Abstract interpretation for uniform proof systems

Paolo Volpe
Dipartimento di Matematica e Applicazioni
Universitá "Federico II"
Via Cintia, 80126 Napoli, Italy
email: volpep@di.unipi.it

Abstract

In this work we propose a framework for the static analysis of logic programming languages founded on the uniform proofs approach. It is based on the abstract interpretation approach and exploits the proof-theoretic features of the considered languages.
In particular we propose two systems of inductive definitions to model the bottom-up and the top-down constructions of proof trees in a sequent system. In this work we analyze a subset of λ-Prolog.
We then interpret the inference rules on an abstract domain, in which a possible less precise but however correct description of computations is given. In the abstract domain, the meaning of a program is still characterized by sets of inference rules and by the induced operators. Anyway the abstract operators we define this way are the most precise approximations of the concrete ones.
Finally we devise some conditions on the abstraction function that allow to restate on the abstract domain the equivalence of the bottom-up and top-down approaches and the compositionality properties that characterize concrete semantics.

1 Introduction

Recently there have been many proposals for new logic programming languages [1, 12, 16, 15, 11, 18]. The common ground of such extensions is traditional proof theory based on sequent calculi, in which an adequate analysis of the primitives of logic languages and of proof search can be done.

The common model is that of *uniform proofs*[16]. The proof search is led by the goal formulas (by which the slogan "goal-driven proof search"), the program formulas are considered when the goal formula becomes atomic. At this point one clause is chosen, instantiated and reduced. The non-determinism of the search is relegated essentially in this choice and instantiation of program formulas.
A very important advantage is the fact that the definition of uniform proof makes sense

for every logic with a sequent calculus that enjoy cut elimination. This means that the logical system in which an Abstract Logic Programming Language is found can be different from the classical framework of Horn Clause Logic (HCL). This suggests the idea to shift to other logics to found declaratively constructs, such as mechanisms for the managing of modules, for abstraction, or supports for concurrence, so as to treat them homogeneously with the rest of the language. Indeed in [16] the basic logical system is no more HCL but the Intuitionist theory of hereditary higher-order Harrop formulas, in which a natural notion of module, of abstraction and higher-order terms can be found, without resorting to extralogical primitives [14]. Based on this idea, many works have recast logic programming in various alternative logics (modal [11], intuitionistic linear [12] and classical linear [1, 15, 18]). The research on efficient implementations of such languages is an active field[3]. It has to face novel constructs to which exsisting, WAM-based, techiques do not easily apply. Think for example to higher-order terms, dynamic modules and private names of λ-Prolog or the managing of communication and multiple head clauses of *LO* or Forum. Indeed already per se the classical WAM based compilation have problems as regards the efficiency of the produced code [17]. In fact not always the compiler can reduce the full generality of the proof search. To optimize, it is necessary to know some properties of the proofs a certain program and its goals can arise. Indeed such analysis must be done formally and possibly efficiently and this obviously conflicts with the general undecidability of properties of programs. A practical framework for static analysis of programs has revealed abstract interpretation [5, 6]. It is a semantics-based technique founded on the idea of relaxing the precision of semantic descriptions to obtain correct approximations of the results. In logic programming (having a simpler semantics) there have been many works on the subject that have established the benefits of the approach. Indeed static analysis of programs can be very useful to devise new compiling techniques and optimization algorithms [17].
We think that the interest in a work aimed to study the abstract interpretation of proof theoretic based logic programming is twofold. In the field of extended logic programming languages, it addresses the problem of devising the tools to make more efficient their implementations. As regards the abstract interpretation theory point of view, it gives the possibility to extend existent results and to face problems such as dynamic linking of modules, higher order terms, managing of private names, concurrency, communication and interaction. We think it is interesting to discover whether or not the declarative character of logic based languages can ease the task.

2 Preliminaries

We recall some standard concept that will be widely used throughout the paper. An *inference rule system* ([8]) is defined as a tuple $(\mathcal{U}, \Pi, \bot, \sqcup)$, with the ***universe*** $\mathcal{U}$, a set; Π a set of *rules instances* $\frac{P}{c}$, with $P \subseteq \mathcal{U}$ and $c \in \mathcal{U}$; $\bot \subseteq U$, the ***basis***; $\sqcup : \mathcal{P}(\mathcal{P}(\mathcal{U})) \to \mathcal{P}(\mathcal{U})$ the *join*. Inference rule systems single out (if they exist at all) subsets of $\mathcal{U}$ *closed* with respect to the inference rules, that is $P \subseteq X$ implies $c \in X$. The *minimal* closed set is the least closed set greater then $\bot$.

The ***sequents*** we will use are expressions $\Gamma \vdash G$, where Γ is a set of formulas and G a formula of a *logic language* (for a better account see [10]). Given a ***sequent calculus***, that is a set of inference rule for sequents, we single out *derivation trees*: each node is a sequent and, with its son(s), gives an instance of an inference rule. The sequent at the root is the *conclusion sequent*. We will use $\underline{\pi}$ for the conclusion sequent of π. A *proof tree* is a derivation tree with only *axioms* at the leaves. Throughout the paper we will use also the term *partial proof tree* to refer to derivation trees. Given π a partial proof tree, $\overline{\pi}$ will denote the multiset of sequent at leaves that are not axioms. $\overline{\pi}_{|_\ell}$ denotes the non axiom leaf ℓ. The notation $\pi_1[\pi_2/\ell]$ will denote the tree obtained from π_1 by substituting the leaf ℓ with the tree π_2 ($\overline{\pi}_{|_\ell}$ and $\underline{\pi_2}$ must be equal). If π_2 is an instance of an inference rule, we will use the notation $\pi_1[\star/\ell]$ where $\star$ is the name of the rule. The subtree relation (same root) induce a partial order on (partial) proof trees.
Substitution (see [2] for a more complete account) are function from variables to terms, that differ from the identity just on a finite set. We will use the representation as a set $\{x_1 \to t_1, \ldots, x_n \to t_n\}$. The *composition* is easily defined as the composition of function. A *preorder* on substitution can be defined by: $\theta \leq \eta$ iff exists δ such that $\theta\delta = \eta$. The result of the application of a substitution θ to an expression E, is defined as the expression obtained from E by substituting each free variable X with $\theta(X)$. The expression E can be a term, a formula, a derivation tree, with its own notions of *free* and *bound* variable. The set of variables an which θ is different from the identity is $dom(\theta)$, the set of variables that appear in terms on which variables are mapped by θ is $rng(\theta)$, $var(\theta) = dom(\theta) \cup rng(\theta)$. All substitutions we will use verify $dom(\theta) \cap rng(\theta) = \emptyset$. Given expression E, $\theta_{|_E}$ will denote the restriction of θ to free variables in E. Two expressions E_1 and E_2 are ***unifiable*** if it exists θ such that $E_1\theta = E_2\theta$. Fixed a signature and a set of variables, we single out the set *Subst* of substitution from variables in terms of the given signature. $\mathcal{A}$ will denote the set of all atomic formulas $p(X_1, \ldots, X_n)$, with $X_1, \ldots, X_n$ distinct variables.

3 Bottom-up and top-down semantics for a subset of λ-Prolog

Our basic commitment is to start with a very concrete semantics for our computations. We think that a very basic account of the computations of a logic language can be given in terms of the proof trees of the sequent calculus that define it. Indeed we think that all implementation-independent observables of computations can easily be seen as abstraction of proof trees. Other conceivable observables depends on the particular way the proof trees construction is carried on. For example, the choice of visiting a tree depth first can be seen as an implementation choice. We think however that an implementation-independent semantics analysis can be still useful. First of all, since at this level we have a good compromise between the complexity of the domain (more concrete domains are more complex) and the precision of the analysis. Besides, this formalization could be useful also for real implementations. The results of the analysis can be imprecise but are still "correct".

Here and in the following we will consider the sequent system Σ (see Fig. 1). It corresponds to a first order version of λ-Prolog [14], that allows the dynamic loading of modules, through the ($\supset$) rule, and the managing of private names (useful for abstraction) through the ($\forall$) rule. For simplicity we suppose that dynamic modules have not free variables.
Already in this basic case we have to front two non-standard features of logic programming. However we want basically to fix the framework, the study of richer sequent system is deferred to future works.
The system Σ individuate a set of proof trees that the interpreter have to visit.

- Let *goal formulas* G be specified as

$$G = \top \mid A \mid G \wedge G \mid G \vee G \mid D \supset G \mid \exists x G \mid \forall x G$$

- Let *program clauses* D be specified as

$$D = \forall (G \supset A)$$

- Let Γ, M be a set of program clauses, G, G_1, G_2 be goal formulas

$$\frac{}{\Gamma \vdash \top}(\top) \qquad \frac{\Gamma \vdash G_1}{\Gamma \vdash G_1 \vee G_2}(\vee_1, G_2) \qquad \frac{\Gamma \vdash G_2}{\Gamma \vdash G_1 \vee G_2}(\vee_2, G_1)$$

$$\frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \wedge G_2}(\wedge) \qquad \frac{\Gamma, M \vdash G}{\Gamma \vdash M \supset G}(\supset, M)$$

$$\frac{\Gamma \vdash G[t/x]}{\Gamma \vdash \exists x G}(\exists x) \qquad \frac{\Gamma \vdash G}{\Gamma \vdash \forall x G}(\forall x) \qquad \frac{\Gamma \vdash G}{\Gamma \vdash A}(\bowtie)$$

The $\forall x$ rule has the proviso that x is not free in the lower sequent.
The $\bowtie$ rule has the proviso that $G \supset A$ is an instance of a clause in Γ.

Figure 1: The sequent calculus Σ

Indeed a very important element of the search is not explicitly represented in Σ: the substitutions for the free variable of the trees. In real terms a computation in a logic language is the contemporary search for a proof tree and for a substitution for the free variables of the root sequent. The real proof tree we are searching at a certain step is the tree $\pi\theta$, with θ a substitution for the free variable in π. Following [9], instances of proofs of atoms in $\mathcal{A}$ will play a relevant role. For the proviso on the ($\forall$) rule, not all substitutions are allowed.
Fixed the set $\mathcal{D}$ of partial runs, throughout the paper we will work on sets of partial runs, that is on members of the domain $(\mathcal{P}(\mathcal{D}), \subseteq)$.

3.1 Bottom-up characterization

The proof trees building can easily be given a bottom-up characterization. We single out a rule inference system $\mathcal{B}_\Sigma$ (see Fig. 2) that models the bottom-up construction of partial proof trees of the system Σ. The rules are directly connected to the rules of the proof system Σ, taking into account the substitutions. This is visible especially in the rules for quantifiers.

Let π, π_1, π_2 be proof trees, Γ, M be sets of clauses and G, G' be goal formulas.

- Axioms

$$\frac{\dfrac{}{}(\top)}{\Gamma \vdash \top}(\top) \qquad \frac{\dfrac{}{}(na)}{\Gamma \vdash G}$$

The rule (na) introduces a non-axiom leaf.

- Connectives

$$\frac{\dfrac{\pi}{}}{\dfrac{\pi}{\Sigma}(\vee_1, G)}(\vee_1, G) \qquad \frac{\dfrac{\pi}{}}{\dfrac{\pi}{\Sigma}(\vee_2, G)}(\vee_2, G)$$

$$\frac{\dfrac{\pi}{}}{\dfrac{\pi}{\Sigma}(\supset, M)}(\supset, M) \qquad \frac{\dfrac{\pi_1 \quad \pi_2}{}}{\dfrac{\pi_1 \quad \pi_2}{\Sigma}(\wedge)}(\wedge)$$

- Quantifiers

$$\frac{\dfrac{\pi}{}}{\dfrac{\pi}{\Sigma}(\exists x)}(\exists x) \qquad \frac{\dfrac{\pi}{}}{\dfrac{\pi}{\Sigma}(\forall x)}(\forall x) \qquad \text{if } x \text{ not free in } \Sigma.$$

- Clause application

$$\frac{\dfrac{\dfrac{\pi}{\Gamma \vdash G'}}{\pi\eta}}{\dfrac{\Gamma \vdash G'\eta}{\Gamma \vdash A\eta}(\bowtie)}(\bowtie) \qquad \text{if } \forall (G \supset A) \in \Gamma, \eta = \text{ mgu}(G', G), \eta \text{ feasible for } \pi.$$

Figure 2: The system $\mathcal{B}_\Sigma$

This system naturally induces the function Φ so defined

$$\Phi(X) = \left\{ \pi \mid \exists \frac{P}{\pi} \in \mathcal{B}_\Sigma \;\; P \subseteq X \;\right\}$$

It collects all the partial runs that can be built from trees in X.
The function is monotonous and continuous on the domain $(\mathcal{P}(\mathcal{D}), \subseteq)$. The fixpoint exists and is equal to

$$|\mathcal{B}_\Sigma| = \bigcup_{k \geq 0} \Phi^k$$

The set $|\mathcal{B}_\Sigma|$ contains all (partial) trees of the system Σ. From the iterates of Φ we can extract the partial proofs that can arise from a program P with an atomic goal. In fact we have

$$\mathcal{B}[\![P]\!]^k = P \cdot \Phi^k$$
$$\text{where } P \cdot D = \left\{ \pi \in D \;\middle|\; \pi = \frac{\pi'}{P \vdash A'}, A \text{ atomic} \right\}$$

for the set of proof trees of P with an atomic goal built at the k-th step. Finally we can associate to a program P the set

$$\mathcal{F}[\![P]\!] = \bigcup_{k \geq 0} \mathcal{B}[\![P]\!]^k$$

of all (partial) proof trees of atoms from program P in the system Σ.

3.2 Top-down characterization

A different characterization can be given by mimicking the traditional top-down proof search strategy of the interpreter given a program P and a goal G. Again we build an inference system $\mathcal{T}_\Sigma$ (see Fig. 3).

Let π, π' be proof trees, ℓ be a non-axiom leaf in a proof tree and Γ, M be sets of clauses.

$$\frac{}{\Gamma \vdash \top}(\top)$$

$$\frac{\pi}{\pi[\vee_1/\ell]}(\vee_1) \qquad \frac{\pi}{\pi[\vee_2/\ell]}(\vee_2)$$

The rules have the proviso that $\Gamma \vdash G_1 \vee G_2 = \overline{\pi}_{|\ell}$ for a suitable ℓ.

$$\frac{\pi}{\pi[\wedge/\ell]}(\wedge) \qquad \exists \ell : \overline{\pi}_{|\ell} = \Gamma \vdash G_1 \wedge G_2$$

$$\frac{\pi}{\pi[\supset/\ell]}(\supset) \qquad \exists \ell : \overline{\pi}_{|\ell} = \Gamma \vdash M \supset G$$

$$\frac{\pi}{\pi[\exists' x/\ell]}(\exists x) \qquad \exists \ell : \overline{\pi}_{|\ell} = \Gamma \vdash \exists x G$$

the rule $(\exists x)$ expands the leaf ℓ using the rule $\dfrac{\Gamma \vdash G[y/x]}{\Gamma \vdash \exists x G}(\exists' x)$, with $y \notin var(\pi)$.

$$\frac{\pi}{\pi[\forall x/\ell]}(\forall x) \qquad \exists \ell : \overline{\pi}_{|\ell} = \Gamma \vdash \forall x G$$

$$\frac{\pi}{\pi[\bowtie/\ell]\eta}(\bowtie) \qquad \exists \ell : \overline{\pi}_{|\ell} = \Gamma \vdash A,\ G' \supset A' \in \Gamma,\ \eta = \text{mgu}(A, A'), \eta \text{ feasible for } \pi.$$

Figure 3: The system $\mathcal{T}_\Sigma$

This time the trees increase at the leaves. We can single out a function Ψ associated to the system

$$\Psi(X) = \left\{ \pi \;\middle|\; \exists \tfrac{P}{\pi} \in \mathcal{T}_\Sigma \;\; P \subseteq X \right\}$$

The function is monotonous and continuous on the domain $(\mathcal{P}(\mathcal{D}), \subseteq)$. We can define the *computations* of a program P and a goal G by

$$\mathcal{C}[\![P \vdash G]\!] = \bigcup_{k \geq 0} \Psi^k(\{\frac{}{P \vdash G}\})$$

that is the set of partial trees with $P \vdash G$ as conclusion sequent. For each P we can establish as its operational semantics the set of all computations starting from an atomic goal, that is

$$\mathcal{O}[\![P]\!] = \bigcup_{A \in \mathcal{A}} \mathcal{C}[\![P \vdash A]\!]$$

It is the set of all (partial) trees of the system Σ that have a sequent $P \vdash A$ as a conclusion.

The semantics of computations is compositional with respect to the operators of the language as the following theorem shows.

Theorem 1

$$\begin{aligned}
\mathcal{C}[\![P \vdash A]\!] &= A \cdot \mathcal{O}[\![P]\!];\\
\mathcal{C}[\![P \vdash A \wedge B]\!] &= \mathcal{C}[\![P \vdash A]\!] \wedge \mathcal{C}[\![P \vdash B]\!];\\
\mathcal{C}[\![P \vdash A \vee B]\!] &= \mathcal{L}(\mathcal{C}[\![P \vdash A]\!], B) \cup \mathcal{R}(\mathcal{C}[\![P \vdash B]\!], A);\\
\mathcal{C}[\![P \vdash M \supset A]\!] &= Impl(\mathcal{C}[\![P, M \vdash A]\!]);\\
\mathcal{C}[\![P \vdash \exists x A]\!] &= \exists x(\mathcal{C}[\![P \vdash A]\!]);\\
\mathcal{C}[\![P \vdash \forall x A]\!] &= \forall x(\mathcal{C}[\![P \vdash A]\!]).
\end{aligned}$$

The semantic operators that appear in the theorem are derived from the rules of the system $\mathcal{B}_\Sigma$ as shown in Fig. 4.

As regards the links between the two semantics, we have the complete equivalence, as shown by the following theorem.

Theorem 2

$$\mathcal{O}[\![P]\!] = \mathcal{F}[\![P]\!]$$

In conclusion we have a well-behaved concrete semantics that is compositional with respect to the principal operators of the language.

4 An abstract interpretation framework

The semantic framework showed in the previous section can be used as a base for abstraction, by substituting the domain $\mathcal{D}$ of partial trees with more abstract descriptions of computations. If we assume some adequate relations between the

$$
\begin{aligned}
A \cdot D &= \left\{ \pi\delta \;\middle|\; \pi \in D, \pi = \frac{\pi'}{\Gamma \vdash A'}\ ,\delta = mgu(A, A'), \delta \text{ feasible for } \pi \right\};\\
D_1 \wedge D_2 &= \left\{ \pi \;\middle|\; \frac{\pi_1 \quad \pi_2}{\pi}(\wedge) \quad \pi_1 \in D_1, \pi_2 \in D_2 \right\};\\
\mathcal{L}(D, A) &= \left\{ \pi \;\middle|\; \frac{\pi'}{\pi}(\vee_1, A) \quad ,\pi' \in D \right\};\\
\mathcal{R}(D, B) &= \left\{ \pi \;\middle|\; \frac{\pi'}{\pi}(\vee_2, B) \quad ,\pi' \in D \right\};\\
Impl(D, M) &= \left\{ \pi \;\middle|\; \frac{\pi'}{\pi}(\supset, M) \quad ,\pi' \in D \right\};\\
\exists x(D) &= \left\{ \pi \;\middle|\; \frac{\pi'}{\pi}(\exists x) \quad ,\pi' \in D \right\};\\
\forall x(D) &= \left\{ \pi \;\middle|\; \frac{\pi'}{\pi}(\forall x) \quad ,\pi' \in D \right\}.
\end{aligned}
$$

Figure 4: The semantic operators

concrete and abstract values [5, 6], we can build another semantics not necessarily with the same precision but however correct and possibly more efficient to work with.

Here and in the following we will suppose that the abstract domain has the form $(\mathcal{P}(\mathcal{H}), \subseteq)$. Given a function $\alpha : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{H})$, the *abstraction function* [7], we can associate to a set D its abstract description $\alpha(D)$.

Example 4.1 As an example of abstraction, we can see the *computed answer substitutions* of a program P. As abstract domain we put $\mathcal{H} = G \times Subst$. Then, given D a set of partial runs, we set

$$\alpha(D) = \left\{ \Delta \vdash G \;\middle|\; \pi \in D, \pi = \frac{\pi'}{P, \Delta \vdash G} \text{ is a proof tree (not partial)} \right\}$$

That is in D the sequent $P, \Delta \vdash G$ succeeds with computed substitution $\theta_{|G}$. The set Δ in the abstraction can be simply a set of indexes (we can associate statically an index to every loadable module of P). ∎

The problem at this point is to abstract correctly also the operators defined on the concrete domain of proof trees. In our case we have to find abstract version of Φ and Ψ.

A natural solution (suggested in [8]) can be given by recalling that these operators are defined starting from inference rule systems. We can easily build abstract versions of the systems $\mathcal{B}_\Sigma$ and $\mathcal{T}_\Sigma$. The induced operators, then, result to be the optimal abstractions of the operators associated to the concrete systems, that is the most precise functions that approximate them. Let us see things in details.

Given a concrete inference system Π with its associated operators Υ, and an abstraction function α, we can build the system Π_α on $\mathcal{H}$ this way

$$\Pi^\alpha = \left\{ \frac{\alpha(P)}{a} \;\middle|\; \frac{P}{c} \in \Pi \text{ and } a \in \alpha(\{c\}) \right\}$$

These inference rules induce the operator Υ_α and we can show the following theorem.

Theorem 3 *If α is a complete join morphism, then Υ_α is the optimal abstraction of Υ, that is*

$\alpha \circ \Upsilon(X) \subseteq \Upsilon_\alpha \circ \alpha(X)$, for each X;
$\alpha \circ \Upsilon(X) \subseteq \Lambda \circ \alpha(X)$ for each X and Λ monotone, implies $\Upsilon_\alpha(A) \subseteq \Lambda(A)$ for each A.

Let us recall that, under our hypotheses, α being a complete join morphism is equivalent to have a *Galois connection* $(\mathcal{P}(\mathcal{D}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{P}(\mathcal{H}), \subseteq)$, where γ is the ***adjoint*** function of α [7]. In general it is not the case that $\alpha \circ \Upsilon = \Upsilon_\alpha \circ \alpha$. A sufficient condition on α for the equality to hold, is the following property, drawn from [8].

Definition 1
The abstraction function α is said *precise* with respect to the inference system Π if

$$\forall \frac{P}{c} \in \Pi : \forall X \subseteq D : \alpha(P) \subseteq \alpha(X) \text{ implies } \exists \frac{P'}{c'} \in \Pi : P' \subseteq X \text{ and } \alpha(\{c\}) \subseteq \alpha(\{c'\}).$$

∎

We can show the following theorem

Theorem 4 *If α is precise with respect to Π, then $\alpha \circ \Upsilon = \Upsilon_\alpha \circ \alpha$.*

Let us see how these results instantiate to our case.
We can easily build the system $\mathcal{B}^\alpha_\Sigma$. Let Φ_α be the associated operator. From theorem 3 it results that Φ_α is the optimal abstraction of Φ. We have then the optimal abstract operator corresponding to the concrete bottom-up construction of Σ trees. Anyway to obtain only the P trees we have also to consider the projection operator $P \cdot (\_)$. We will say that a function α is *bottom-up precise* if it is precise with respect to $\mathcal{B}_\Sigma$ and for an opportune abstract function $P \bar{\cdot} (\_)$ verifies

$$\alpha(P \cdot (X)) = P \bar{\cdot} (\alpha(X)) \tag{1}$$

In general it can be found only a correct approximation of $P \cdot (\_)$, so Equation (1) is true by substituting containment to equality. Anyway given a function $P \bar{\cdot} (\_)$ that abstracts $P \cdot (\_)$, we can define $T_\alpha[\![P]\!]$ and $\mathcal{F}_\alpha[\![P]\!]$ this way

$$\begin{aligned}
&T_\alpha[\![P]\!]^k = P \bar{\cdot} \Phi^k_\alpha \qquad \text{for each } k\ ;\\
&\mathcal{F}_\alpha[\![P]\!] = \textstyle\bigcup_{k \geq 0} T_\alpha[\![P]\!]^k.
\end{aligned} \tag{2}$$

We have then the following theorem.

Theorem 5 *Let $\alpha : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{H})$ be a complete join morphism and T_α and $\mathcal{F}_\alpha$ defined as in (2). Then*

$$\alpha(T[\![P]\!]^k) \subseteq T_\alpha[\![P]\!]^k \quad \text{for each } k\ ;$$
$$\alpha(\mathcal{F}[\![P]\!]) \subseteq \mathcal{F}_\alpha[\![P]\!]$$

If α is bottom-up precise

$$\alpha(T[\![P]\!]^k) = T_\alpha[\![P]\!]^k \quad \text{for each } k\ ;$$
$$\alpha(\mathcal{F}[\![P]\!]) = \mathcal{F}_\alpha[\![P]\!]$$

Example 4.2 As an example of an abstract bottom-up system, we can see an extension of *s-semantics*[9]. The abstraction function has been defined in example 4.1. It can be simply showed that α is bottom-up precise. In fact for each $\frac{P}{c} \in \mathcal{B}_\Sigma$ we have that $\alpha(P) \subseteq \alpha(X)$ implies $P \subseteq X$. Moreover once defined $P : D = \{\vdash A \in D \mid A \text{ atomic } \}$ it can be checked that equation (1) is verified. The relative system $\mathcal{B}_\Sigma^\alpha$ can be written as follow.

$$\frac{}{\Delta \vdash \top}(\top) \qquad \frac{\Delta \vdash G_1}{\Delta \vdash G_1 \vee G_2}(\vee_1) \qquad \frac{\Delta \vdash G_2}{\Delta \vdash G_1 \vee G_2}(\vee_2)$$

$$\frac{\Delta \vdash G}{\Delta \setminus \{i\} \vdash M_i \supset G}(\supset) \qquad \frac{\Delta \vdash G_1 \quad \Delta \vdash G_2}{\Delta \vdash G_1 \wedge G_2}(\wedge)$$

$$\frac{\Delta \vdash G[t/x]}{\Delta \vdash \exists x G}(\exists) \qquad \frac{\Delta \vdash G[y/x]}{\Delta \vdash \forall x G}(\forall) \qquad \text{if } y \notin var(G)$$

$$\frac{\Delta \vdash G}{\Delta \vdash A\eta}(\bowtie) \qquad \text{if } G' \supset A \in P \text{ or } (G' \supset A \in M_i \text{ and } i \in \Delta) \text{ and } \eta = \text{mgu}(G', G).$$

∎

As regards the system $\mathcal{T}_\Sigma$, given α we can easily build $\mathcal{T}_\Sigma^\alpha$. Let Ψ_α be the induced operator. Fixed a program P and a goal G, let

$$\mathcal{C}_\alpha[\![P \vdash G]\!] = \bigcup_{k \geq 0} \Psi_\alpha^k(\alpha(\{\frac{}{P \vdash G}\})) \tag{3}$$

be the abstract computations operator. We can define the abstract operational semantics as

$$\mathcal{O}_\alpha[\![P]\!] = \bigcup_{A \in \mathcal{A}} \mathcal{C}_\alpha[\![P \vdash A]\!] \tag{4}$$

A function α is said *top-down precise* if it is precise with respect to $\mathcal{T}_\Sigma$. We can state the following theorem

Theorem 6 *Let $\alpha : \mathcal{P}(D) \to \mathcal{P}(H)$ be a complete join morphism and $\mathcal{C}_\alpha[\![P \vdash G]\!]$ and $\mathcal{O}_\alpha[\![P]\!]$ be defined as in (3) and (4). Then*

$$\alpha(\mathcal{O}[\![P]\!]) \subseteq \mathcal{O}_\alpha[\![P]\!]$$
$$\alpha(\mathcal{C}[\![P \vdash G]\!]) \subseteq \mathcal{C}_\alpha[\![P \vdash G]\!]$$

If α is top-down precise

$$\alpha(\mathcal{O}[\![P]\!]) = \mathcal{O}_\alpha[\![P]\!]$$
$$\alpha(\mathcal{C}[\![P \vdash G]\!]) = \mathcal{C}_\alpha[\![P \vdash G]\!]$$

Example 4.3 An example of top-down analysis is the *call patterns analysis* for a fixed goal G, that gives the way clauses are called in a computation starting with G. We choose as abstraction the function

$$\alpha(D) = \left\{ (G; \Delta \vdash F) \;\middle|\; \pi \in D,\ P, \Delta \vdash F \in \overline{\pi}, \pi = \frac{\pi'}{P, \Lambda \vdash G} \right\}$$

It can be checked that the function is precise. Once calculated the abstract semantics $\mathcal{O}_\alpha[\![P]\!]$, exploiting the induced operator Ψ_α and equation (3) and (4), we can project on the real call pattern, that is we single out atomic formulas

$$CallPatt = At \cdot \mathcal{O}_\alpha[\![P]\!]$$
$$\text{where } At \cdot D = \{(G; \Delta \vdash A) \mid (G; \Delta \vdash A) \in D, A \text{ atomic } \}$$

The system $\mathcal{T}_\Sigma^\alpha$, relative to the proposed abstraction, is showed below.

$$\frac{(G; \Delta \vdash F_1 \vee F_2)}{(G; \Delta \vdash F_1)}(\vee_1) \qquad \frac{(G; \Delta \vdash F_1 \vee F_2)}{(G; \Delta \vdash F_2)}(\vee_2)$$

$$\frac{(G; \Delta \vdash F_1 \wedge F_2)}{(G; \Delta \vdash F_1)}(\wedge) \qquad \frac{(G; \Delta \vdash F_1 \wedge F_2)}{(G; \Delta \vdash F_2)}(\wedge)$$

$$\frac{(G; \Delta \vdash M_i \supset F)}{(G; \Delta \cup \{i\} \vdash F)}(\supset) \qquad \frac{(G; \Delta \vdash \exists x F)}{(G; \Delta \vdash F[y/x])}(\exists) \qquad \frac{(G; \Delta \vdash \forall x F)}{(G; \Delta \vdash F[y/x])}(\forall)$$

In the rule $\forall$, $y \in W$.

$$\frac{(G; \Delta \vdash A)}{(G; \Delta \vdash G\eta)}(\bowtie)$$

The clause $\forall(G' \supset A')$ is in $\Delta \cup P$, $\eta = mgu(A, A')$, η does not instantiate variables in $W \cap var(A)$.

∎

At this point we may ask about the compositionality of the operator $\mathcal{C}_\alpha[\![\Gamma \vdash G]\!]$. It can be seen that the top-down precision does not suffices. In fact it can be noted

$$
\begin{aligned}
D_1 \tilde{\wedge} D_2 &= \left\{ a \;\middle|\; \frac{\pi_1 \quad \pi_2}{\pi}(\wedge) \quad \alpha(\{\pi_1\}) \subseteq D_1, \;\; \alpha(\{\pi_2\}) \subseteq D_2, a \in \alpha(\{\pi\}) \right\};\\
\mathcal{L}_\alpha(D, A) &= \left\{ a \;\middle|\; \frac{\pi'}{\pi}(\vee_1, A') \quad \alpha(\{\pi'\}) \subseteq D, a \in \alpha(\{\pi\}) \right\};\\
\mathcal{R}_\alpha(D, B) &= \left\{ a \;\middle|\; \frac{\pi'}{\pi}(\vee_2, B') \quad \alpha(\{\pi'\}) \subseteq D, a \in \alpha(\{\pi\}) \right\};\\
Impl_\alpha(D, M) &= \left\{ a \;\middle|\; \frac{\pi'}{\pi}(\supset, M') \quad \alpha(\{\pi'\}) \subseteq D, a \in \alpha(\{\pi\}) \right\};\\
\tilde{\exists} x(D) &= \left\{ a \;\middle|\; \frac{\pi'}{\pi}(\exists x) \quad \alpha(\{\pi'\}) \subseteq D, a \in \alpha(\{\pi\}) \right\};\\
\tilde{\forall} x(D) &= \left\{ a \;\middle|\; \frac{\pi'}{\pi}(\forall x) \quad , \alpha(\{\pi'\}) \subseteq D, a \in \alpha(\{\pi\}) \right\}.
\end{aligned}
$$

Figure 5: The abstract semantic operators

that the concrete compositionality depends on operators that are essentially bottom-up inference rules. This suggests that to obtain compositionality for the abstract computations, a sufficient property is the precision with respect to the inference rules of $\mathcal{B}_\Sigma$, at least those relative to connectives that appear in goal formulas.

Actually we can then define the operators on $\mathcal{P}(\mathcal{H})$ of Fig. 5, that abstract the corresponding concrete operators and correspond to inference rules of $\mathcal{B}^g_\Sigma$. Next theorem shows under which conditions we can demonstrate the compositionality of $\mathcal{C}_\alpha[\![P \vdash G]\!]$.

Theorem 7 *Let $\alpha : \mathcal{P}(D) \to \mathcal{P}(H)$ be a complete join morphism, P, M be programs and A, G, G_1, G_2 goal formulas. If α is top-down precise, precise with respect to $\mathcal{B}_\Sigma$ (at least wrt to the rules $\wedge, \vee, \supset, \exists, \forall$) and it exists $A \tilde{\cdot} (\_)$ such that $\alpha(A \cdot (X)) = A \tilde{\cdot} (\alpha(X))$, then*

$$
\begin{aligned}
\mathcal{C}_\alpha[\![P \vdash A]\!] &= A \tilde{\cdot} \mathcal{O}_\alpha[\![P]\!], \text{ if } A \text{ is an atom};\\
\mathcal{C}_\alpha[\![P \vdash G_1 \wedge G_2]\!] &= \mathcal{C}_\alpha[\![P \vdash G_1]\!] \;\tilde{\wedge}\; \mathcal{C}_\alpha[\![P \vdash G_2]\!];\\
\mathcal{C}_\alpha[\![P \vdash G_1 \vee G_2]\!] &= \mathcal{L}_\alpha(\mathcal{C}_\alpha[\![P \vdash G_1]\!], G_2) \cup \mathcal{R}_\alpha(\mathcal{C}_\alpha[\![P \vdash G_2]\!], G_1);\\
\mathcal{C}_\alpha[\![P \vdash M \supset G]\!] &= Impl_\alpha(\mathcal{C}_\alpha[\![P, M \vdash G]\!]);\\
\mathcal{C}_\alpha[\![P \vdash \exists x G]\!] &= \tilde{\exists} x(\mathcal{C}_\alpha[\![P \vdash G]\!]);\\
\mathcal{C}_\alpha[\![P \vdash \forall x G]\!] &= \tilde{\forall} x(\mathcal{C}_\alpha[\![P \vdash G]\!]);
\end{aligned}
$$

Finally we can set out the relations between the abstract bottom-up and top-down semantics. It can be easily derived the following result.

Theorem 8 *Let $\alpha : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{H})$ be a complete join morphism, P a program, $\mathcal{F}_\alpha[\![P]\!]$ and $\mathcal{O}_\alpha[\![P]\!]$ defined as in (2) and (4). Then*

$$
\begin{aligned}
\alpha(\mathcal{F}[\![P]\!]) &\subseteq \mathcal{O}_\alpha[\![P]\!]\\
\alpha(\mathcal{O}[\![P]\!]) &\subseteq \mathcal{F}_\alpha[\![P]\!].
\end{aligned}
$$

If α is bottom-up precise, then

$$\mathcal{F}_\alpha[\![P]\!] = \alpha(\mathcal{F}[\![P]\!]) \subseteq \mathcal{O}_\alpha[\![P]\!].$$

If α is top-down precise, then

$$\mathcal{O}_\alpha[\![P]\!] = \alpha(\mathcal{O}[\![P]\!]) \subseteq \mathcal{F}_\alpha[\![P]\!].$$

If α is both bottom-up and top-down precise, then

$$\mathcal{F}_\alpha[\![P]\!] = \alpha(\mathcal{F}[\![P]\!]) = \alpha(\mathcal{O}[\![P]\!]) = \mathcal{O}_\alpha[\![P]\!].$$

5 Conclusion

In this work we have set out a general framework to do static analysis of programs written in proof theory based extensions of logic programming. The framework is based on the abstract interpretation approach and on proof-theoretic features of the considered languages. In particular we have proposed a framework based on inductive definitions to express the bottom-up and the top-down constructions of proof trees in the system Σ, which is a subset of λ-Prolog. We have proposed the extension of the framework to a more abstract domain, in which a possible less precise but anyway correct account of computations is given. In the abstract domain, the system can be characterized again in terms of inference rules. Finally some conditions on the abstraction function have been devised that allow to restate the equivalence of the bottom-up and top-down approaches and compositionality of abstract operators also on the abstract domain.

The framework is extensible to other languages in a straightforward way, by exploiting sequent calculi formalizations. This can be a subject for next works. Indeed we think it could be quite interesting to give an analogue treatment for linear logic based languages. This can be a base for the study of properties of resource conscious and concurrent languages [12, 1, 15, 18]. We think that this can be a good test for the suitability of abstract interpretation approach to the study of consumption of resource, of communication and synchronization.

We think a problem to solve is the reduction of the properties of precision with respect to $\mathcal{B}_\Sigma$ and $\mathcal{T}_\Sigma$, to properties relative to the base system Σ. We think that opportune constraints can be found on α relatively to Σ that imply the bottom-up or top-down precision.

Finally another possible direction to develop is the extension of the framework to constraint based languages [13]. In our case we have worked on tuples (π, θ), with θ a set of equality constraints on the Herbrand Universe. In the same way we can consider inference rules on tuples (π, c), with c a set of constraints. We are working

to establish if this view can be carried on coherently.
As regards relations with other works, we are not aware of other papers that face the static analysis of proof theory based languages. Anyway it must be said that our work is reminiscent of results found in [4]. However in our case we are more oriented to find effective calculi for abstract properties rather than depict an algebraic framework to classify them.

References

[1] J. M. Andreoli and R. Pareschi. Linear Objects: logical processes with built-in inheritance. In D. H. D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, pages 495–510. The MIT Press, Cambridge, Mass., 1990.

[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] P. Bresset and O. Ridoux. The architecture of an Implementation of λ-Prolog: Prolog/Mali. Technical Report 879, IRISA, 1994.

[4] M. Comini, G. Levi, and M.C. Meo. Compositionality of *SLD*-derivations and their Abstractions. In J. Lloyd, editor, *Proc. of 1995 International Symposium on Logic Programming*, 1995.

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

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

[7] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *J. Logic Computat.*, 2(4):511–547, 1992.

[8] P. Cousot and R. Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages*, pages 83–94, 1992.

[9] 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.

[10] J. Gallier. *Logic for Computer Science*. Harper & Row, 1986.

[11] L. Giordano and A. Martelli. A Modal Reconstruction of Blocks and Modules in Logic Programming. In *Proc. 8th Int. Conf. on Logic Programming*, pages 239–253. The MIT Press, Cambridge, Mass., 1991.

[12] J. S. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 32–42, 1991.

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

[14] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6:79–108, 1989.

[15] D. Miller. A Multiple-Conclusion Meta Logic. In S. Abramsky, editor, *Ninth Annual IEEE Symposium on Logic in Computer Science*, 1994.

[16] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[17] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Department of Computer Science, UC Berkeley, 1990. Report UCB/CSD 90/600.

[18] P. Volpe. Concurrent Logic Programming as Uniform Linear Proofs. In G. Levi and M. Rodriguez-Artalejo, editors, *Proc. of the Fourth International Conference on Algebraic and Logic Programming*, volume 850 of *Lecture Notes in Computer Science*, pages 133–149. Springer-Verlag, Berlin, 1994.