly abstract, domains for analysis. The approach is validated in the framework of abstract interpretation. The abduced information provides an abstract specification for program modules which can be of assistance both in top-down development of programs and in compile-time optimization. To the best of our knowledge this is the first application of abductive reasoning in dataflow analysis of logic programs.

Keywords: Static analysis, abstract interpretation, logic programming, modular logic programs, abduction, program specification.

1 Introduction

Dataflow analysis by abstract interpretation is typically intended to statically derive approximated information (analysis) about run-time properties of programs, by applying rules (clauses) to initial states (goals), with the inference of a result. In logic programming, this *deductive* approach to program analysis is usually based on abstract unfolding (i.e. replacement + abstract unification) and it is shared by most of the frameworks proposed in the literature (e.g., see [3, 6, 7, 8, 14, 29]). A similar deductive method is also applied by Codish *et al.* [8] in compositional analysis in presence of modules. Compositional abstract interpretation is essential for large program development so that altering one module does not require re-analysis of the entire program. A logic program is viewed as consisting of a set of modules, each module defining a subset of the program's predicates.

J. Logic Computat., To appear 1997

The abstract meaning of a module corresponds to its analysis and composition of abstract meanings corresponds to composition of analyses. Analyses are constructed by abstract interpretation of the compositional semantics in [5]. Deduction is obtained by iterating unfolding of program clauses with particular definitions for their extensional part. These definitions can either be assumed imported from other modules (leaving them "open") or as part of the deduction itself in the fixpoint iteration. In compositional analysis and semantics, abstract and concrete semantic objects have then a clause-like form.

While deduction is a common aspect in both compositional and non compositional dataflow analysis of logic programs, the reverse of deduction makes sense in modular logic programming. The process of reversing deduction is logically justified by the *abductive reasoning* (an excellent survey on the wide literature on abduction is in [27]). Abductive reasoning is formulated as follows: "given a set of rules (clauses) P and a conjunction of atoms (an *observation* or simply a goal) G, find the set of atoms Q (the *hypothesis*) such that G is a logical consequence of $P \cup Q$" (see [27, 30]). Atoms in Q are indicated as *abducible information*. In this paper we consider abduction in the context of static program analysis. We join abductive reasoning and abstract interpretation to derive program properties of modules, once the specification of the properties of their composition is given. We call this analysis *abductive*. The idea is to show which abstract properties a program module Q (or the semantics of Q) have to satisfy, so that the composition with a program module P (i.e., $P \cup Q$) satisfies a given observation.

Abductive analysis is particularly useful in analysis and development of modular logic programs, as it provides a systematic method to analyze module interaction. Indeed, the applicability of transformation or optimization techniques to program modules usually depends on the definition of their "imported" (open) predicates. We specify this dependency by a kind of "precondition" on the semantics of each module, which can be statically derived by abductive analysis. The resulting *abstract specification* can be interpreted as a constraint on the possible semantics for the open predicates of a module, which has to be satisfied in order to perform program transformations and optimizations. This specification can also be used as a guide in top-down development of module extensions which are consistent with the intended program behavior. Indeed, there is an interesting relation between abduction and specification techniques based on *weakest-prespecifications* ([23]). We prove that abductive analysis precisely corresponds to compute an abstract weakest-prespecification with respect to program composition defined by set-union of clauses, i.e., the weakest observation that must be met by the program module Q in order that the composition $P \cup Q$ will accomplish the original task specified by its specification. A semantic account of this is provided by showing that abstract abduction corresponds to the abstract *weak-op-inverse* (cf. [23]) of the traditional T_P-based semantics of computed answer substitutions introduced in [16]. This result further justifies the use of abductive analysis in systematic program development.

The rest of this paper is structured as follows: Section 2 presents some preliminary definitions and notations. Section 3 resumes some results on compositional semantics and analysis of modular logic programs. The main contribution is in Section 4, where we characterize abduction as composition inverse, and we introduce a practical method for abductive analysis of modular logic programs. Termination of abductive analysis is discussed in Section 5, while applications are discussed in Section 6. Section 7 concludes. The Appendix contains proofs of related results. This paper is the extended version of [19].

2 Preliminaries

In the following we assume familiarity with the standard definitions and notation for logic programs [2], constraint logic programs [26, 21], abstract interpretation [11, 12] and lattice theory [4].

2.1 Mathematical notation

Let A and B be sets. The powerset of A is denoted by $\wp(A)$. Sequences in A^* are typically denoted by $\langle a_1, \ldots, a_n \rangle$ or simply $a_1, \ldots, a_n$, for a_i's symbols in A. We let $\langle a_i \rangle_{i=1}^{n}$ denote the sequence $a_1, \ldots, a_n$. The empty sequence is denoted Λ. Concatenation of sequences $s_1, s_2 \in A^*$ is denoted $s_1 :: s_2$. $a|s$ denotes a sequence with first element a. The set A equipped with a partial order $\leq$ is denoted $\langle A, \leq \rangle$. If A is a poset, we usually denote $\leq_A$ the corresponding partial order. By $x < y$ we mean $x \leq y$ and $x \neq y$. If $I \subseteq A$ then $\downarrow I = \{x \in A \mid \exists y \in I.\ x \leq_A y\}$. For $x \in A$, $\downarrow x$ is a shorthand for $\downarrow \{x\}$. $\wp^{\downarrow}(A)$ denotes the set of *order ideals* of A, where $I \subseteq A$ is an order ideal if $I = \downarrow I$. $\wp^{\downarrow}(A)$ is a complete lattice with respect to set-theoretic inclusion. A complete lattice A with partial ordering $\leq$, least upper bound $\vee$, greatest lower bound $\wedge$, least element $\bot = \vee\emptyset = \wedge A$, and greatest element $\top = \wedge\emptyset = \vee A$, is denoted $\langle A, \leq, \vee, \wedge, \top, \bot \rangle$. In the following we will often abuse notation by denoting lattices with their poset notation.

We write $f : A \to B$ to mean that f is a total function of A into B. The identity operator $\lambda x.x$ is often denoted id. By $g \circ f$ we denote the composition of the functions f and g, i.e., $\forall x.\ (g \circ f)(x) = g(f(x))$. The set of fixpoints of a function f is denoted $\mathit{fp}(f)$, and the least fixpoint (if it exists) is denoted $\mathit{lfp}(f)$. The ordinal power of an operator f on a complete lattice $\langle A, \leq, \vee, \wedge, \top, \bot \rangle$ is defined as $f \uparrow 0(X) = X$, $f{\uparrow}\alpha(X) = f(f{\uparrow}(\alpha - 1)(X))$ for every successor ordinal α, and $f{\uparrow}\alpha(X) = \vee_{\delta<\alpha} f{\uparrow}\delta(X)$ for every limit ordinal α. We omit X if $X = \bot$. It is known that when f is continuous, $\mathit{lfp}(f) = f \uparrow \omega$. If A and B are posets and $\alpha : B \to A$, $\gamma : A \to B$ are monotone maps such that for any $x \in B$ and $y \in A$, $x \leq_B \gamma(\alpha(x))$ and $\alpha(\gamma(y)) \leq_A y$, then we call the quadruple (B, α, A, γ) a *Galois connection* between B and A. If in addition for any $y \in A$, $\alpha(\gamma(y)) = y$, then we call (B, α, A, γ) a *Galois insertion* of A in B. In the following we call (B, α, A, γ) a Galois insertion. In the setting of abstract interpretation, B and A are called, respectively, the *concrete* and the *abstract domain*, and they are assumed to be complete lattices, whereas α and γ are called the *abstraction* and the *concretization* maps, respectively. If (B, α, A, γ) is a Galois insertion, then the concretization and abstraction mappings, γ and α, are 1-1 and onto, respectively. Moreover, in any Galois connection α is additive and γ is co-additive, viz. for any $X \subseteq B$ and $Y \subseteq A$: $\alpha(\vee_B X) = \vee_A \alpha(X)$ and $\gamma(\vee_A Y) = \vee_B \gamma(Y)$. Any Galois connection may be lifted to a Galois insertion identifying in an equivalence class those values of the abstract domain with the same concrete meaning. This process is known as *reduction* of the abstract domain.

2.2 Logic programming

Throughout, we assume a fixed (non-trivial) set of function symbols Σ, a fixed set of predicate symbols Π and a fixed denumerable set of variables *Var*, defining a first-order language $\mathcal{L}$. With each function symbol $f \in \Sigma$ and predicate symbol $p \in \Pi$ is associated a unique natural number called its *arity*: A (predicate or function) symbol f with arity n is written f/n. The set of terms over Σ and *Var* is denoted *Term*. Herbrand constraints are finite conjunctions of equations between terms. The system of such Herbrand 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*, and $\wedge$ as *glb*. We abuse by denoting $(\mathcal{C}, \wedge)$ the corresponding *concrete constraint system*. 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 on $\mathcal{H}$ (see, for example, [26]). This simplifies somehow the presentation. A *goal* is of the form $\sigma \,[\!]\, \bar{a}$, where σ is a Herbrand constraint and $\bar{a}$ is a sequence of atoms. We occasionally abuse notation and treat the sequence of atoms $\bar{a}$ as a set, and write $b \in \bar{a}$ to refer to an element of this set. The empty sequence of atoms is denoted by *true*. A (constrained) *Horn clause* is an object of the form $h \longleftarrow \sigma \,[\!]\, \bar{b}$ where h is an atom (the *head*), and $\sigma \,[\!]\, \bar{b}$ is a goal (the *body*). When $\sigma = true$ we write $h \longleftarrow \bar{b}$. The set of clauses constructed from elements of *Atom* is denoted *Clause*. We will abuse by considering unit clauses and atoms as equivalent notions. We say that a clause $h \longleftarrow \sigma \,[\!]\, \bar{b}$ is *normalized* if σ is in solved form and $h \longleftarrow \bar{b}$ is linear, i.e., contains no multiple occurrences of any variable, and flat, i.e., contains no function symbols or constants. The set of variables occurring in a syntactic object (viz. a term, atom, clause) t is denoted $var(t)$. In the following we assume *finitary syntactic objects* only, which are syntactic objects s such that $var(s)$ is a finite set. The set of predicates in a syntactic object s is denoted $pred(s)$. A logic program (or module) is a finite set of clauses.

A *substitution* is a mapping from *Var* to *Term* which acts as the identity almost everywhere. 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 *most general unifier*) of σ. In the following of this paper we will often abuse notation by using $\models$ to denote different notions of satisfiability. We let the reader to distinguish them according to the context. We fix a partial function *mgu* which maps a pair of syntactic objects to an idempotent most general unifier of the objects if it exists. For a syntactic object t and constraint σ, we write $\sigma(t)$ to denote $t\theta$ where $\theta = mgu(\sigma)$. For a set of constraints Θ, $\Theta(t) = \{\sigma(t) | \sigma \in \Theta\}$. A syntactic object t is *more general* than t' (denoted $t' \leq t$) iff there exists a substitution θ such that $t' = t\theta$. Syntactic objects t_1 and t_2 are *equivalent up to renaming*, denoted $t_1 \sim t_2$, iff $t_1 \leq t_2$ and $t_2 \leq t_1$. $\Theta_1 \sim \Theta_2$ iff for any syntactic object s: $\Theta_1(s) \sim \Theta_2(s)$. $Term_{/\sim}$, $Atom_{/\sim}$, $Clause_{/\sim}$ and $\mathcal{C}_{/\sim}$ are all complete lattices with respect to $\leq$. With abuse of notation, they will be often denoted *Term*, *Atom*, *Clause* and $\mathcal{C}$. Moreover, since all the definitions in the paper are independent on syntactic variable names, we will let the corresponding objects to denote their equivalence classes by renaming. In particular, following Bossi *et al.* [5], we consider sets of clauses modulo the following notion of equivalence between clauses: We say that $(h \longleftarrow \sigma \mathrel{[\![} \bar{b}) \approx (h' \longleftarrow \sigma' \mathrel{[\![} \bar{b}')$ iff there exists a permutation $\bar{b}''$ of atoms in $\bar{b}'$ such that $\sigma(h \longleftarrow \bar{b}) \sim \sigma'(h' \longleftarrow \bar{b}'')$. This identifies clauses which differ syntactically (up to permutation of atoms in the body), yet providing the same implicational relationship between the clause body and its head. For simplicity of exposition, we will abuse notation and assume that a clause represents its equivalence class under $\approx$, and write *Clause* rather than $Clause_{/\approx}$. For a syntactic object s and a set of equivalence classes of objects I under $\sim$ or $\approx$, 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.

3 Concrete and abstract semantics for composition

In the following, we summarize the main results in concrete semantics for composition in [5] and compositional analysis in [8]. If $P_1, \ldots, P_n$ are logic program modules, then $P = \cup_{i=1}^{n} P_i$ is a *modular logic program*. Such a program is said to be *predicate disjoint* if no predicate is defined in more than one module. We assume that modular programs are predicate disjoint unless otherwise specified. This assumption, considered also in [8, 27], somewhat simplifies the presentation. For a 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 (we often call them *abducible predicates*)[1].

[1]The predicate disjoint restriction has the advantage that abducible information (for *open*(P)) is *basic* [27], i.e. it is not defined in terms of other abducible information. For a set of abducible predicates, we can always transform P (as suggested in [27]) in such a way that it does not have definitions for them: For each abducible predicate p, we can introduce a new predicate δ_p and add

3.1 Concrete semantics

The base semantics is the compositional bottom-up semantics of Bossi *et al.* [5], instantiated to the case of predicate disjoint modules. In [5], an *interpretation* is any element in $Int = \wp(Clause)$. Programs and interpretations have then the same structure (i.e. sets of clauses). The concrete semantics is formalized in terms of unfolding of clauses $unf : Int \times Int \to Int$, specifying every possible way to unfold each literal in each clause of P_1 once using clauses in P_2 (see [28]):

$$unf(P_1, P_2) = \left\{ h \longleftarrow \sigma' \,[\!]\, \bar{b}_1 :: \cdots :: \bar{b}_n \;\middle|\; \begin{array}{l} c = h \longleftarrow \sigma \,[\!]\, g_1, \ldots, g_n \in P_1 \\ \langle h_i \longleftarrow \sigma_i \,[\!]\, \bar{b}_i \rangle_{i=1}^{n} \ll_c P_2 \\ \sigma' = \sigma \;\wedge\; \bigwedge_{i=1}^{n} (\sigma_i \wedge \{g_i = h_i\}) \\ \mathcal{H} \models \sigma' \end{array} \right\}$$

unf is a binary associative operator (see [15]) on *Int*, which is additive on its first argument and continuous on its second one. This operator is of interest as it can be applied to formalize both top-down and bottom-up semantics for logic programs (cf. [28]). A bottom-up semantics for open logic programs is defined in [5] in terms of "iterated unfolding" — that is, repeatedly unfolding the clauses in a program until further unfolding produces no change. The fixpoint semantics of a program P is given by the function $\mathcal{F} : Int \to Int$, defined as $\mathcal{F}(P) = lfp(T_P)$, where $T_P : Int \to Int$ is defined as $T_P(I) = unf(P, I \cup \Phi_P)$, and where for any program P,

$$\Phi_P = \left\{ \; p(\bar{x}) \longleftarrow p(\bar{x}) \;\middle|\; p \in open(P) \; \right\}.$$

The idea of Bossi *et al.* [5], is to unfold only those predicates which have a definition, so that the meaning of a program becomes dependent on the meaning of its open predicates. Technically, this is obtained by adding the tautological clauses Φ_P for the open predicates of a program P and unfolding all predicates in the body of a clause. Notice that the unfolding of an open predicate in a clause body with a tautological clause is basically a "no-op". Moreover, note that if $open(P) = \emptyset$ then $\Phi_P = \emptyset$ and $T_P(I) = unf(P, I)$. When I consists of unit clauses (facts), the fixpoint operator gives precisely the (non-ground) fixpoint semantics of Falaschi *et al.* [16]. The following theorem (see Theorem 2.13 and Corollary 4.15 in [5]) formalizes the notion of compositional semantics based on clause unfolding and provides the basis for the compositional analyses described below.

THEOREM 3.1 [concrete composition [5]]
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 3.2 Consider the following modular logic program with modules P_{lg} and P_{num}:

the clause $p(x) \longleftarrow \delta_p(x)$. The resulting program is such that $p \notin open(P)$ but $\delta_p \in open(P)$ (i.e. δ_p becomes abducible). It is worth noting that this restriction still allows mutually recursive modules.

P_{lg}
```
gt(s(X),0) :- num(X).
gt(s(X),s(Y)) :- gt(X,Y).
le(0,Y) :- num(Y).
le(s(X),s(Y)) :- le(X,Y).
```

P_{num}
```
num(0).
num(s(X)) :- num(X).
```

The unfoldings of P_{lg} specify the relations "less or equal" and "greater than" on a domain of values specified by the predicate `num` which is open. Evaluation of $\mathcal{F}(P_{lg})$ proceeds as follows:

$$(1)\ \begin{array}{l} gt(s(X),0) \longleftarrow num(X) \\ le(0,Y) \longleftarrow num(Y) \end{array} \quad (2)\ \begin{array}{l} gt(s(s(X)),s(0)) \longleftarrow num(X) \\ le(s(0),s(Y)) \longleftarrow num(Y) \end{array} \quad \ldots.$$

Since $open(P_{num}) = \emptyset$, $\mathcal{F}(P_{num}) = \{num(0), num(s(0)), num(s(s(0))), \ldots\}$. The meaning of $P_{lg} \cup P_{num}$ can be evaluated directly or by applying Theorem 3.1. In both cases the result corresponds to the standard meaning as provided by the semantics of [16].

3.2 Abstract semantics and deductive analysis

The relevance of compositionality in logic program analysis has been firstly addressed in [8] by introducing a framework which is based on the compositional semantics $\mathcal{F}$, and which formalizes the analysis in terms of the standard theory of abstract interpretation ([12]). In this section we briefly introduce some notions and results from [8, 18]. For the sake of completeness, we include in the appendix the proofs of those results which are relevant for abductive analysis. We follow [8] by assuming that $(Int, \alpha, AInt, \gamma)$ is a Galois insertion, and that the abstract semantics $\mathcal{F}^{\mathcal{A}} : AInt \rightarrow AInt$ safely mimics $\mathcal{F}$, i.e. for each $I \in Int$: $\alpha(\mathcal{F}(I)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\alpha(I))$. The abstract semantics of a program module P is $\mathcal{F}^{\mathcal{A}}(\alpha(P))$.

THEOREM 3.3 [correctness of compositional analysis [8]]
Let $(Int, \alpha, AInt, \gamma)$ be a Galois insertion, and $\mathcal{F}^{\mathcal{A}} : AInt \rightarrow AInt$ be a monotonic and safe approximation of $\mathcal{F}$. Then, for any program module $P_1, P_2 \in Int$:

$$\alpha(\mathcal{F}(P_1 \cup P_2)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P_1)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(P_1)))$$

PROOF. See the Appendix. ∎

A safe approximation of $\mathcal{F}$ can be obtained by abstracting unfolding. The definition of the function $\mathcal{F}^{\mathcal{A}}$ is given in [8] in terms of *abstract unfolding* of clauses, which is clause unfolding relatively to an abstract operator of unification, defined on an abstract domain of substitutions. We generalize the approach in [8], by abstracting unfolding on arbitrary domains of constraints. We firstly introduce the notion of *abstract constraint system*, which generalizes in a single algebraic structure both the notion of abstract unification and the domain of abstract substitution.

Definition 3.4 [abstract constraint system]
An *abstract constraint system* $(\mathcal{A}, \otimes)$, is a complete lattice $\langle \mathcal{A}, \preceq \rangle$, the domain of abstract constraints, together with a binary associative and commutative operator $\otimes$, such that $(\mathcal{C}, \alpha_{\mathcal{A}}, \mathcal{A}, \gamma_{\mathcal{A}})$ forms a Galois insertion, and $\otimes$ is correct with respect to $\wedge$, i.e. $\alpha_{\mathcal{A}}(c_1 \wedge c_2) \preceq_{\mathcal{A}} \alpha_{\mathcal{A}}(c_1) \otimes \alpha_{\mathcal{A}}(c_2)$.

An abstract constraint system $(\mathcal{A}, \otimes)$ is noetherian if $\mathcal{A}$ does not contain an infinite ascending chain of constraints with a bounded set of free variables. Formally, this condition can be specified by assuming that there is no infinite set $\{\kappa_i\}_{i \geq 0} \subseteq \mathcal{A}$ such that $\gamma_{\mathcal{A}}(\kappa_i)(s) \subset \gamma_{\mathcal{A}}(\kappa_{i+1})(s)$ for some syntactic object s and $i \geq 0$. Analogously, a constraint system is finite if there is no infinite set $\{\kappa_i\}_{i \geq 0} \subseteq \mathcal{A}$ such that $\gamma_{\mathcal{A}}(\kappa_i)(s) \neq \gamma_{\mathcal{A}}(\kappa_j)(s)$ for some syntactic object s and $i \neq j$ (for more details on noetherian and finite constraint systems see [21]). In the following we assume noetherian abstract constraint systems $\mathcal{A}$, where $\otimes$ and $\oplus$ are the greatest lower bound and least upper bound in $\langle \mathcal{A}, \preceq \rangle$.

Example 3.5 The running example of abstract domain throughout the paper is *Pos* ([1, 10]), defined by Marriott and Søndergaard for ground dependency analysis. We restrict the definition of *Pos* to a finite (non-empty) set of *variables of interest* $V \subseteq \mathit{Var}$. Pos_V is the finite lattice (indeed Boolean lattice) of positive Boolean functions on V, where a Boolean function f is positive if $f(\mathit{true}, \ldots, \mathit{true}) = \mathit{true}$. In the following we will often denote *Pos* the domain Pos_V for some finite (non-empty) set $V \subseteq \mathit{Var}$. *Pos* is ordered by logical implication $\rightarrow$, and therefore the *lub* and *glb* in *Pos* are given by logical disjunction and conjunction respectively. The domain *Pos* can therefore be represented by propositional formulae, with basic connectives $\wedge$, $\vee$ and $\leftrightarrow$. *Pos* provides a concise representation for abstract substitutions which describe ground dependency relations among their arguments. The generalization to Herbrand constraints is straightforward (e.g. see [21]). A propositional formula with connectives $\{\leftrightarrow, \vee, \wedge\}$ is associated with each constraint. For example, the formula x represents any constraint ϑ such that $\vartheta(x)$ is ground, while $x \leftrightarrow (y \wedge z)$ represents any constraint ϑ such that $\mathit{var}(\vartheta(x)) = \mathit{var}(\vartheta(y)) \cup \mathit{var}(\vartheta(z))$ (x is ground iff y and z are ground). $\vee$ is used to represent sets containing substitutions, and exploiting different ground dependencies. We call *ground dependency* any element in *Pos*. We say that a truth assignment ξ satisfies a propositional formula f, written $\xi \models f$ iff $\xi(f)$ is a tautology. Define the truth assignment $\mathit{asg} : \mathcal{C} \rightarrow \mathit{Var} \rightarrow \{\mathit{true}, \mathit{false}\}$ such that, given $\theta \in \mathcal{C}$, $\mathit{asg}(\theta)(x) = \mathit{true}$ iff $\theta(x)$ is ground (i.e. $\mathit{var}(\theta(x)) = \emptyset$). Note that ground dependencies are closed under instantiation, namely if θ grounds x then any $\theta' \leq \theta$ grounds x too. Thus $\gamma_{\mathit{Pos}}(\kappa) = \{\theta \in \mathcal{C} \mid \theta' \leq \theta \Rightarrow \mathit{asg}(\theta') \models \kappa\}$. The corresponding structure is $(\mathit{Pos}, \wedge)$ which is well known to be finite, correct and to form a Galois insertion with the concrete constraint system $(\mathcal{C}, \wedge)$ (e.g. see [10]), indeed being a finite abstract constraint system.

Abstract semantic objects are descriptions (on $\mathcal{A}$) of clauses. An *abstract clause* is an object of the form $h \longleftarrow \kappa \;[\![\; \bar{b}$ such that $\kappa \in \mathcal{A}$ and $h \longleftarrow \mathit{true} \;[\![\; \bar{b}$ is normalized. The set of abstract clauses on $\mathcal{A}$ is denoted $Clause^{\mathcal{A}}$. An abstract interpretation is then a set of abstract clauses in $\wp(Clause^{\mathcal{A}})$ modulo reduction, viz. modulo equivalence under concretization[2]: $I_1^a \cong I_2^a$ iff $\gamma(I_1^a) = \gamma(I_2^a)$, where $\gamma : \wp(Clause^{\mathcal{A}}) \to \wp(Clause)$ is such that for any $I^a \in \wp(Clause^{\mathcal{A}})$:

$$\gamma(I^a) = \left\{ h \longleftarrow \sigma \parallel \bar{b} \;\middle|\; \begin{array}{l} h \longleftarrow \kappa \parallel \bar{b} \in I^a \\ \sigma \in \gamma_{\mathcal{A}}(\kappa) \end{array} \right\}.$$

We denote $AInt_{\mathcal{A}}$ the set of abstract interpretations on $\mathcal{A}$, and we will often omit $\mathcal{A}$ when this is clear from the context. It is worth noting that, from the definition of γ, $\cong$ includes variable hiding and equivalence up to renaming. This construction induces a Galois insertion $(Int, \alpha, AInt, \gamma)$ as proved in [8]. In particular $\langle AInt, \sqsubseteq, \sqcup, \sqcap, [Clause^{\mathcal{A}}]_{\cong}, [\emptyset]_{\cong}\rangle$ is a complete lattice. The abstract unfolding operator $unf^{\mathcal{A}} : AInt \times AInt \to AInt$ is defined in [8] as:

$$unf^{\mathcal{A}}(P_1^a, P_2^a) = \left\{ h \longleftarrow \kappa' \parallel \bar{b}_1 :: \cdots :: \bar{b}_n \;\middle|\; \begin{array}{l} c = h \longleftarrow \kappa \parallel g_1, \ldots, g_n \in P_1^a \\ \langle h_i \longleftarrow \kappa_i \parallel \bar{b}_i \rangle_{i=1}^n \ll_c P_2^a \\ \kappa' = \kappa \otimes \bigotimes_{i=1}^{n} (\kappa_i \otimes \alpha_{\mathcal{A}}(g_i = h_i)) \end{array} \right\}.$$

The following lemmata specify the basic algebraic properties of abstract unfolding.

LEMMA 3.6
For any abstract constraint system $(\mathcal{A}, \otimes)$, $unf^{\mathcal{A}}$ is left-additive.

PROOF. See the Appendix. ∎

$\cong$ is a congruence relation with respect to $unf^{\mathcal{A}}$, i.e. $unf^{\mathcal{A}}$ is well-defined. Therefore, in the following, we let sets of abstract clauses to denote their equivalence classes under $\cong$.

LEMMA 3.7
If $P_1^a \cong Q_1^a$ and $P_2^a \cong Q_2^a$ then $unf^{\mathcal{A}}(P_1^a, P_2^a) \cong unf^{\mathcal{A}}(Q_1^a, Q_2^a)$.

PROOF. See the Appendix. ∎

As well as concrete unfolding, $unf^{\mathcal{A}}$ is associative, and right-continuous.

LEMMA 3.8
For any abstract constraint system $(\mathcal{A}, \otimes)$, $unf^{\mathcal{A}}$ is associative and right-continuous.

PROOF. See the Appendix. ∎

Componentwise correctness of abstract unfolding is a consequence of correctness of abstract constraint systems.

[2]The name *abstract interpretation* is clearly overloaded here. In the following we leave to the reader to distinguish, according to the context, between abstract interpretation as the theory developed in [11] and as the abstraction of (logical) interpretations of Horn clause logic.

LEMMA 3.9 [unfolding correctness]
Let $(\mathcal{A}, \otimes)$ be an abstract constraint system. Then, for every $I_1, I_2 \in Int$,

$$\alpha(unf(I_1, I_2)) \sqsubseteq unf^{\mathcal{A}}(\alpha(I_1), \alpha(I_2))$$

PROOF. See the Appendix. ∎

The abstract semantics $\mathcal{F}^{\mathcal{A}} : AInt \to AInt$ is defined for a program module P as $\mathcal{F}^{\mathcal{A}}(\alpha(P)) = lfp(T^{\mathcal{A}}_{\alpha(P)})$ where $T^{\mathcal{A}}_{\alpha(P)} : AInt \to AInt$ is defined by $T^{\mathcal{A}}_{\alpha(P)}(I^a) = unf^{\mathcal{A}}(\alpha(P), I^a \sqcup \alpha(\Phi_P))$. By Lemma 3.9, because α is additive in any Galois connection, and $\lambda I^a.unf^{\mathcal{A}}(\alpha(P), I^a)$ is continuous: $\mathcal{F}^{\mathcal{A}}$ is correct with respect $\mathcal{F}$, therefore satisfying the hypothesis of Theorem 3.3.

PROPOSITION 3.10
For any $I \in Int$: $\alpha(\mathcal{F}(I)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\alpha(I))$.

The following example shows a simple compositional analysis of modular logic programs (see [8] for more details).

EXAMPLE 3.11 The deductive compositional analysis for ground dependencies for P_{lg} and P_{num} in Example 3.2 are respectively:

$$\mathcal{F}^{Pos}(\alpha(P_{lg})) = \left\{ \begin{array}{l} gt(x, y) \longleftarrow y \; [\![\; num(x) \\ le(x, y) \longleftarrow x \; [\![\; num(y) \end{array} \right\}$$

$$\mathcal{F}^{Pos}(\alpha(P_{lg})) = \{num(x) \longleftarrow x \; [\![\; true\}$$

The analysis for the program $P_{lg} \cup P_{num}$ can be obtained by composing the analysis for P_{lg} and P_{num}. This provides an approximation for the concrete semantics $\mathcal{F}(P_{lg} \cup P_{num})$.

$$\begin{array}{rcl} \alpha(\mathcal{F}(P_{lg} \cup P_{num})) & \sqsubseteq & \mathcal{F}^{Pos}(\mathcal{F}^{Pos}(\alpha(P_{lg})) \sqcup \mathcal{F}^{Pos}(\alpha(P_{num}))) \\ & \sqsubseteq & \left\{ \begin{array}{l} gt(x, y) \longleftarrow y \wedge x \; [\![\; true \\ le(x, y) \longleftarrow x \wedge y \; [\![\; true \\ num(x) \longleftarrow x \; [\![\; true \end{array} \right\} \end{array}$$

4 Abductive analysis

In this section we introduce the notion of abductive analysis of modular logic programs. We firstly investigate the relation between abduction of program composition in the following section.

4.1 Abduction as composition inverse

Given a program module P, our goal is to find a specification (property) for *open*(P) once a reasonable specification of P is given[3]. A formal account of this process can be given in terms of the well known Dijkstra's *weakest precondition* or *weakest prespecification* [23].

DEFINITION 4.1
f^- is a weak-op-inverse of $f : D \rightarrow D'$ iff (D, f, D', f^-) is a Galois connection.

This terminology has been introduced in [23] to generalize the inverse of sequential command composition ; on specifications. In this context, given a program specification R, the weak-op-inverse f_R^- of $f_R = \lambda X.X; R$ (when it exists) provides the weakest-prespecification, i.e. the weakest specification that must be met by the subprogram P in order that, given the specification W, $P; R$ will accomplish the task W. It is immediate that: f^- exists iff f is additive, and f^- is uniquely determined by f (e.g. see [12]).

Let $\mathcal{F} : \mathit{Int} \rightarrow \mathit{Int}$ be an idempotent operator which is compositional (in the sense of Theorem 3.1), i.e. for any $P_1, P_2 \in \mathit{Int}$: $\mathcal{F}(P_1 \cup P_2) = \mathcal{F}(\mathcal{F}(P_1) \cup \mathcal{F}(P_2))$ (e.g. the unfolding-based semantics in [5] meets these requirements). In this case, the semantics of P is $\mathcal{F}(P)$.

DEFINITION 4.2 [program properties]
A *program property* is any element in $\wp(\mathit{Int})$. A program P *satisfies* a property $\mathcal{P} \in \wp(\mathit{Int})$ (written $P \models \mathcal{P}$) iff $\mathcal{F}(P) \in \mathcal{P}$.

An additive semantics for logic programs can be provided by lifting $\mathcal{F}$ on program properties. For any property $\mathcal{X} \in \wp(\mathit{Int})$, we define the *deductive semantics for composition* for a program P, as follows: $\mathcal{S}_P(\mathcal{X}) = \{\mathcal{F}(P \cup X) \mid X \in \mathcal{X}\}$. The weak-op-inverse of $\mathcal{S}_P$ exists and it is uniquely defined as $\mathcal{S}_P^-(\mathcal{X}) = \{Y \mid \mathcal{F}(P \cup Y) \in \mathcal{X}\}$.

The following result provides an alternative view of abduction, relating abducible information with weak-op-inverses of deductive semantics for composition.

THEOREM 4.3
$Q \models \mathcal{S}_P^-(\mathcal{P})$ iff $P \cup Q \models \mathcal{P}$.

[3] We are considering a rather simple notion of abduction, which is specialized for positive logic programs without *integrity constraints*.

PROOF. Let P and Q be programs and $\mathcal{P} \in \wp(Int)$:

$$\begin{array}{rll} Q \models \mathcal{S}^-_P(\mathcal{P}) & \text{iff} & \mathcal{F}(Q) \in \{\, Y \mid \mathcal{F}(P \cup Y) \in \mathcal{P} \,\} \\ & & [\ \text{definition}\ ] \\ & \text{iff} & \mathcal{F}(P \cup \mathcal{F}(Q)) \in \mathcal{P} \\ & & [\ \text{definition}\ ] \\ & \text{iff} & \mathcal{F}(\mathcal{F}(P) \cup \mathcal{F}(Q)) \in \mathcal{P} \\ & & [\ \text{idempotence}\ ] \\ & \text{iff} & \mathcal{F}(P \cup Q) \in \mathcal{P} \\ & & [\ \text{compositionality}\ ] \\ & \text{iff} & P \cup Q \models \mathcal{P} \\ & & [\ \text{definition}\ ] \end{array}$$

∎

$\mathcal{S}^-_P$ is often useless even when computed on abstract domains as it is not constructively defined from $\mathcal{F}$. In the following of this section we provide: (1) a concise definition of program properties that can easily cope with abstract domains; and (2) a practical method to approximate abduction on abstract domains, providing terminating abductive analysis of modular logic programs. We will be mainly concerned with properties of successful computations, therefore we will always refer to properties as collections of sets of unit-clauses, representing success patterns, in $\wp(Int)$.

4.2 Program properties as (abstract) observations

We introduce the notion of *abstract observation* for a module in a Galois insertion $(Int, \alpha, AInt, \gamma)$. Intuitively, an abstract observation for a program has to describe the meaning of its defined predicates, so that they satisfy a given property. In our case, abstract observations for a program module P will be abstract goals on the predicates defined in P. A goal $\kappa \parallel \bar{b}$ is then intended to describe any program P providing meaning for the atoms in $\bar{b}$, so that the constraint κ is satisfied in any successful derivation for $\bar{b}$ in P.

EXAMPLE 4.4 The abstract observation $x \leftrightarrow y \parallel p(x, y)$ is intended to describe any possible definition for p: $p(x, y) \longleftarrow \kappa$, such that $\kappa \to (x \leftrightarrow y)$. It is easy to see that any program with abstract semantics: $p(x, y) \longleftarrow x \wedge y$ satisfies the observation.

In the concrete semantics, for a program P and goal $G = \sigma \parallel \bar{b}$, G has a *successful derivation* in P iff, by extending G to a clause: $ans \longleftarrow G$ with a new atom ans such that $var(ans) = var(\bar{b})$, there exists $ans \longleftarrow \sigma' \parallel true \in unf(\{ans \longleftarrow G\}, \mathcal{F}(P))$ (e.g. see [28]). In this case we say that σ' is the *answer constraint*. In the following, for any set of variables V, we denote $ans(V)$ a flat and linear atom such that $var(ans(V)) = V$. To simplify the notation, for any syntactic object s, we denote $ans(s)$ the atom $ans(var(s))$, and we will omit V when it can be deduced from the context. We extend this notation to abstract goals (observations) and programs: Let P be a program module and G^a be an abstract observation (or simply an abstract goal), the goal G^a has an *abstract successful derivation* in $\alpha(P)$ iff $ans \longleftarrow \kappa \,[\!]\, true \in unf^{\mathcal{A}}(\{ans \longleftarrow G^a\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))$. The correctness of abstract successful derivations is a consequence of the following lemma.

Lemma 4.5
Let $unf^{\mathcal{A}}$ be a safe and monotone approximation for unf in $(Int, \alpha, AInt, \gamma)$, and $\mathcal{F}^{\mathcal{A}}$ be the corresponding abstract fixpoint semantics. For a program module P and goal G^a:

$$unf(\gamma(\{ans \longleftarrow G^a\}), \mathcal{F}(P)) \subseteq \gamma(unf^{\mathcal{A}}(\{ans \longleftarrow G^a\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))).$$

Proof. Note that by correctness in Proposition 3.10, $\mathcal{F}(P) \subseteq \gamma(\mathcal{F}^{\mathcal{A}}(\alpha(P)))$. The proof follows by $unf^{\mathcal{A}}$ correctness (Lemma 3.9) and monotonicity:

$$\begin{array}{rcl} unf(\gamma(\{ans \longleftarrow G^a\}), \mathcal{F}(P)) & \subseteq & unf(\gamma(\{ans \longleftarrow G^a\}), \gamma(\mathcal{F}^{\mathcal{A}}(\alpha(P)))) \\ & & [\quad \text{monotonicity} \quad] \\ & \subseteq & \gamma(unf^{\mathcal{A}}(\{ans \longleftarrow G^a\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))) \\ & & [\quad \text{correctness} \quad] . \end{array}$$

∎

We are now able to formally define when a program satisfies an abstract observation. Intuitively a program *satisfies* an abstract observation $\kappa \,[\!]\, \bar{b}$ if κ is weaker than (logically implied by) any successful (abstract) derivation for $\bar{b}$ in $\alpha(P)$. This is formalized by the following

Definition 4.6
A program P satisfies an abstract observation $\kappa \,[\!]\, \bar{b}$ with respect to a set of variables V, assuming an abstract constraint $\tilde{\kappa}$, denoted $P \models_V^{\tilde{\kappa}} \kappa \,[\!]\, \bar{b}$, iff for each

$$ans(V \cup var(\bar{b})) \longleftarrow \kappa' \,[\!]\, \bar{b}' \in unf^{\mathcal{A}}(\{ans(V \cup var(\bar{b})) \longleftarrow \tilde{\kappa} \,[\!]\, \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P))),$$

we have $\bar{b}' = true$ and $\gamma_{\mathcal{A}}(\kappa')(ans(V \cup var(\bar{b}))) \subseteq \gamma_{\mathcal{A}}(\kappa)(ans(V \cup var(\bar{b})))$.

$\tilde{\kappa}$ in Definition 4.6 represents the current store (constraint) of the computation. The evaluation of the goal $\bar{b}$ is therefore obtained with respect to $\tilde{\kappa}$. The distinction between $\tilde{\kappa}$ and κ is essential in order to distinguish between the current store of the computation, and what is required to be satisfied in the observation. The condition $\gamma_{\mathcal{A}}(\kappa')(ans(V \cup var(\bar{b}))) \subseteq \gamma_{\mathcal{A}}(\kappa)(ans(V \cup var(\bar{b})))$ specifies that κ' is stronger than κ when restricted to the variables in $V \cup var(\bar{b})$. If $P \models_V^{\tilde{\kappa}} \kappa \,[\!]\, \bar{b}$ and $V \subseteq var(\bar{b})$ and $\tilde{\kappa} = true$, we simply say that P satisfies $\kappa \,[\!]\, \bar{b}$ (denoted $P \models \kappa \,[\!]\, \bar{b}$).

The following example shows the relevance of the set V in the relation $P \models_V G^a$.

EXAMPLE 4.7 Consider the abstract observation $x \parallel p(y)$. It is easy to see that for any program $P = \{p(y) \longleftarrow \kappa \parallel \mathit{true}\}$ (i.e. where κ defines a constraint on y): $P \models x \parallel p(y)$ (viz. $(\exists_x x) \wedge \kappa = \kappa$) but $P \not\models_{\{x\}} x \parallel p(y)$ (viz. $x \wedge \kappa \neq \kappa$).

The notion of observation and the relation $\models$ can be easily extended to cope with abstract programs P^a (viz. interpretations in $AInt$) in the obvious way, i.e. by substituting $\mathcal{F}^{\mathcal{A}}(\alpha(P))$ with $\mathcal{F}^{\mathcal{A}}(P^a)$ in Definition 4.6.

EXAMPLE 4.8
Consider the goal $G^a = x \leftrightarrow z \parallel p(x,y), p(y,z)$ and the program

$$P^a = \{p(x,y) \longleftarrow x \leftrightarrow y \parallel \mathit{true}\}$$

$\mathcal{F}^{Pos}(P^a) = P^a$, moreover, $\mathit{unf}^{Pos}(\{\mathit{ans} \longleftarrow p(x,y), p(y,z)\}, \mathcal{F}^{Pos}(P^a)) = \{\mathit{ans} \longleftarrow (x \leftrightarrow y) \wedge (y \leftrightarrow z) \parallel \mathit{true}\}$, and because $((x \leftrightarrow y) \wedge (y \leftrightarrow z)) \rightarrow (x \leftrightarrow z)$ is a tautology, we can conclude that $P^a \models G^a$.

PROPOSITION 4.9 [downward closure of observations]
Given an abstract observation G^a, $\tilde{\kappa} \in \mathcal{A}$, $V \subseteq \mathit{Var}$ and $I^a, J^a \in AInt$ defining the same set of predicates: If $I^a \models_V^{\tilde{\kappa}} G^a$ and $\gamma(J^a) \subseteq \gamma(I^a)$ then $J^a \models_V^{\tilde{\kappa}} G^a$.

PROOF. Assume $I^a \models_V^{\tilde{\kappa}} G^a$ and $\gamma(J^a) \subseteq \gamma(I^a)$, where $G^a = \kappa \parallel \bar{b}$. It is worth noting that $\gamma(J^a) \subseteq \gamma(I^a)$ iff $J^a \sqsubseteq I^a$ (cf. [12]). By definition, for each

$$\mathit{ans}(V \cup \mathit{var}(\bar{b})) \longleftarrow \kappa' \parallel \bar{b}' \in \mathit{unf}^{\mathcal{A}}(\{\mathit{ans}(V \cup \mathit{var}(\bar{b})) \longleftarrow \tilde{\kappa} \parallel \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(\gamma(I^a))))$$

we have $\bar{b}' = \mathit{true}$ and $\gamma_{\mathcal{A}}(\kappa')(\mathit{ans}(V \cup \mathit{var}(\bar{b}))) \subseteq \gamma_{\mathcal{A}}(\kappa)(\mathit{ans}(V \cup \mathit{var}(\bar{b})))$. By Galois insertions, $\alpha(\gamma(I^a)) = I^a$, and by monotonicity of $\mathit{unf}^{\mathcal{A}}$ and $\mathcal{F}^{\mathcal{A}}$ we have

$$\mathit{unf}^{\mathcal{A}}(C, \mathcal{F}^{\mathcal{A}}(J^a)) \sqsubseteq \mathit{unf}^{\mathcal{A}}(C, \mathcal{F}^{\mathcal{A}}(I^a))$$

where $C = \{\mathit{ans}(V \cup \mathit{var}(\bar{b})) \longleftarrow \tilde{\kappa} \parallel \bar{b}\}$. Therefore, by definition of $\sqsubseteq$, for each

$$\mathit{ans}(V \cup \mathit{var}(\bar{b})) \longleftarrow \kappa'' \parallel \bar{b}' \in \mathit{unf}^{\mathcal{A}}(\{\mathit{ans}(V \cup \mathit{var}(\bar{b})) \longleftarrow \tilde{\kappa} \parallel \bar{b}\}, \mathcal{F}^{\mathcal{A}}(J^a))$$

we have: $\bar{b}' = \mathit{true}$ and $\gamma_{\mathcal{A}}(\kappa'')(\mathit{ans}(V \cup \mathit{var}(\bar{b}))) \subseteq \gamma_{\mathcal{A}}(\kappa')(\mathit{ans}(V \cup \mathit{var}(\bar{b})))$, for some

$$\mathit{ans}(V \cup \mathit{var}(\bar{b})) \longleftarrow \kappa' \parallel \mathit{true} \in \mathit{unf}^{\mathcal{A}}(\{\mathit{ans}(V \cup \mathit{var}(\bar{b})) \longleftarrow \tilde{\kappa} \parallel \bar{b}\}, \mathcal{F}^{\mathcal{A}}(I^a)).$$

The thesis follows because by hypothesis:

$$\gamma_{\mathcal{A}}(\kappa')(\mathit{ans}(V \cup \mathit{var}(\bar{b}))) \subseteq \gamma_{\mathcal{A}}(\kappa)(\mathit{ans}(V \cup \mathit{var}(\bar{b}))).$$

∎

An abstract program property (i.e. an element in $\wp(AInt)$) can be associated with each abstract observation G^a, $\kappa \in \mathcal{A}$ and $V \subseteq \mathit{Var}$:

$$\mathcal{P}_V^{\kappa}(G^a) = \{I^a \in AInt \mid I^a \models_V^{\kappa} G^a \text{ and } I^a \text{ contains only facts}\}$$

It is well known (e.g. see [12]) that, if $\langle Int, \subseteq\rangle$ is a complete lattice, and $(Int, \alpha, AInt, \gamma)$ is a Galois insertion, then $I^a \sqsubseteq J^a$ iff $\gamma(J^a) \subseteq \gamma(I^a)$, and $\langle AInt, \sqsubseteq\rangle$ is a complete lattice. By Proposition 4.9, $\mathcal{P}^\kappa_V(G^a)$ is an abstract program property which is downward closed with respect to $\sqsubseteq$, viz. an order-ideal in $\wp^\downarrow(AInt)$. Goals, or equivalently observations, are therefore adequate to concisely represent program properties which are order-ideals of interpretations. Of course, observations cannot describe arbitrary program properties (viz. elements in $\wp(AInt)$), as $(\wp(AInt), \lambda x. \downarrow x, \wp^\downarrow(AInt), id)$ is a Galois insertion, i.e. $\wp^\downarrow(AInt)$ is a strict abstraction of $\wp(AInt)$. The following result is a straightforward consequence of Lemma 4.5 and Proposition 4.9.

Corollary 4.10
Let P be a program, G^a be an observation, $\kappa \in \mathcal{A}$ and $V \subseteq Var$. Then

1. $P \models^\kappa_V G^a \Leftrightarrow \mathcal{F}^\mathcal{A}(\alpha(P)) \in \mathcal{P}^\kappa_V(G^a)$
2. $P \models^\kappa_V G^a \Rightarrow \alpha(\mathcal{F}(P)) \in \mathcal{P}^\kappa_V(G^a)$.

The above notion of satisfiability for abstract observations can be extended to concrete goals and concrete derivations in the obvious way. The following result specifies a relation between satisfiability and concrete derivations for a program.

Theorem 4.11 [correctness of abstract observations]
If a program module P satisfies an abstract observation $\kappa \,[\!]\, \bar{b}$, then for any successful derivation for $\bar{b}$ in P with answer constraint σ, $\sigma(\bar{b}) \in \gamma_\mathcal{A}(\kappa)(\bar{b})$.

Proof. Assume P satisfies the abstract observation $\kappa \,[\!]\, \bar{b}$. Then, by definition, for each $ans(\bar{b}) \longleftarrow \kappa'' \,[\!]\, true \in unf^\mathcal{A}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, \mathcal{F}^\mathcal{A}(\alpha(P)))$:

$$\gamma_\mathcal{A}(\kappa'')(ans(\bar{b})) \subseteq \gamma_\mathcal{A}(\kappa)(ans(\bar{b})).$$

By $\gamma_\mathcal{A}$ monotonicity and because $\gamma_\mathcal{A}(\kappa'')(ans(\bar{b})) \subseteq \gamma_\mathcal{A}(\kappa)(ans(\bar{b}))$ implies $\gamma_\mathcal{A}(\kappa'')(\bar{b}) \subseteq \gamma_\mathcal{A}(\kappa)(\bar{b})$, then for each $ans(\bar{b}) \longleftarrow \sigma \,[\!]\, true \in \gamma(unf^\mathcal{A}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, \mathcal{F}^\mathcal{A}(\alpha(P))))$:

$$\sigma(\bar{b}) \in \gamma_\mathcal{A}(\kappa'')(\bar{b}) \subseteq \gamma_\mathcal{A}(\kappa)(\bar{b}).$$

By Lemma 4.5, for each $ans(\bar{b}) \longleftarrow \sigma \,[\!]\, true \in unf(\{ans(\bar{b}) \longleftarrow \bar{b}\}, \mathcal{F}(P))$:

$$\sigma(\bar{b}) \in \gamma_\mathcal{A}(\kappa)(\bar{b})$$

which concludes the proof. ∎

4.3 Abductive program analysis

In the case of program modules with open predicates, any abstract observation specifies implicitly an abstract observation on the semantics of the open predicates. The abstract observations for the open predicates of a module P can be derived from any observation of P via unfolding. Notice that given an abstract observation G^a for P, $unf^{\mathcal{A}}(\{ans \longleftarrow G^a\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))$ contains only abstract observations for predicates in $open(P)$, thus specifying a set of abstract observations for the open predicates. We say that Q *extends* P iff $open(Q) = \emptyset$ and $open(P \cup Q) = \emptyset$, namely when Q provides a complete definition for the meaning of the open predicates in P.

Correctness as specified in Theorem 3.3 is not enough to define abductive analysis based on $\mathcal{F}^{\mathcal{A}}$. Indeed, the relation $\alpha(\mathcal{F}(P \cup Q)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(Q)))$ does not specify how $\mathcal{F}^{\mathcal{A}}$ distributes over module composition. For abductive analysis we distinguish between two forms of composition.

Definition 4.12 [compositional correctness and completeness]
Let $\mathcal{F}^{\mathcal{A}}$ be a safe approximation for $\mathcal{F}$ in a Galois insertion $(Int, \alpha, AInt, \gamma)$. $\mathcal{F}^{\mathcal{A}}$ is compositional correct iff $\mathcal{F}^{\mathcal{A}}(\alpha(P \cup Q)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(Q)))$. $\mathcal{F}^{\mathcal{A}}$ is compositional complete iff $\mathcal{F}^{\mathcal{A}}(\alpha(P \cup Q)) = \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(Q)))$.

The reason of this distinction will be more clear in Section 5, where the weaker notion of compositional correctness characterizes the loss of precision resulting from some termination strategies. In the following, when not specified otherwise, we assume compositional correctness for $\mathcal{F}^{\mathcal{A}}$. The following theorem justifies the use of "abstract abduction" for abductive analysis of modular logic programs.

Theorem 4.13 [abductive analysis I]
Let P, Q be program modules such that Q extends P, and $\kappa \parallel \bar{b}$ be an abstract observation for P. If for each $ans \longleftarrow \tilde{\kappa} \parallel \bar{g} \in unf^{\mathcal{A}}(\{ans \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))$: $Q \models^{\tilde{\kappa}}_{var(\bar{b})} \kappa \otimes \tilde{\kappa} \parallel \bar{g}$, then $P \cup Q \models \kappa \parallel \bar{b}$.

Proof. The proof is based on $unf^{\mathcal{A}}$ associativity (Lemma 3.8) and compositional correctness. To simplify the notation, in this proof we denote by $\preceq_s$ the preorder on abstract constraints such that $\kappa \preceq_s \kappa'$ iff $\gamma_{\mathcal{A}}(\kappa)(s) \subseteq \gamma_{\mathcal{A}}(\kappa')(s)$. Notice that by the properties of $\otimes$ and because $\gamma_{\mathcal{A}}$ is co-additive in any Galois connection, then for each $\kappa, \kappa' \in \mathcal{A}$: $\gamma_{\mathcal{A}}(\kappa \otimes \kappa')(s) = \gamma_{\mathcal{A}}(\kappa) \wedge \gamma_{\mathcal{A}}(\kappa')(s)$. Moreover, note that $\otimes$ is still the greatest lower bound in $\mathcal{A}/_{=_s}$, where $=_s$ is the equivalence relation induced by $\preceq_s$.
By definition: $Q \models^{\kappa'}_{var(\bar{b})} \kappa' \otimes \kappa \parallel \bar{g}$, where $ans(\bar{b}) \longleftarrow \kappa' \parallel \bar{g} \in unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))$, iff for each $ans(\bar{b} :: \bar{g}) \longleftarrow \kappa'' \parallel true \in unf^{\mathcal{A}}(\{ans(\bar{b} :: \bar{g}) \longleftarrow \kappa' \parallel \bar{g}\}, \mathcal{F}^{\mathcal{A}}(\alpha(Q)))$: $\kappa'' \preceq_{\bar{b}::\bar{g}} \kappa' \otimes \kappa$. By the $unf^{\mathcal{A}}$ definition the previous condition is satisfied iff for each

$$ans(\bar{b} :: \bar{g}) \longleftarrow \kappa'' \parallel true \in unf^{\mathcal{A}}(unf^{\mathcal{A}}(\{ans(\bar{b} :: \bar{g}) \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P))), \mathcal{F}^{\mathcal{A}}(\alpha(Q)))$$

$\kappa'' \preceq_{\bar{b}::\bar{g}} \kappa' \otimes \kappa$, where $ans(\bar{b} :: \bar{g}) \longleftarrow \kappa' \parallel \bar{g} \in unf^{\mathcal{A}}(\{ans(\bar{b} :: \bar{g}) \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))$.
By $unf^{\mathcal{A}}$ associativity (see Lemma 3.8), for each

$$ans(\bar{b} :: \bar{g}) \longleftarrow \kappa'' \parallel true \in unf^{\mathcal{A}}(\{ans(\bar{b} :: \bar{g}) \longleftarrow \bar{b}\}, unf^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)), \mathcal{F}^{\mathcal{A}}(\alpha(Q))))$$

$\kappa'' \preceq_{\bar{b}::\bar{g}} \kappa' \otimes \kappa$ where $ans(\bar{b} :: \bar{g}) \longleftarrow \kappa' \parallel \bar{g} \in unf^{\mathcal{A}}(\{ans(\bar{b} :: \bar{g}) \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))$. Because $\kappa'' \preceq_{\bar{b}::\bar{g}} \kappa' \otimes \kappa$ implies $\kappa'' \preceq_{\bar{b}} \kappa' \otimes \kappa$ and $\kappa' \otimes \kappa \preceq_{\bar{b}} \kappa$, then for each

$$ans(\bar{b}) \longleftarrow \kappa'' \parallel true \in unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, unf^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)), \mathcal{F}^{\mathcal{A}}(\alpha(Q))))$$

$\kappa'' \preceq_{\bar{b}} \kappa$. By the hypothesis, Q extends P, namely $open(Q) = \emptyset$ and $open(P \cup Q) = \emptyset$. Thus:

$$unf^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)), \mathcal{F}^{\mathcal{A}}(\alpha(Q))) = \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)) \cup \mathcal{F}^{\mathcal{A}}(\alpha(Q))).$$

Moreover, by compositional correctness:

$$\mathcal{F}^{\mathcal{A}}(\alpha(P \cup Q)) = \mathcal{F}^{\mathcal{A}}(\alpha(P) \cup \alpha(Q)) \sqsubseteq unf^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)), \mathcal{F}^{\mathcal{A}}(\alpha(Q))).$$

Thus, for each $ans(\bar{b}) \longleftarrow \xi \parallel true \in unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P \cup Q)))$ there exists

$$ans(\bar{b}) \longleftarrow \kappa'' \parallel true \in unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, unf^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)), \mathcal{F}^{\mathcal{A}}(\alpha(Q))))$$

such that $\xi \preceq_{\bar{b}} \kappa''$. Hence, because $\kappa'' \preceq_{\bar{b}} \kappa$, we obtain that for each

$$ans(\bar{b}) \longleftarrow \xi \parallel true \in unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P \cup Q))) : \quad \xi \preceq_{\bar{b}} \kappa$$

which concludes the proof. ∎

An abstract observation for the open predicates of a module P can then be derived by solving the required observation for P in the compositional analysis of the module itself. Notice that, because Q extends P, Q can also be joined with the semantics of all the modules that define predicates in $open(P)$. The converse of Theorem 4.13 is valid only for compositional complete abstract semantics.

THEOREM 4.14 [abductive analysis II]
Let P, Q be program modules such that Q extends P, $\mathcal{F}^{\mathcal{A}}$ be compositional complete and $\kappa \parallel \bar{b}$ be an abstract observation for P. $P \cup Q \models \kappa \parallel \bar{b}$ iff for each $ans \longleftarrow \tilde{\kappa} \parallel \bar{g} \in unf^{\mathcal{A}}(\{ans \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))$: $Q \models^{\tilde{\kappa}}_{var(\bar{b})} \kappa \otimes \tilde{\kappa} \parallel \bar{g}$.

PROOF. The proof is analogous to that of Theorem 4.13. By substituting compositional correctness with compositional completeness in the proof of Theorem 4.13, we obtain that

$$ans(\bar{b}) \longleftarrow \xi \parallel true \in unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P \cup Q)))$$

iff

$$ans(\bar{b}) \longleftarrow \xi \parallel true \in unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, unf^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)), \mathcal{F}^{\mathcal{A}}(\alpha(Q))))$$

Therefore, by hypothesis, we have that for each

$$ans(\bar{b}) \longleftarrow \xi \parallel true \in unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, unf^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P)), \mathcal{F}^{\mathcal{A}}(\alpha(Q))))$$

$\xi \preceq_{\bar{b}} \kappa$. By associativity (Lemma 3.8) we have that for each

$$ans(\bar{b}) \longleftarrow \xi \, [\![\, true \in unf^{\mathcal{A}}(unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P))), \mathcal{F}^{\mathcal{A}}(\alpha(Q)))$$

$\xi \preceq_{\bar{b}} \kappa$. Hence, for each

$$ans(\bar{b} :: \bar{g}) \longleftarrow \tilde{\kappa} \, [\![\, \bar{g} \in unf^{\mathcal{A}}(\{ans(\bar{b}) \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))$$

we have

$$ans(\bar{b} :: \bar{g}) \longleftarrow \xi \, [\![\, true \in unf^{\mathcal{A}}(\{ans(\bar{b} :: \bar{g}) \longleftarrow \tilde{\kappa} \, [\![\, \bar{g}\}, \mathcal{F}^{\mathcal{A}}(\alpha(Q)))$$

and $\xi \preceq_{\bar{b}} \kappa$. We can assume without loss of generality that κ does not affect variables in $\bar{g}$. Therefore we have $\xi \preceq_{\bar{b}::\bar{g}} \kappa$, which implies that $\xi \otimes \tilde{\kappa} \preceq_{\bar{b}::\bar{g}} \kappa \otimes \tilde{\kappa}$. It is worth noting that $\xi \otimes \tilde{\kappa} = \xi$, as ξ is obtained by unfolding of $ans(\bar{b} :: \bar{g}) \longleftarrow \tilde{\kappa} \, [\![\, \bar{g}$. Hence, for each

$$ans(\bar{b} :: \bar{g}) \longleftarrow \xi \, [\![\, true \in unf^{\mathcal{A}}(\{ans(\bar{b} :: \bar{g}) \longleftarrow \tilde{\kappa} \, [\![\, \bar{g}\}, \mathcal{F}^{\mathcal{A}}(\alpha(Q)))$$

$\xi \preceq_{\bar{b}::\bar{g}} \kappa \otimes \tilde{\kappa}$, and therefore by definition: $Q \models^{\tilde{\kappa}}_{var(\bar{b})} \kappa \otimes \tilde{\kappa} \, [\![\, \bar{g}$. ∎

It is worth noting that this result provides an efficient and concise way to approximate, on generic abstract domains, the weak-op-inverse of program composition (cf. Theorem 4.3).

Example 4.15 Let $P = \{p(x, z) \longleftarrow q(x, y), q(y, z)\}$ be an open program, and $G^a = x \leftrightarrow y \, [\![\, p(x, y)$ be an abstract observation for P. As shown in Example 4.8, $Q = \{q(x, y) \longleftarrow x \leftrightarrow y \, [\![\, true\}$ satisfies the abstract observation for the open predicate q: $x \leftrightarrow z \, [\![\, q(x, y), q(y, z)$ which is obtained by solving the observation G^a in the compositional abstract semantics for P. By Theorem 4.13: $P \cup Q$ satisfies G^a.

The unfolding of an abstract observation for a module P into the abstract semantics of P produces a set of abstract observations for the open predicates, yet specifying relations on the original predicates defined in P. Projecting out this information (on the variables of the new observation only) may lead to an inconsistent result, as shown in the following example.

Example 4.16 Consider the program $P = \{p(x, y) \longleftarrow q(y)\}$ and the abstract observation $x \, [\![\, p(x, y)$. By unfolding we obtain the observation $x \, [\![\, q(y)$. Intuitively, no definition for q can instantiate x to ground. This property is correctly modeled by the abductive analysis. However, as shown in Example 4.7, projecting out x can lead to a contradiction, because for any $Q = \{q(x) \longleftarrow \kappa \, [\![\, true\}$: $Q \models x \, [\![\, q(y)$, but $Q \not\models_{\{x\}} x \, [\![\, q(y)$.

This approach provides a "goal-directed" way of figuring out what the properties we should analyze a module Q for, given that we know the relevant properties (e.g. from the perspective of allowing certain optimizations) of another module P.

EXAMPLE 4.17 Consider the definition for `gt` and `le` in Example 3.2 (i.e. P_{lg}) and an abstract observation for `gt` and `le` which assumes their arguments being all ground:

$$x \wedge y \wedge z \wedge w \;[\!]\; gt(x,y), le(z,w).$$

Consider the abduced abstract observation for the open predicate `num`:

$$ans(x,y,z,w) \longleftarrow \kappa \;[\!]\; num(x'), num(w') \in$$
$$unf^{Pos}(\{ans \longleftarrow gt(x,y), le(z,w)\}, \mathcal{F}^{Pos}(\alpha(P_{lg})))$$

where $\kappa = (x \leftrightarrow x') \wedge y \wedge z \wedge (w \leftrightarrow w')$. Intuitively, `num` has to be ground to satisfy the observation for `gt` and `le`. Indeed, if $P_{num} = \{num(x) \longleftarrow x \;[\!]\; true\}$, then: $P_{num} \models^{\kappa}_{\{x,y,z,w\}} x \wedge y \wedge z \wedge w \wedge \kappa \;[\!]\; num(x'), num(w')$.

5 Termination in Abductive Analysis

Termination in abductive analysis boils down into termination of compositional analysis. The first step is to restrict abstract constraint systems to the point of becoming finite or to satisfy the ascending chain condition. However, as observed in [8], compositional analysis based on unfolding may not terminate even with finite or noetherian abstract domains for substitutions (or constraints). This, because the fixpoint iteration may introduce clauses with bodies containing an arbitrary number of open atoms.

EXAMPLE 5.1 Consider the standard program P_{anc} with `par` open:

```
anc(X,Y) :- par(X,Y).
anc(X,Y) :- par(Y,Z),anc(Z,X).
```

It is immediate to prove that

1. $T^{Pos}_{\alpha(P_{anc})} \uparrow 1 = \{anc(x_1,x_2) \longleftarrow (x_1 \leftrightarrow x_3) \wedge (x_2 \leftrightarrow x_4) \;[\!]\; par(x_3,x_4)\}$
2. $T^{Pos}_{\alpha(P_{anc})} \uparrow 2 = \left\{ \begin{array}{ll} anc(x_1,x_2) \longleftarrow & (x_1 \leftrightarrow x_3) \wedge (x_2 \leftrightarrow x_4) \;[\!]\; par(x_3,x_4) \\ anc(x_1,x_2) \longleftarrow & (x_1 \leftrightarrow x_6) \wedge (x_2 \leftrightarrow x_3) \wedge (x_5 \leftrightarrow x_4) \\ & [\!]\; par(x_3,x_4), par(x_5,x_6) \end{array} \right\}$

$\vdots$

Some solutions to this problem have been proposed in [8, 17].

In [8] an additional (and orthogonal) layer of abstraction, the $\star$-*abstraction*[4], is applied to deal with unbounded clause bodies in the abstract semantics. This approach

[4]This technique was originally proposed in [9] as *state-approximation* for suspension analysis of concurrent logic programs.

provides finitary descriptions for arbitrary large clauses, but makes the analysis less precise. Clause bodies are restricted to contain at most one occurrence of each (open) predicate symbol. In this case, since Π is usually finite, the domain of abstract interpretations *AInt* results a finite-height lattice. The $\star$-abstracted compositional semantics is denoted $\mathcal{F}^{\mathcal{A}}_{\star}$. This semantics is compositional correct (see [8, 18]), therefore, by Theorem 4.13, it can be correctly used in abstract abduction. The $\star$-abstraction is specific for some abstract domain for analysis. We will see the case for ground dependencies in Example 5.3 below.

The approach in [17] is instead oriented to provide a compositional complete abstract semantics, which is domain independent. This is obtained by taking, as semantics for P, the result of a finite number of iterations of the $T^{\mathcal{A}}_{\alpha(P)}$ operator, without reaching the fixpoint. While this would not be correct in general, for finite abstract constraint systems (e.g. for *Pos* [1] or *Sharing* [25]) the resulting denotation has the same compositionality properties of the concrete fixpoint semantics. The fixpoint iteration sequence $T^{\mathcal{A}}_{\alpha(P)}\!\uparrow\! n$ $(n \geq 1)$ is stopped when T*-stability* is reached, i.e. whenever for each $Q \in \mathit{AInt}$ containing only unit clauses with predicates in $\mathit{open}(P)$: $\mathit{unf}^{\mathcal{A}}(T^{\mathcal{A}}_{\alpha(P)}\!\uparrow\! n, Q) = \mathit{unf}^{\mathcal{A}}(T^{\mathcal{A}}_{\alpha(P)}\!\uparrow\!(n+1), Q)$. This corresponds to test if the same patterns for the intensional part of clauses in $T^{\mathcal{A}}_{\alpha(P)}\!\uparrow\! n$ and $T^{\mathcal{A}}_{\alpha(P)}\!\uparrow\!(n+1)$ are generated, once any definition for their extensional (open) part is provided. In this case, any additional unfolding step does not add information with respect to the possible extensions of P. The T-stable semantics is denoted $\mathcal{F}^{\mathcal{A}}_{T}$ and it is proved to be compositional complete in [17, 18].

Example 5.2 Consider the Example 5.1. It is easy to prove that $T^{Pos}_{\alpha(P_{anc})} \uparrow 1$ is not T-stable, while $T^{Pos}_{\alpha(P_{anc})} \uparrow 2$ is T-stable. Take for instance

$$Q = \{\mathit{par}(x_1, x_2) \longleftarrow x_1 \leftarrow x_2 \;[\!]\; \mathit{true}\}$$

then:

$$\mathit{unf}^{\mathcal{A}}(T^{Pos}_{\alpha(P_{anc})} \uparrow 1, Q) = \{\mathit{anc}(x_1, x_2) \longleftarrow x_1 \leftarrow x_2 \;[\!]\; \mathit{true}\}$$

$$\mathit{unf}^{\mathcal{A}}(T^{Pos}_{\alpha(P_{anc})} \uparrow 2, Q) = \left\{ \begin{array}{l} \mathit{anc}(x_1, x_2) \longleftarrow x_1 \leftarrow x_2 \;[\!]\; \mathit{true} \\ \mathit{anc}(x_1, x_2) \longleftarrow x_1 \rightarrow x_2 \;[\!]\; \mathit{true} \end{array} \right\}$$

Therefore $\mathit{unf}^{Pos}(T^{Pos}_{\alpha(P_{anc})} \uparrow 1, Q) \neq \mathit{unf}^{Pos}(T^{Pos}_{\alpha(P_{anc})} \uparrow 2, Q)$. However, for any set of unit clauses $Q \in \mathit{AInt}$, $\mathit{unf}^{Pos}(T^{Pos}_{\alpha(P_{anc})} \uparrow 2, Q) = \mathit{unf}^{Pos}(T^{Pos}_{\alpha(P_{anc})} \uparrow 3, Q)$.

The following example shows the applicability of T-stable and $\star$-abstracted semantics in abductive analysis.

Example 5.3 In this example we consider the simpler abstract domain *Dep* adopted from [9]. Elements in Dep_V are propositional formulae on V, for any finite non-empty set $V \subseteq \mathit{Var}$, with connectives $\wedge$ and $\leftrightarrow$, ordered by logical implication. It is straightforward to derive *Dep* by abstractaction of *Pos*. In the following of this example we assume a kind of *merge over all paths* MOP abstract interpretation, i.e. interpretations can only contain single occurrences for any clause. Therefore, any two clauses $h \longleftarrow \kappa \,[\!]\, \bar{b}$ and $h \longleftarrow \kappa' \,[\!]\, \bar{b}$ generated for $h \longleftarrow \bar{b}$ in $\mathcal{F}^{Dep}$ are "merged" into (approximated by) a single clause $h \longleftarrow \kappa \oplus \kappa' \,[\!]\, \bar{b}$. This restriction reduces the search space for T-stability, even if it may be cause of a loss of precision (see [21] for a formal treatment of MOP in analysis of constraint logic programs). Moreover, as proved in [8], $\star$-abstraction for *Dep* with MOP is compositional correct. Consider the "naive reverse" routine P_{rev} where the `append` procedure is open.

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

A finite compositional abstract semantics can be obtained by iterating unfolding until a T-stable sequence is reached. This is obtained after two iterations:

$$\mathcal{F}_T^{Dep}(\alpha(P_{rev})) = \left\{ \begin{array}{l} rev(x_1, x_2) \longleftarrow x_1 \wedge x_2 \,[\!]\, true \\ rev(x_1, x_2) \longleftarrow x_1 \leftrightarrow x_6 \wedge x_2 \leftrightarrow x_7 \wedge x_5 \,[\!]\, append(x_5, x_6, x_7) \end{array} \right\}$$

A $\star$-abstracted semantics is generated by collapsing all occurrences of the same predicate in a body to one "canonical" atom. The collapsing of a sequence $p(\bar{x}_1), \ldots, p(\bar{x}_n)$ of atoms is an atom $p(\bar{x})$ with $\bar{x} \neq \bar{x}_i$ and a dependency relation κ such that each argument x_j of $p(\bar{x})$ represents the set of the j^{th} arguments (x_{i_j} for $i = 1 \ldots n$) in $p(\bar{x}_1), \ldots, p(\bar{x}_m)$, i.e.

$$\kappa = \bigwedge_{x_j \in \bar{x}} \left(x_j \leftrightarrow (\bigwedge_{i=1}^{n} x_{i_j}) \right)$$

Therefore we have:

$$\mathcal{F}_\star^{Dep}(\alpha(P_{rev})) = \left\{ \begin{array}{l} rev(x_1, x_2) \longleftarrow x_1 \wedge x_2 \,[\!]\, true \\ rev(x_1, x_2) \longleftarrow x_1 \leftrightarrow x_6 \wedge x_7 \leftrightarrow (x_2 \wedge x_5) \,[\!]\, append(x_5, x_6, x_7) \end{array} \right\}$$

By Theorem 4.13 with abstract observation $G^a = x_1 \leftrightarrow x_2 \,[\!]\, rev(x_1, x_2)$ for P_{rev}, we abduce the observations

$$ans(x_1, x_2) \longleftarrow \tilde{\kappa}_i \,[\!]\, append(x_5, x_6, x_7) \in unf^{Dep}(\{ans \longleftarrow rev(x_1, x_2)\}, \mathcal{F}_i^{Dep}(\alpha(P_{rev})))$$

for $i \in \{\star, T\}$, where: $\tilde{\kappa}_T = x_1 \leftrightarrow x_6 \wedge x_2 \leftrightarrow x_7 \wedge x_5$ and $\tilde{\kappa}_\star = x_1 \leftrightarrow x_6 \wedge x_7 \leftrightarrow (x_2 \wedge x_5)$.

It is easy to observe that for $i \in \{\star, T\}$: $(\kappa_i \wedge x_5 \wedge (x_6 \leftrightarrow x_7)) \rightarrow (x_1 \leftrightarrow x_2)$. Therefore, for any definition Q of `append` with semantics $\{append(x_1, x_2, x_3) \longleftarrow x_1 \wedge (x_2 \leftrightarrow x_3) \,[\!]\, true\}$ and $i \in \{\star, T\}$:

$$Q \models_{\{x_1, x_2\}}^{\tilde{\kappa}_i} (x_1 \leftrightarrow x_2) \wedge \tilde{\kappa}_i \,[\!]\, append(x_5, x_6, x_7)$$

Therefore, by Theorem 4.13 [5], $P_{rev} \cup Q$ satisfies the observation G^a. Note that the standard definition of `append`, i.e., $\{append(x_1, x_2, x_3) \longleftarrow (x_1 \wedge x_2) \leftrightarrow x_3 \,[\!]\, true\}$

[5] For T-stable semantics ($i = T$) we can also apply Theorem 4.14 and obtain a necessary condition on the semantics of `append`.

satisfies the abduced observation for $i = T$, i.e., $(\kappa_T \wedge (x_5 \wedge x_6) \leftrightarrow x_7) \rightarrow (x_1 \leftrightarrow x_2)$, while being $\kappa_\star$ weaker than κ_T, we need a stronger definition for `append` to satisfy the observation for $i = \star$.

6 Applications

In this section we consider some applications of abductive analysis in typical dataflow analysis problems in logic programming.

6.1 Invariance of module composition

In addition to correctness of abductive analysis, Theorem 4.13 gives also a characterization of the invariance of module composition with respect to an abstract observation. A module P is *invariant* with respect to $open(P)$ for an abstract observation G^a iff for each extension Q of P: $P \cup Q \models G^a$.

PROPOSITION 6.1
Let $\kappa \;[\!]\; \bar{b}$ be an abstract observation for a program module P. If for each $ans \longleftarrow \kappa' \;[\!]\; \bar{g} \in unf^{\mathcal{A}}(\{ans \longleftarrow \bar{b}\}, \mathcal{F}^{\mathcal{A}}(\alpha(P)))$: $\gamma_{\mathcal{A}}(\kappa')(\bar{b} :: \bar{g}) = \gamma_{\mathcal{A}}(\kappa \otimes \kappa')(\bar{b} :: \bar{g})$ then P is invariant with respect to $open(P)$ for $\kappa \;[\!]\; \bar{b}$.

PROOF. Immediate by $\otimes$ monotonicity and Theorem 4.13. ∎

EXAMPLE 6.2 Consider an "insertion sort" routine defined in terms of the following modules

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

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

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

The compositional semantics for P_{sort} (`insert` is open) and P_{ins} (`gt` and `le` are open) are obtained in two steps as T-stable semantics:

$$\mathcal{F}_T^{Pos}(\alpha(P_{sort})) = \left\{ \begin{array}{l} sort(x_1, x_2) \longleftarrow x_1 \wedge x_2 \;[\!]\; true \\ sort(x_1, x_2) \longleftarrow (x_1 \leftrightarrow x_5) \wedge (x_2 \leftrightarrow x_7) \wedge x_6 \;[\!]\; insert(x_5, x_6, x_7) \end{array} \right\}$$

$\mathcal{F}_T^{Pos}(\alpha(P_{ins})) =$

$$\left\{\begin{array}{ll} insert(x_1,x_2,x_3) \longleftarrow & (x_1 \leftrightarrow x_3) \wedge x_2 \parallel true \\ insert(x_1,x_2,x_3) \longleftarrow & x_3 \leftrightarrow (x_1 \wedge x_2) \wedge x_1 \leftrightarrow x_4 \wedge x_2 \rightarrow x_5 \parallel le(x_4,x_5) \\ insert(x_1,x_2,x_3) \longleftarrow & x_3 \leftrightarrow (x_1 \wedge x_2) \wedge x_1 \leftrightarrow x_4 \wedge x_2 \leftrightarrow x_5 \parallel gt(x_4,x_5) \\ insert(x_1,x_2,x_3) \longleftarrow & x_3 \leftrightarrow (x_1 \wedge x_2) \wedge x_1 \leftrightarrow x_4 \leftrightarrow x_4' \wedge x_2 \rightarrow (x_5 \wedge x_5') \\ & \parallel gt(x_4,x_5), le(x_4',x_5') \end{array}\right\}$$

The semantics of P_{gl} is:

$$\mathcal{F}_T^{Pos}(\alpha(P_{lg})) = \left\{\begin{array}{l} gt(x_1,x_2) \longleftarrow x_1 \leftrightarrow x_2 \parallel true \\ le(x_1,x_2) \longleftarrow x_1 \leftrightarrow x_2 \parallel true \end{array}\right\}.$$

Given the abstract observation $G^a = x_1 \leftrightarrow x_2 \parallel sort(x_1,x_2)$, we abduce

$$ans(x_1,x_2) \longleftarrow \tilde{\kappa} \parallel insert(x_5,x_6,x_7) \in unf^{Pos}(\{ans \longleftarrow sort(x_1,x_2)\}, \mathcal{F}_T^{Pos}(\alpha(P_{sort})))$$

where $\tilde{\kappa} = x_1 \leftrightarrow x_5 \wedge x_2 \leftrightarrow x_7 \wedge x_6$. As before, it is easy to observe that the program $P_{ins} \cup P_{gl}$ satisfies the abduced observation, i.e.

$$P_{ins} \cup P_{gl} \models^{\tilde{\kappa}}_{\{x_1,x_2\}} (x_1 \leftrightarrow x_2) \wedge \tilde{\kappa} \parallel insert(x_5,x_6,x_7).$$

By Theorem 4.13, $P_{sort} \cup P_{ins} \cup P_{gl}$ satisfies the observation G^a.

However, notice that $P_{sort} \cup P_{ins}$ is invariant with respect to `gt` and `le` for the observation (on `sort`) G^a. To prove this invariance, we apply Proposition 6.1. $\mathcal{F}_T^{Pos}(\alpha(P_{sort} \cup P_{ins}))$ contains the following definitions for `sort`:

$$\left\{\begin{array}{l} sort(x_1,x_2) \longleftarrow x_1 \leftrightarrow x_2 \parallel true \\ sort(x_1,x_2) \longleftarrow (x_1 \leftrightarrow x_2 \leftrightarrow x_4') \wedge x_5' \parallel le(x_4',x_5') \\ sort(x_1,x_2) \longleftarrow (x_1 \leftrightarrow x_2 \leftrightarrow x_4') \wedge x_5' \parallel gt(x_4',x_5') \\ sort(x_1,x_2) \longleftarrow (x_1 \leftrightarrow x_2 \leftrightarrow x_4' \leftrightarrow x_4'') \wedge x_5' \wedge x_5'' \parallel gt(x_4',x_5'), le(x_4'',x_5'') \end{array}\right\}$$

Note that, for each

$$ans \longleftarrow \kappa' \parallel \bar{b} \in unf^{Pos}(\{ans \longleftarrow sort(x_1,x_2)\}, \mathcal{F}_T^{Pos}(\alpha(P_{sort} \cup P_{ins})))$$

we have $\kappa' = \kappa' \wedge (x_1 \leftrightarrow x_2)$. Therefore, independently on the definition Q of `gt` and `le`: $P_{sort} \cup P_{ins} \cup Q \models G^a$. This is enough to conclude that a different definition for `gt` and `le`, e.g. P_{lg} in Example 3.2, does not affect the ground dependency of `sort`. Moreover, note that both P_{gl} and P_{lg}, when composed with P_{sort}, produce (only) ground bindings for the first and second argument of `sort`.

6.2 Call patterns

Abductive analysis can be used in conjunction with call pattern information. This is achieved by applying the same *Magic Sets* transformation $\mathcal{M}$ suggested in [7]. Recall that for a program P and goal G, $\sigma \in \mathcal{C}$ is a call pattern for a predicate p in $P \cup \{G\}$ iff there exists an SLD-derivation from G to $b|G'$ with computed constraint σ and where $pred(b) = \{p\}$.

Let P be a program and G be the goal $A_1, \ldots, A_n$. It is well known that call patterns depend on the selection rule [2]. The magic-set transformed program $\mathcal{M}(P, G)$ for the Prolog left-to-right selection rule, is obtained by transforming $P \cup \{G\}$ as follows[6]:

1. replace G by the clauses
$$A_i^{\mathbf{c}} \longleftarrow A_1^{\mathbf{a}}, \ldots, A_{i-1}^{\mathbf{a}}$$
for $1 \le i \le n$.

2. replace each clause $H \longleftarrow B_1, \ldots, B_m$ $(m \ge 0)$ in P by the clauses
$$B_i^{\mathbf{c}} \longleftarrow H^{\mathbf{c}}, B_1^{\mathbf{a}}, \ldots, B_{i-1}^{\mathbf{a}}$$
for $1 \le i \le m$, and
$$H^{\mathbf{a}} \longleftarrow H^{\mathbf{c}}, B_1^{\mathbf{a}}, \ldots, B_m^{\mathbf{a}}$$

$\mathcal{M}$ characterizes the call patterns of P with initial goal G as successful computations of the transformed program $\mathcal{M}(P, G)$ (cf. [7]). Therefore, properties can be specified for call patterns as well as for success patterns in $\mathcal{M}(P, G)$. In particular, it is immediate from the transformation $\mathcal{M}$ that for any program P and goal G, if $p \in open(P)$, then $p^{\mathbf{a}} \in open(\mathcal{M}(P, G))$. Note however that if P and Q are program modules, and G is a goal then $\mathcal{M}(P, G)$ and $\mathcal{M}(Q, G)$ may not be in general predicate disjoint, as they may share predicates with call pattern annotations. In the following we assume that $p^{\mathbf{c}} \in open(\mathcal{M}(P, Q))$ whenever $p \in open(P)$.

Abductive analysis can characterize the dependency between call patterns in $P \cup \{G\}$, and success patterns for the open predicates in P. The following proposition is an immediate application of Theorem 4.13.

PROPOSITION 6.3
Let P, Q be program modules such that Q extends P, G be a goal, and $\kappa \parallel b^{\mathbf{c}}$ be an abstract observation for $\mathcal{M}(P, G)$. If for each $ans \longleftarrow \tilde{\kappa} \parallel \bar{g} \in unf^{\mathcal{A}}(\{ans \longleftarrow b^{\mathbf{c}}\}, \mathcal{F}^{\mathcal{A}}(\alpha(\mathcal{M}(P, G))))$: $\mathcal{M}(Q, G) \models^{\tilde{\kappa}}_{var(b^{\mathbf{c}})} \kappa \otimes \tilde{\kappa} \parallel \bar{g}$, then $\mathcal{M}(P \cup Q, G) \models \kappa \parallel b^{\mathbf{c}}$.

PROOF. The proof is immediate by Theorem 4.13, and because for any program module P and Q and goal G: $\mathcal{M}(P, G) \cup \mathcal{M}(Q, G) = \mathcal{M}(P \cup Q, G)$. ∎

A consequence of Proposition 6.3 is that for any call pattern σ for b in $P \cup Q \cup \{G\}$: $\sigma(b) \in \gamma(\kappa)(b)$. This claim can be immediately proved by applying the equivalence between successful computations for $b^{\mathbf{c}}$ in $\mathcal{M}(P \cup Q, G)$ and call patterns for b in $P \cup Q \cup \{G\}$, as proved in [7].

We show abductive analysis for call patterns by a simple example

[6]In the following **a** and **c** annotations state respectively for *success* and *call*.

EXAMPLE 6.4 Consider the program P_{rev} in Example 5.3, and an initial call (goal) G for `rev` with first argument ground. $\alpha(\mathcal{M}(P_{rev}, G))$ with left-to-right selection rule is:

$$
\begin{array}{rcl}
rev^{\mathbf{c}}(x_1, x_2) & \longleftarrow & x_1 \parallel true \\
rev^{\mathbf{a}}(x_1, x_2) & \longleftarrow & x_1 \wedge x_2 \parallel rev^{\mathbf{c}}(x_1, x_2) \\
rev^{\mathbf{c}}(x_1, x_2) & \longleftarrow & x_3 \rightarrow x_1 \parallel rev^{\mathbf{c}}(x_3, x_4) \\
append^{\mathbf{c}}(x_1, x_2, x_3) & \longleftarrow & x_4 \leftrightarrow (x_2, \wedge x_6) \wedge x_1 \leftrightarrow x_7 \wedge x_3 \leftrightarrow x_5 \parallel rev^{\mathbf{c}}(x_4, x_5), rev^{\mathbf{a}}(x_6, x_7) \\
rev^{\mathbf{a}}(x_1, x_2) & \longleftarrow & x_1 \leftrightarrow x_3 \leftrightarrow (x_5 \wedge x_8) \wedge x_2 \leftrightarrow x_4 \leftrightarrow x_9 \wedge x_6 \leftrightarrow x_7 \parallel \\
 & & rev^{\mathbf{c}}(x_3, x_4), rev^{\mathbf{a}}(x_5, x_6), append^{\mathbf{a}}(x_7, x_8, x_9)
\end{array}
$$

Here, the abstract observation $x_1 \wedge x_2 \parallel append^{\mathbf{c}}(x_1, x_2, x_3)$ specifies that any call pattern for `append` in the left-to-right execution of G in P_{rev} must have the first two arguments ground. By abduction we can characterize the semantics of `append` (in this case $\texttt{append}^{\mathbf{a}}$) so that this observation is satisfied. The compositional abstract semantics $\mathcal{F}_T^{Pos}(\alpha(\mathcal{M}(P_{rev}, G)))$ contains the following clauses for `append` (it is worth noting that $open(\mathcal{M}(P_{rev}, G)) = \{append^{\mathbf{a}}, append^{\mathbf{c}}\}$):

$$
\begin{array}{rcl}
append^{\mathbf{c}}(x_1, x_2, x_3) & \longleftarrow & x_1 \wedge x_2 \\
append^{\mathbf{c}}(x_1, x_2, x_3) & \longleftarrow & x_2 \wedge x_8 \wedge x_7 \wedge x_1 \leftrightarrow x_9 \parallel append^{\mathbf{a}}(x_7, x_8, x_9)
\end{array}
$$

The abduced observation is $ans(x_1, x_2, x_3) \longleftarrow \kappa \parallel append^{\mathbf{a}}(x_7, x_8, x_9)$ where $\kappa = x_2 \wedge x_8 \wedge x_7 \wedge x_1 \leftrightarrow x_9$. Therefore, by Theorem 4.13, only success patterns κ' such that $(\kappa \wedge \kappa') \rightarrow (\kappa \wedge x_1 \wedge x_2)$ are allowed for $\texttt{append}^{\mathbf{a}}$, namely:

$$
\left\{
\begin{array}{l}
append^{\mathbf{a}}(x_7, x_8, x_9) \longleftarrow x_7 \wedge x_8 \wedge x_9 \\
append^{\mathbf{a}}(x_7, x_8, x_9) \longleftarrow x_9 \leftrightarrow (x_7 \wedge x_8) \\
append^{\mathbf{a}}(x_7, x_8, x_9) \longleftarrow x_9 \leftrightarrow x_7 \\
append^{\mathbf{a}}(x_7, x_8, x_9) \longleftarrow x_9 \leftrightarrow x_8
\end{array}
\right\}
$$

It is worth noting that both P_{app} and P'_{app} below satisfy this requirement.

P_{app}

```
append([],X,X).
append([X|W],Y,[X|Z]):-append(W,Y,Z).
```

P'_{app}

```
append([],[],[]).
append([],[X|Y],[X|Y]):-append([], Y, Y).
append([X|W],Y,[X|Z]):-append(W,Y,Z).
```

Therefore, any call pattern for `append` in $P_{rev} \cup P_{app}$ or $P_{rev} \cup P'_{app}$ with initial goal for `rev` having the first argument ground, will have the first two arguments ground.

6.3 Optimizing T-stability tests

Abductive analysis can be used in conjunction with $\star$-abstraction to reduce the search space for T-stability check. We sketch this idea in this section.

It is known that while T-stable semantics provides a compositionally complete abstract semantics (i.e. with no loss of precision), testing for T-stability is expensive. Indeed, this test is performed by searching over any possible definition (semantics) for open predicates. The search space is huge—often exponential in the number of arguments of open predicates. Here we propose an improvement by statically restricting (by abduction) the number of possible semantics that the open predicates can have, according to a given program specification. This leads to a notion of *relative T-stability*, where only a fixed number of possible patterns are allowed to be imported from open predicates. In this case, if P is a program module and $\mathcal{Q} \subseteq AInt$ contains interpretations having only unit clauses with predicates in $open(P)$, then $T^{\mathcal{A}}_{\alpha(P)}\uparrow n$ is *T-stable with respect to* $\mathcal{Q}$ iff for each $Q \in \mathcal{Q}$: $unf^{\mathcal{A}}(T^{\mathcal{A}}_{\alpha(P)}\uparrow n, Q) = unf^{\mathcal{A}}(T^{\mathcal{A}}_{\alpha(P)}\uparrow (n+1), Q)$. It is clear that a T-stable semantics $T^{\mathcal{A}}_{\alpha(P)}\uparrow n$ with respect to $\mathcal{Q}$ is compositional with respect to any program module which extends P and has semantics in $\mathcal{Q}$. Therefore, once an observation for P has been fixed, in order to have an useful T-stability notion, we need at least that $\mathcal{Q}$ contains the set of all possible extensions Q of P such that $P \cup Q$ satisfies the original observation. $\mathcal{Q}$ above can be for instance the result of the abductive analysis on P based on the less precise $\star$-abstraction. We show this by an example.

EXAMPLE 6.5 Consider the program module P_{rev} defined in Example 5.3 with `append` open, and $\mathcal{F}^{Dep}_{\star}(\alpha(P_{rev}))$. Assume the observation $G^a = x_1 \leftrightarrow x_2 \; [\!] \; rev(x_1, x_2)$ on P_{rev}, then

$$ans(x_1, x_2) \longleftarrow \tilde{\kappa} \; [\!] \; append(x_5, x_6, x_7) \in unf^{Dep}(\{ans \longleftarrow rev(x_1, x_2)\}, \mathcal{F}^{Dep}_{\star}(\alpha(P_{rev})))$$

where $\tilde{\kappa} = (x_1 \leftrightarrow x_6) \wedge x_7 \leftrightarrow (x_5 \wedge x_2)$. Denote $\mathcal{Q}$ the set of possible definitions for `append` that satisfy the observation, i.e.

$$\begin{array}{rcl} \mathcal{Q} & = & \left\{ \; Q \;\middle|\; Q \models^{\tilde{\kappa}}_{\{x_1, x_2\}} \tilde{\kappa} \wedge (x_1 \leftrightarrow x_2) \; [\!] \; append(x_5, x_6, x_7) \; \right\} \\ & = & \left\{ \; append(x_5, x_6, x_7) \longleftarrow \kappa \;\middle|\; \kappa \in \mathcal{K} \; \right\} \end{array}$$

where

$$\mathcal{K} = \left\{ \kappa \;\middle|\; \begin{array}{l} \kappa \in Dep_{\{x_5, x_6, x_7\}} \\ (\kappa \wedge \tilde{\kappa}) \;\rightarrow\; (\tilde{\kappa} \wedge (x_1 \leftrightarrow x_2)) \end{array} \right\} = \left\{ \begin{array}{c} x_6 \wedge x_7 \;,\; x_5 \wedge x_6 \wedge x_7 \\ x_5 \wedge (x_7 \leftrightarrow x_6) \\ x_7 \wedge (x_5 \leftrightarrow x_6) \end{array} \right\}$$

It is worth noting that $|\mathcal{K}| < |Dep_{\{x_5, x_6, x_7\}}|$. $\mathcal{Q}$ provides a restricted search space for possible definitions of `append` all leading to a definition of `rev` where the first argument is ground iff the second is. A T-stable compositional semantics for P_{rev} can be efficiently computed relatively to $\mathcal{Q}$. It is compositional complete with respect to any of the definitions of `append` in $\mathcal{Q}$. However, note that because of compositional correctness of $\star$-abstraction, (i.e., because of the application of Theorem 4.13), $\mathcal{Q}$ does not provide (in general) a complete characterization of extensions for P_{rev} that satisfy G^a.

7 Conclusions

We believe that the main application field of abductive analysis is dataflow analysis of module interaction. This is obtained by statically deriving, by abduction, a set of constraints on each single module, given a constraint on their composition. Constraints are goals, and therefore the same engine for deductive analysis (i.e. unfolding) can be used for the abductive analysis. The implementation costs of abductive analysis are therefore reduced. Of course, the use of the *Bossi semantics* in [5] as a base semantics for abduction is only one of the possibilities provided in the literature. Different frameworks for abductive analysis can be defined by modifying abductive logic programming (e.g. [27]) to cope with abstract interpretation. This may provide an interesting alternative approach for abductive analysis by *abstract compilation* ([24]). In a recent work, Giacobazzi and Ranzato [22] proved that most well known compositional semantics for logic programs, like the semantics in [5], can be obtained systematically by abstraction from a given operational semantics of logic programs. The idea is that any semantics of logc programs (and more in general of a programming language, see [13]), can be viewed as an abstraction of a fixed, more concrete, operational semantics which is basically SLD resolution (e.g. see [20]). Therefore, composing abstractions corresponds to compose semantics. In [22] the authors proved that compositionality for logic program semantics can be systematically achieved by composing, by a suitable *functional combination* operator, the abstractions corresponding to more abstract, possibly non compositional semantics. The resulting framework is powerful enough to encode well known compositional semantics for logic programming as functional composition of simpler non-compositional ones, and provide an interesting link between semantics and well known function spaces. For instance, the *Bossi semantics* in [5] can be viewed as the semantics of the functional composition of the standard non-compositional s semantics for *computed answer substitutions* in [16]. Moreover, other more concrete semantics, like the semantics for *computed answers*, *call patterns*, *partial answers* etc. can be all functionally composed in the same way, yet providing new compositional semantics for logic programs. It is worth noting that in this framework, any semantics which can be obtained by functional composition of simpler ones can also be used as a basis for different forms of abductive analysis.

Acknowledgments

The stimulating discussions with Mike Codish, Patrick Cousot, Saumya Debray and Giorgio Levi are gratefully acknowledged. This work has been done while the author was visiting the *LIX, Laboratoire d'Informatique, École Polytechnique*, Paris. He thanks *LIX*, and in particular Radhia Cousot, for the kind hospitality. This work has been partly supported by the EEC *Human Capital and Mobility* individual grant: "Semantic Definitions, Abstract Interpretation and Constraint Reasoning", N. ERBCHBICT930822.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: algebraic properties and efficient representation. In *Proc. of SAS '94*, LNCS 864, pp. 266–280, 1994.

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

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

[4] G. Birkhoff. *Lattice Theory.* AMS Colloq. Public. vol. XXV, 3rd ed., 1967.

[5] 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.

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

[7] M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. *Theoretical Computer Science*, 124(1):93–126, 1994.

[8] M. Codish, S. K. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Proc. Twentieth Annual ACM Symp. Principles of Programming Languages, POPL '93*, pages 451–464. ACM Press, 1993.

[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*, pages 331– 345. The MIT Press, Cambridge, Mass., 1991.

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

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Fourth Annual ACM Annual ACM Symp. Principles of Programming Languages, POPL '77*, pp. 238–252, 1977.

[12] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth Annual ACM Symp. Principles of Programming Languages, POPL '79*, pages 269–282, 1979.

[13] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages (POPL '92)*, pages 83–94. ACM Press, 1992.

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

[15] 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.

[16] 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.

[17] 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. Also available as Technical Report LIX – Ecole Polytechnique LIX/RR/94/04, URL `http://www.di.unipi.it/~giaco/papers.html`.

[18] R. Giacobazzi. *Semantic Aspects of Logic Program Analysis.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1993. TD – 18/93. URL `http://www.di.unipi.it/~giaco/papers.html`.

[19] R. Giacobazzi. *Abductive Analysis of Modular Logic Programs.* In M. Bruynooghe, editor, *Proc. of the Eleventh International Logic Programming Symposium, ILPS '94*, pages 377–391. The MIT Press, Cambridge, Mass., 1994.

[20] R. Giacobazzi. "Optimal" collecting semantics for analysis in a hierarchy of logic program semantics. In C. Puech, editor, *Proceedings of the 13th International Symposium on Theoretical Aspects of Computer Science (STACS '96)*, LNCS 1046, pages 503–514. Springer-Verlag, 1996.

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

[22] 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.

[23] C.A.R. Hoare, I.J. Hayes, H. Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Comm. of the ACM*, 30(8):672–686, 1987.

[24] M. Hermenegildo, R. Warren, and S.K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(4):349–366, 1992.

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

[26] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages, POPL '87*, pages 111–119. ACM, 1987.

[27] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

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

[29] K. Marriott and H. Søndergaard. Bottom-up Dataflow Analysis of Normal Logic Programs. *Journal of Logic Programming*, 13(2 & 3):181–204, 1992.

[30] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27–47, 1988.

A Appendix: Proofs of related results

The appendix contains the proofs of selected lemmata and theorems which are essential for deductive and abductive analysis.

Theorem 3.3 *[Codsh et al., [8]]*
Let $(Int, \alpha, AInt, \gamma)$ be a Galois insertion, and $\mathcal{F}^{\mathcal{A}} : AInt \rightarrow AInt$ be a monotonic and safe approximation of $\mathcal{F}$. Then, for any program module $P_1, P_2 \in Int$:

$$\alpha(\mathcal{F}(P_1 \cup P_2)) \sqsubseteq \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P_1)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(P_1)))$$

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

$$\begin{array}{rcl}
\alpha(\mathcal{F}(P_1 \cup P_2)) & = & \alpha(\mathcal{F}(\mathcal{F}(P_1) \cup \mathcal{F}(P_2))) \\
& & \qquad [\quad \text{Theorem 3.1} \quad] \\
& \sqsubseteq & \mathcal{F}^{\mathcal{A}}(\alpha(\mathcal{F}(P_1) \cup \mathcal{F}(P_2))) \\
& & \qquad [\quad \text{safety} \quad] \\
& = & \mathcal{F}^{\mathcal{A}}(\alpha(\mathcal{F}(P_1)) \sqcup \alpha(\mathcal{F}(P_2))) \\
& & \qquad [\quad \alpha \text{ additivity} \quad] \\
& \sqsubseteq & \mathcal{F}^{\mathcal{A}}(\mathcal{F}^{\mathcal{A}}(\alpha(P_1)) \sqcup \mathcal{F}^{\mathcal{A}}(\alpha(P_2))) \\
& & \qquad [\quad \text{safety and monotonicity} \quad]
\end{array}$$

∎

Lemma 3.6 *[Giacobazzi, [18]]*
For any abstract constraint system $(\mathcal{A}, \otimes)$, $unf^{\mathcal{A}}$ is left-additive.

PROOF. Left-additivity follows immediately by definition. ∎

Lemma 3.7 *[Giacobazzi, [18]]*
If $P_1^a \cong Q_1^a$ and $P_2^a \cong Q_2^a$ then $unf^{\mathcal{A}}(P_1^a, P_2^a) \cong unf^{\mathcal{A}}(Q_1^a, Q_2^a)$.

PROOF. Because γ is (by definition) an additive extension of $\gamma_{\mathcal{A}}$ on abstract clauses, and $unf^{\mathcal{A}}$ is left-additive, we can prove the lemma for the more simple case where P_1^a and Q_1^a are singletons. We assume, without loss of generality, that the clauses in P_1^a, Q_1^a, P_2^a and Q_2^a do not share common variables. Let $P_1^a = \{c\}$ and $Q_1^a = \{c'\}$, where $c = h \longleftarrow \kappa \parallel g_1, \ldots, g_n$ and $c' = h' \longleftarrow \kappa' \parallel g_1', \ldots, g_n'$. Then

$$h \longleftarrow \sigma \parallel \bar{b}_1 :: \cdots :: \bar{b}_n \in \gamma(unf^{\mathcal{A}}(P_1^a, P_2^a))$$

iff

$$\begin{array}{l}
\langle h_i \longleftarrow \kappa_i \parallel \bar{b}_i \rangle_{i=1}^n \ll_c P_2^a \\
\sigma \in \gamma_{\mathcal{A}}(\kappa \otimes \bigotimes_{i=1}^{n} (\kappa_i \otimes \alpha_{\mathcal{A}}(g_i = h_i)))
\end{array}$$

Because the concretization is always a co-additive function in any Galois connection, the condition above is satisfied iff

$$\begin{array}{l} \langle h_i \longleftarrow \kappa_i \parallel \bar{b}_i \rangle_{i=1}^n \ll_c P_2^a \\ \sigma \in \gamma_{\mathcal{A}}(\kappa) \wedge \wedge_{i=1}^n (\gamma_{\mathcal{A}}(\kappa_i) \wedge \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(g_i = h_i))) \end{array}$$

By definition:

$$\sigma = \sigma_\kappa \quad \wedge \quad \wedge_{i=1}^n (\sigma_{\kappa_i} \wedge \sigma_=)$$

for some $\sigma_\kappa \in \gamma_{\mathcal{A}}(\kappa)$, $\sigma_{\kappa_i} \in \gamma_{\mathcal{A}}(\kappa_i)$ and $\sigma_= \in \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(g_i = h_i))$.

Because we assume that P_1^a, Q_1^a, P_2^a and Q_2^a contain renamed apart clauses, and $P_1^a \cong Q_1^a$ and $P_2^a \cong Q_2^a$, then the condition above implies that there exist:

$$\begin{array}{l} \langle h'_i \longleftarrow \kappa'_i \parallel \bar{b'}_i \rangle_{i=1}^n \ll_{c'} Q_2^a \\ \delta_{\kappa'} \in \gamma_{\mathcal{A}}(\kappa') \\ \delta_{\kappa'_i} \in \gamma_{\mathcal{A}}(\kappa'_i) \\ \delta_= \in \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(g'_i = h'_i)) \end{array}$$

such that if $\delta = \delta_{\kappa'} \quad \wedge \quad \wedge_{i=1}^n (\delta_{\kappa'_i} \wedge \delta_=)$ then

$$h \longleftarrow \sigma \parallel \bar{b}_1 :: \cdots :: \bar{b}_n \sim h' \longleftarrow \delta \parallel \bar{b'}_1 :: \cdots :: \bar{b'}_n$$

Note that

$$h' \longleftarrow \delta \parallel \bar{b'}_1 :: \cdots :: \bar{b'}_n \in \gamma(\mathit{unf}^{\mathcal{A}}(Q_1^a, Q_2^a)).$$

This concludes the proof, by proving that $\gamma(\mathit{unf}^{\mathcal{A}}(P_1^a, P_2^a)) \subseteq \gamma(\mathit{unf}^{\mathcal{A}}(Q_1^a, Q_2^a))$. The other inclusion is similar. ∎

Lemma 3.8 *[Giacobazzi, [18]]*
For any abstract constraint system $(\mathcal{A}, \otimes)$, $\mathit{unf}^{\mathcal{A}}$ is associative and right-continuous.

Proof. The proof of continuity for the function $\lambda I^a.\mathit{unf}^{\mathcal{A}}(P^a, I^a)$ for any $P^a \in \mathit{AInt}$ is not immediate, as AInt is not in general noetherian, even if $\mathcal{A}$ is noetherian. However, the proof can be based on the finiteness property of abstract constraint systems (for the more general case of noetherian or non-noetherian abstract constraint systems see [21]).

Let $D \subseteq \mathit{AInt}$ be a (possibly infinite) chain and let $h \longleftarrow \kappa' \parallel \bar{b}_1, \ldots, \bar{b}_n \in \mathit{unf}^{\mathcal{A}}(P^a, \sqcup D)$. Then, by the definition there exist $c = \; h \longleftarrow \kappa \parallel g_1, \ldots, g_n \ll P^a$ and $\langle h_i \longleftarrow \kappa_i \parallel \bar{b}_i \rangle_{i=1}^n <$ $<_c \sqcup D$ such that $\kappa' = \kappa \quad \otimes \quad \otimes_{i=1}^n (\kappa_i \otimes \alpha_{\mathcal{A}}(g_i = h_i))$. Because $\mathcal{A}$ is noetherian, for each $i = 1, \ldots, n$ there exists $I_i \in D$ such that $h_i \longleftarrow \kappa_i \parallel \bar{b}_i \ll_c I_i$. The rest of this proof is standard.

The proof of associativity follows from the basic properties of $\otimes$ (i.e. associativity and commutativity). Let $P^a, Q^a, W^a \in \mathit{AInt}$, we prove that

$$\mathit{unf}^{\mathcal{A}}(\mathit{unf}^{\mathcal{A}}(P^a, Q^a), W^a) = \mathit{unf}^{\mathcal{A}}(P^a, \mathit{unf}^{\mathcal{A}}(Q^a, W^a)).$$

By the definition:

$$c \in \mathit{unf}^{\mathcal{A}}(\mathit{unf}^{\mathcal{A}}(P^a, Q^a), W^a)$$

iff $c = h \longleftarrow \kappa \parallel \bar{g}_{1_1} :: \ldots :: \bar{g}_{nm_i}$ where

$$\begin{array}{l} h \longleftarrow \kappa_P \parallel a_1, \ldots, a_n \in P^a \\ a'_i \longleftarrow \kappa_{Q_i} \parallel b_{i_1}, \ldots, b_{im_i} \in Q^a \quad \text{for } i = 1, \ldots, n \\ b'_{i_j} \longleftarrow \kappa_{W_{i_j}} \parallel \bar{g}_{i_j} \in W^a \quad \text{for } i = 1, \ldots, n \text{ and } j = 1, \ldots, m_i \end{array}$$

are clauses which share no variable, and by defining $\kappa' = \kappa_{Q_1} \otimes \ldots \otimes \kappa_{Q_n}$ and $\kappa'' = \kappa_{W_{1_1}} \otimes \ldots \otimes \kappa_{W_{nm_n}}$, we have:

$$\kappa = \kappa_P \otimes \kappa' \otimes \bigotimes_{i=1}^{n} \alpha_{\mathcal{A}}(a_i = a'_i) \otimes \kappa'' \otimes \bigotimes_{i=1}^{n} \bigotimes_{j=1}^{m_i} \alpha_{\mathcal{A}}(b_{i_j} = b'_{i_j}).$$

Analogously: $c \in unf^{\mathcal{A}}(P^a, unf^{\mathcal{A}}(Q^a, W^a))$ iff $c = h \longleftarrow \tilde{\kappa} \parallel \bar{g}_{1_1} :: \ldots :: \bar{g}_{nm_i}$ where

$$\begin{array}{l} h \longleftarrow \kappa_P \parallel a_1, \ldots, a_n \in P^a \\ a'_i \longleftarrow \kappa_{Q_i} \parallel b_{i_1}, \ldots, b_{im_i} \in Q^a \quad \text{for } i = 1, \ldots, n \\ b'_{i_j} \longleftarrow \kappa_{W_{i_j}} \parallel \bar{g}_{i_j} \in W^a \quad \text{for } i = 1, \ldots, n \text{ and } j = 1, \ldots, m_i \end{array}$$

are clauses which share no variable and

$$\tilde{\kappa} = \kappa_P \otimes \kappa' \otimes \kappa'' \otimes \bigotimes_{i=1}^{n} \bigotimes_{j=1}^{m_i} \alpha_{\mathcal{A}}(b_{i_j} = b'_{i_j}) \otimes \bigotimes_{i=1}^{n} \alpha_{\mathcal{A}}(a_i = a'_i).$$

In both cases, we can choose without loss of generality the same variables for corresponding clauses. Thus, by $\otimes$ commutativity and associativity, $\tilde{\kappa} = \kappa$, which concludes the proof. ∎

Lemma 3.9 *[Giacobazzi, [18]]*
Let $(\mathcal{A}, \otimes)$ be an abstract constraint system. Then, for every $I_1, I_2 \in Int$,

$$\alpha(unf(I_1, I_2)) \sqsubseteq unf^{\mathcal{A}}(\alpha(I_1), \alpha(I_2))$$

Proof. It is equivalent to prove that:

$$unf(\gamma(I_1^a), \gamma(I_2^a)) \subseteq \gamma(unf^{\mathcal{A}}(I_1^a, I_2^a))$$

Assume $h \longleftarrow \sigma \parallel \bar{b}_1 :: \ldots :: \bar{b}_n \in unf(\gamma(I_1^a), \gamma(I_2^a))$. By the definition this is true iff

$$\begin{array}{l} c = h \longleftarrow \vartheta \parallel g_1, \ldots, g_n \in \gamma(I_1^a) \\ \langle h_i \longleftarrow \vartheta_i \parallel \bar{b}_i \rangle_{i=1}^n \ll_c \gamma(I_2^a) \\ \sigma = \vartheta \ \wedge \ \bigwedge_{i=1}^{n} (\vartheta_i \ \wedge \ g_i = h_i) \ \text{ and } \ \mathcal{H} \models \sigma \end{array}$$

By γ definition, there exist $h \longleftarrow \kappa \parallel g_1, \ldots, g_n \in I_1^a$, such that $\vartheta \in \gamma_{\mathcal{A}}(\kappa)$ and $\langle h_i \longleftarrow \kappa_i \parallel \bar{b}_i \rangle_{i=1}^n \ll_c I_2^a$ such that for each $i = 1, \ldots, n$: $\vartheta_i \in \gamma_{\mathcal{A}}(\kappa_i)$.
By $\otimes$ correctness and because $(\mathcal{C}, \alpha_{\mathcal{A}}, \mathcal{A}, \gamma_{\mathcal{A}})$ is a Galois insertion:

$$\vartheta \ \wedge \ \bigwedge_{i=1}^{n} (\vartheta_i \ \wedge \ g_i = h_i) \in \gamma_{\mathcal{A}}\left(\kappa \ \otimes \ \bigotimes_{i=1}^{n} (\kappa_i \ \otimes \ \alpha_{\mathcal{A}}(g_i = h_i))\right)$$

This concludes the proof. ∎