Abstract Interpretation Based Semantics of Sequent Calculi

Gianluca Amato and Giorgio Levi

Università di Pisa, Dipartimento di Informatica
corso Italia 40, 56125 Pisa, Italy
{amato,levi}@di.unipi.it

Abstract. In the field of logic languages, we try to reconcile the proof theoretic tradition, characterized by the concept of uniform proof, with the classic approach based on fixpoint semantics. Hence, we propose a treatment of sequent calculi similar in spirit to the treatment of Horn clauses in logic programming. We have three different semantic styles (operational, declarative, fixpoint) that agree on the set of all the proofs for a given calculus. Following the guideline of abstract interpretation, it is possible to define abstractions of the concrete semantics, which model well known observables such as correct answers or groundness. This should simplify the process of extending important results obtained in the case of positive logic programs to the new logic languages developed by proof theoretic methods. As an example of application, we present a top-down static analyzer for properties of groundness which works for full intuitionistic first order logic.

1 Introduction

One of the greatest benefits of logic programming, as presented in [14], is that it is based upon the notion of *executable specifications*. The text of a logic program is endowed with both an operational (algorithmic) interpretation and an independent mathematical meaning which agree each other in several ways. The problem is that operational expressiveness (intended as the capability of directing the flow of execution of a program) tends to obscure the declarative meaning. Research in logic programming strives to find a good balance between these opposite needs.

Uniform proofs [17] have widely been accepted as one of the main tools for approaching the problem and to distinguish between logic without a clear computational flavor and logic programming languages. However, that of uniform proofs being a concept heavily based on proof theory, researches conducted along this line have always been quite far from the traditional approach based on fixpoint semantics. In turn, this latter tradition has brought up several important

[1] The proofs of the properties which appear in the text are available at the the following URL: `http://www.di.unipi.it/~amato/papers`.

results concerning the effective utilization of Horn clauses as a real programming language. Among the others, problems such as compositionality of semantics [7], modularity [5, 6], static analysis [13], debugging [8], have been tackled in this setting. Adapting these results to the new logic languages developed via the proof theoretic approach, such as λProlog [19] or LinLog [3], would probably require at least two things:

- provide a fixpoint semantics for these new languages;
- generalize a great number of concepts whose definition is too much tied to the case of Horn clauses.

This paper proposes a semantic framework which can be useful in such an effort. The main idea is to recognize proofs in the sequent calculi as the general counterpart of SLD resolutions for positive logic programs. Thus, the three well-known semantics (operational, declarative and fixpoint) for Horn clause logic can be reformulated within this general setting and directly applied to all the logic languages based on sequent calculi.

Moreover, these semantics are generalized to be parametric w.r.t. a *pre-interpretation*, which is essentially a choice of semantic domains and intended meanings for the inference rules. When a pre-interpretation is given, we have fixed a particular property of the proofs we want to focus our attention on (correct answers, resultants, groundness). Hence, classical abstractions such as correct answers or resultants, used in the semantic studies of logic programs, and abstractions for static analysis like groundness, can be retrieved in terms of properties of proofs. Expressed in such a way, rather than referring to a computational procedure like SLD resolution, they are more easily extendable to other logic languages.

It turns out that the most convenient way of defining pre-interpretations is through abstract interpretation theory [11]. In this way, we provide a semantic framework for the new proof-theoretic based logic languages to which most of the studies we have for positive logic programs can be easily adapted.

The paper is much inspired by [7]. After some preliminaries, we introduce the three semantic styles for sequent calculi, with respect to a particular concrete pre-interpretation. It is shown that the three semantics coincide on the set of proofs for a given calculus. Later, using techniques of abstract interpretation theory, the concept of observable is introduced, as an abstraction of a set of proofs. This gives corresponding notions of abstract semantics for sequent calculi. In general, the properties of the abstract semantics will depend on completeness properties of the abstract optimal semantic operators. For some observables, those that *separate sequents*, there is a strict correspondence between abstract semantics and semantics in a particular pre-interpretation induced by the observable.

Then, some examples of observables are provided: success sets, correct answers, groundness. All of them are presented in the case of full first-order intuitionistic logic or in a yet more general setting. This gives an idea of the process of importing well know observables for positive logic programs to broader fragments of logic. This should be the first step towards a more detailed semantic analysis of these extended languages. In particular, as an example of application

of the proposed methodology, a top-down static interpreter for groundness analysis of intuitionistic logic has been implemented in PROLOG and it is presented in summary. Finally, possible future developments are discussed.

2 Proofs and Proof Schemas

2.1 Basic Definitions

Logics can be presented in several different ways: we will stick to a Gentzen-like proof-theoretic formulation. Let us fix two languages $\mathcal{D}$ and $\mathcal{G}$. We call *clauses* the elements of $\mathcal{D}$ and *goals* the elements of $\mathcal{G}$. If Γ and Δ are two sequences of clauses and goals respectively, $\Gamma \twoheadrightarrow \Delta$ is a *sequent*. If Δ is a sequence of length not greater than one, we have an *intuitionistic* sequent. In the following, we will use the letter S and its variants to denote sequents.

A *proof schema* is a rooted ordered tree with sequents as nodes. Given a proof schema p, we call $\mathrm{hyp}(p)$ (*hypotheses*) the sequence of leaves of p taken from a pre-order visit of p and we write $\mathrm{th}(p)$ (*theorem*) for the root of p. When we want to state that p is a proof schema with $\mathrm{hyp}(p) = S_1, \ldots, S_n$ and $\mathrm{th}(p) = S$, we write

$$p : S_1, \ldots, S_n \vdash S \ . \tag{1}$$

Note that a schema of height zero (when the root is also the only hypothesis) and a schema of height one with no hypotheses are different objects. If S is a sequent, a proof schema of S with height zero is the *empty proof schema* for S, and is denoted by ϵ_S. This is because we assume that leaves are different from nodes with out-degree zero. The former correspond to hypotheses, while the latter are sequents introduced by an axiom.

Example 1. If $\mathcal{D}$ and $\mathcal{G}$ are first order languages with unary predicate symbols p and r, the following is a proof schema

$$\dfrac{\dfrac{p(x) \twoheadrightarrow r(x) \qquad \cdot \twoheadrightarrow r(y)}{\cdot \twoheadrightarrow r(x) \wedge r(y)} \qquad \cdot \twoheadrightarrow \exists z.p(z)}{\forall z.p(z) \twoheadrightarrow \forall w.r(w)} \tag{2}$$

Note that it is not a proof in any of the standard logical systems. The following are respectively a proof schema $p : \cdot \vdash p(x) \twoheadrightarrow p(x)$ and the empty proof schema $p' : q(x) \twoheadrightarrow p(x) \vdash q(x) \twoheadrightarrow p(x) = \epsilon_{q(x) \twoheadrightarrow p(x)}$

$$p : \ \dfrac{}{p(x) \twoheadrightarrow p(x)} \qquad\qquad p' : \ \ q(x) \twoheadrightarrow p(x) \tag{3}$$

according to our convention on nodes with out-degree zero. □

Now, we fix a set $\mathcal{R}$ of proof schemas of height one. We call *inference rules* the elements of $\mathcal{R}$. A proof schema p, which is obtained by gluing together the empty proof schemas and the inference rules, is called *proof*. A proof with no hypothesis is said to be *final*. A sequent S is *provable* if there is a final proof rooted at S. Finally, we call *logic* a triple $\langle \mathcal{D}, \mathcal{G}, \mathcal{R} \rangle$.

Example 2. Assume $\mathcal{R}$ is the set of inference rules for first order logic. Then p and p' in the previous example are proofs. In particular, p is a final proof. Another proof, a bit more involved, is the following

$$\dfrac{\dfrac{\Gamma \twoheadrightarrow \forall x.p(x)}{\Gamma \twoheadrightarrow p(w)}}{\Gamma \twoheadrightarrow \exists w.p(w)} \tag{4}$$

where Γ is a list of sequents. □

In the following, we will often work with the logic $\mathcal{L}_{hc}$ of Horn clauses and with the first order intuitionistic logic $\mathcal{L}_i$. However, with the exception of Sect. 5, all the framework can be directly applied to a generic logic system. Just note that what we traditionally call inference rule, i.e. something like

$$\dfrac{\Gamma \twoheadrightarrow G_1 \qquad \Gamma \twoheadrightarrow G_2}{\Gamma \twoheadrightarrow G_1 \wedge G_2} \ ,$$

should be viewed as a collection of different inference rules, one for each instance of G_1, G_2 and Γ.

2.2 Semantic Operators

Given a sequent S, we denote by Sch_S the set of all the proof schemas rooted at S. For each $p \in \mathsf{Sch}_S$ of the form

$$p : S_1, \ldots, S_n \vdash S \ , \tag{5}$$

we have a corresponding semantic operator $p : \mathsf{Sch}_{S_1} \times \cdots \times \mathsf{Sch}_{S_n} \to \mathsf{Sch}_S$ which works by gluing proof schemas of the input sequents together with p, to obtain a new proof schema of the output sequent S. If Sch is the set of all the proof schemas, $p : S_1, \ldots, S_n \vdash S \in \mathsf{Sch}$ and $X_i \subseteq \mathsf{Sch}$ for each i, we define a collecting variant of the semantic operator p, defined as

$$p(X_1, \ldots, X_n) = \{p(p_1, \ldots, p_n) \mid \forall i.\ p_i \in X_i \cap \mathsf{Sch}_{S_i}\} \ . \tag{6}$$

We will write $p(X)$ as a short form for $p(X, \ldots, X)$ with n identical copies of X as input arguments.

Working with a semantic operator for each proof schema is quite uncomfortable, especially when reasoning in terms of abstractions. We can actually resort to a unique *gluing operator*. Given X_1 and X_2 subsets of Sch, we denote by $X_1 \rhd X_2$ the set

$$X_1 \rhd X_2 = \bigcup_{p \in X_1} p(X_2) \ . \tag{7}$$

In other words, $X_1 \rhd X_2$ is the result of gluing together each proof schema in X_1 with all the "compatible" proof schemas in X_2. It turns out that $\rhd$ is (roughly) the counterpart for sequent calculi of the $\bowtie$ operator for SLD derivations defined in [9].

Example 3. Consider the proof p in $\mathcal{L}_{hc}$ given by

$$\dfrac{\dfrac{\forall x.p(x) \twoheadrightarrow p(a) \qquad \forall x.r(x) \twoheadrightarrow r(b)}{\forall x.p(x), \forall x.r(x) \twoheadrightarrow p(a) \wedge r(b)}}{\forall x.p(x) \wedge \forall x.r(x) \twoheadrightarrow p(a) \wedge r(b)} \tag{8}$$

and the proofs p'

$$\dfrac{\dfrac{}{p(a) \twoheadrightarrow p(a)}}{\forall x.p(x) \twoheadrightarrow p(a)} \tag{9}$$

and $p'' = \epsilon_{\forall x.r(x) \twoheadrightarrow r(b)}$. Then, the proof $p(p', p'')$ is

$$\dfrac{\dfrac{\dfrac{}{p(a) \twoheadrightarrow p(a)}}{\forall x.p(x) \twoheadrightarrow p(a)} \qquad \forall x.r(x) \twoheadrightarrow r(b)}{\dfrac{\forall x.p(x), \forall x.r(x) \twoheadrightarrow p(a) \wedge r(b)}{\forall x.p(x) \wedge \forall x.r(x) \twoheadrightarrow p(a) \wedge r(b)}} \tag{10}$$

In particular, note that gluing with empty proofs has no effects. □

3 The Concrete Semantics

Given a logic $\mathcal{L} = \langle \mathcal{D}, \mathcal{G}, \mathcal{R} \rangle$, we can introduce three different styles of semantics, similar in spirit to the operational, declarative and fixpoint semantics of classic logic programming. We follow the idea underlying [9] of having a common set of semantic operators for both the top-down (operational) and the bottom-up (fixpoint) styles.

3.1 Declarative Semantics

The fundamental idea is that a logic can be viewed as a signature for Σ-algebras, where sequents correspond to sorts and inference rules to term symbols. A Σ-algebra gives a choice of a semantic domain for each sequent and of a semantic function for each inference rule. Roughly, a model for a logic in a given Σ-algebra should assign, to each sequent, an element of its corresponding semantic domain, in such a way that this assignment is well-behaved w.r.t. the inference rules.

To be more precise, we call *pre-interpretation* the choice of a nonempty ordered set $\mathcal{I}(\Gamma \twoheadrightarrow \Delta)$ for each sequent and of a monotonic function $\mathcal{I}(r)$ for each inference rule r, where if $\mathrm{hyp}(r) = S_1, \ldots, S_n$ and $\mathrm{th}(r) = S$,

$$\mathcal{I}(r) = \mathcal{I}(S_1) \times \cdots \times \mathcal{I}(S_n) \to \mathcal{I}(S) \ . \tag{11}$$

Therefore, the concept of *pre-interpretation* is the same of ordered Σ-algebras as defined in [18].

Given a logic $\mathcal{L}$ with a pre-interpretation $\mathcal{I}$, an *interpretation* is a choice of an element $[\![S]\!] \in \mathcal{I}(S)$ for each sequent S. An interpretation is a *model* when, for each inference rule

$$r : S_1, \ldots, S_n \vdash S \ , \tag{12}$$

the following relation holds

$$\mathcal{I}(r)([\![S_1]\!], \ldots, [\![S_n]\!]) \sqsubseteq [\![S]\!] \ . \tag{13}$$

The notion of pre-interpretation gives us a great flexibility. In [1] it is shown how to obtain well known semantics such as correct answers or Heyting semantics for a generic sequent calculus.

When we talk of programming languages, the idea is that a program P corresponds to a sequence of clauses. Given a goal G and a model $[\![\_]\!]$, the corresponding semantics of G in the program P is given by $[\![P \twoheadrightarrow G]\!]$.

Example 4. In the logic $\mathcal{L}_{hc}$, consider the pre-interpretation $\mathcal{I}$ given by

- $\mathcal{I}(S) = \{\mathsf{true}, \mathsf{false}\}$ with $\mathsf{false} \sqsubseteq \mathsf{true}$;
- if $r \in \mathcal{R}$ is the inference rule $r : S_1, \ldots, S_n \vdash S$, then $\mathcal{I}(r)$ is the logical conjunction of the n input values. If r has no hypothesis, then $\mathcal{I}(r) = \mathsf{true}$.

If P is a definite logic program, i.e. a set of definite clauses, and $[\![\_]\!]$ is an interpretation, the set

$$I_P = \left\{ A \mid \left[\!\!\left[\vec{\forall} P \twoheadrightarrow A \right]\!\!\right] = \mathsf{true} \text{ and } A \text{ is a ground atomic goal} \right\} \tag{14}$$

is a Herbrand interpretation, where $\vec{\forall} P$ is the universal closure of the clauses in P. Moreover, if $[\![\_]\!]$ is a model, I_P is a Herbrand model.

Note that, given a `PROLOG` clause `G :- B`, the corresponding clause in $\mathcal{L}_{hc}$ is the universally quantified formula $\vec{\forall}.(B \supset G)$. As a result, a query `G` for a definite program becomes $\exists.G$ in $\mathcal{L}_{hc}$. Actually, the sequent

$$\forall x.(p(x) \supset q(x)), \forall x.p(x) \twoheadrightarrow \exists y.q(y) \tag{15}$$

has an obvious final proof, but

$$p(x) \supset q(x), p(x) \twoheadrightarrow q(y) \tag{16}$$

is not provable since free variables are never instantiated in the sequent calculus for $\mathcal{L}_{hc}$. □

A major drawback of this approach is that the process of defining a pre-interpretation is quite arbitrary, especially for what concerns the inference rules. In the following, we try to overcome this problem by just sticking to a specific concrete pre-interpretation and deriving all the others by abstraction functions, according to the theory of abstract interpretation.

Given a logic $\mathcal{L}$, consider the *syntactic* pre-interpretation $\mathcal{I}_{\mathcal{L}}$ given by

– $\mathcal{I}_{\mathcal{L}}(S) = \langle \mathcal{P}(\mathsf{Sch}_S), \subseteq \rangle$ for each sequent S;
– $\mathcal{I}_{\mathcal{L}}(r)$ is the semantic function corresponding to $r \in \mathcal{R}$, as in (6).

Interpretations for $\mathcal{I}_{\mathcal{L}}$ are called *syntactical interpretations*. In the following, these will be denoted by subsets of Sch. The convention does not rise any ambiguities, since if $S_1 \neq S_2$, then $\mathsf{Sch}_{S_1} \cap \mathsf{Sch}_{S_2} = \emptyset$. A syntactical model, therefore, is a set of proof schemas closed under application of inference rules. We denote by Int the set of all the syntactical interpretations, which is a complete lattice under subset ordering. In the remaining of this section, when we talk of interpretations or models we always refer to the syntactical ones, unless otherwise stated.

It is possible to concisely express the condition of a syntactical interpretation I being a model using the glue operator. The property to be satisfied is

$$\mathcal{R} \rhd I \subseteq I \ . \tag{17}$$

Models form a complete lattice under the same ordering of the interpretations. However, it is not a sublattice, since the join operator and the bottom element differ. In particular, the bottom element of the lattice of models is what we call *declarative semantics* of $\mathcal{L}$ and we denote it by $\mathcal{D}(\mathcal{L})$.

$\mathcal{D}(\mathcal{L})$ turns out to be the set of final proofs of $\mathcal{L}$. Hence, the declarative semantics precisely captures all the terminating computations. For a valid treatment of compositionality, we also need information about partial computations [5]. If ϵ is the set of all the empty proof schemas, we call *complete declarative semantics* of $\mathcal{L}$, and we denote it by $\mathcal{D}_c(\mathcal{L})$, the least model greater then ϵ. It is possible to prove that $\mathcal{D}_c(\mathcal{L})$ is actually the set of all the proofs of $\mathcal{L}$.

3.2 Top-down and Bottom-up Semantics

The definition of the declarative semantics is non-constructive. We now present a bottom-up construction of the least model using an operator similar to the immediate consequence operator T_P of logic programming. The $T_{\mathcal{L}}$ operator, mapping interpretations to interpretations, is defined as follows

$$T_{\mathcal{L}}(I) = I \cup (\mathcal{R} \rhd I) \ . \tag{18}$$

We can prove that all the results which hold for the T_P operator apply to $T_{\mathcal{L}}$ as well. In particular an interpretation I is a model iff it is a fixpoint of $T_{\mathcal{L}}$. Moreover $T_{\mathcal{L}}$ is continuous, hence $T_{\mathcal{L}} \uparrow \omega$ is its least fixpoint. We call $T_{\mathcal{L}} \uparrow \omega$ the *fixpoint semantics* of $\mathcal{L}$. It trivially follows that the fixpoint and declarative semantics do coincide. Analogously to the complete declarative semantics, we can define a *complete fixpoint semantics* as $T^{\omega}_{\mathcal{L}}(\epsilon)$. As in the previous case, $T^{\omega}_{\mathcal{L}}(\epsilon) = \mathcal{D}_c(\mathcal{L})$.

Note that inference rules are essentially treated like Horn clauses for a predicate `is_a_proof`/1. For example, an inference rule like

$$\frac{\Gamma \twoheadrightarrow \varphi \quad \Gamma \twoheadrightarrow \psi}{\Gamma \twoheadrightarrow \varphi \wedge \psi} \tag{19}$$

corresponds to the Horn clause

$$\begin{aligned}\texttt{is\_a\_proof}(\texttt{der}(\Gamma \twoheadrightarrow \varphi \wedge \psi, [P_1, P_2])) :- \\ P_1 = \texttt{der}(\Gamma \twoheadrightarrow \varphi, \_), P_2 = \texttt{der}(\Gamma \twoheadrightarrow \psi, \_), \\ \texttt{is\_a\_proof}(P_1),\ \texttt{is\_a\_proof}(P_2)\end{aligned} \tag{20}$$

where $\texttt{der}(\mathit{Sequent}, \mathit{List\_of\_Proof\_Schemas})$ is a coding for proof schemas. In general, we have an infinite set of ground Horn clauses, since every instance of (19) counts as a different inference rule and variables in the logic $\mathcal{L}$ are coded as ground objects at the Horn clause level. These properties play a fundamental role when we try to modify the set of inference rules to obtain new derived logic systems, such as uniform logics [1].

The fixpoint construction is essentially a bottom-up process. Real interpreters, on the contrary, follow a top-down approach, since it is generally more efficient. We consider here a transition system $(\mathsf{Sch}, \longmapsto)$ that emulates such a behavior. Assume $p : S_1, \ldots, S_n \vdash S$ is a proof schema and $r : S'_1, \ldots, S'_m \vdash S_i$ is an inference rule. We can define a new proof schema $p' = p(\epsilon_{S_1}, \ldots, r, \ldots, \epsilon_{S_n})$ just replacing S_i in the hypotheses of p with the inference rule r. We write $p \longmapsto p'$ when the above conditions are satisfied. In general, it is possible to replace more than one hypothesis, hence we have the following transition rule

$$p \longmapsto p(r_1, \ldots, r_n) \text{ when } \begin{cases} p : S_1, \ldots, S_n \vdash S, \\ r_i \in \mathcal{R} \cup \epsilon \text{ and } \mathrm{th}(r_i) = S_i \text{ for each } 1 \leq i \leq n. \end{cases} \tag{21}$$

We call *complete operational semantics* of $\mathcal{L}$ the interpretation

$$\mathcal{O}_c(\mathcal{L}) = \{p \in \mathsf{Sch} \mid \exists S.\ \epsilon_S \longmapsto^\star p\}\ . \tag{22}$$

It is possible to give a collecting variant of the operational semantics construction, via a fixpoint operator $U_\mathcal{L}$ on interpretations which uses the gluing semantic operator:

$$U_\mathcal{L}(I) = I \rhd (\mathcal{R} \cup \epsilon)\ . \tag{23}$$

The idea is that $U_\mathcal{L}(I)$ contains all the proof schemas derived by I with a step of the transition system, i.e.

$$U_\mathcal{L}(I) = \{p' \mid \exists p \in I.\ p \longmapsto p'\}\ . \tag{24}$$

Actually, we can prove that $U^\omega_\mathcal{L}(\epsilon) = \mathcal{O}_c(\mathcal{L})$. Moreover, we have $U^\omega_\mathcal{L}(\epsilon) = \mathcal{D}_c(\mathcal{L})$. Hence, all the different styles of semantics do coincide.

From the implementation viewpoint, the great advantage of the top-down operational semantic w.r.t. the bottom-up fixpoint one is that we do not need to compute the entire semantics if we are only interested in part of it. An interpreter for a logic language typically works with a program P and a goal G, trying to obtain the proofs of the sequent $P \twoheadrightarrow G$. The semantics of every other sequent in the logic is computed only if it is needed for computing the semantics of $P \twoheadrightarrow G$.

We call *query* whatever sequent in the logic $\mathcal{L}$. According to this definition, a query is a pair made of a program and a goal. We define the *operational behavior* of $\mathcal{L}$ as a function $\mathcal{B}(\mathcal{L}) : \mathsf{Query} \to \mathsf{Int}$ such that

$$\mathcal{B}(\mathcal{L})_Q = \{p \in \mathsf{Sch} \mid \epsilon_Q \longmapsto^\star p\} \ . \tag{25}$$

In other words, $\mathcal{B}(\mathcal{L})_Q$ is the set of proofs for the sequent Q in the logic $\mathcal{L}$. The fixpoint operator $U_\mathcal{L}$ can be used to compute $\mathcal{B}(\mathcal{L})$ since it is $\mathcal{B}(\mathcal{L})_Q = U^\omega_\mathcal{L}(\{\epsilon_Q\})$.

There is an immediate result of compositionality for $\mathcal{B}$. For each sequent S, consider the set $R = \{r_i\}_{i\in I}$ of all the inference rules rooted at S, such that $r_i : S_{i,1}, \ldots, S_{i,m_i} \vdash S$. We have

$$\mathcal{B}(\mathcal{L})_S = \bigcup_{i\in I} r_i \left(\mathcal{B}(\mathcal{L})_{S_{i,1}}, \ldots, \mathcal{B}(\mathcal{L})_{S_{i,m_i}}\right) \ . \tag{26}$$

Unfortunately, this result is not what we desire in most of the cases, as shown by the following example.

Example 5. When we work in $\mathcal{L}_{hc}$, the above compositionality result gives us the following property:

$$\mathcal{B}(\mathcal{L}_{hc})_{P\twoheadrightarrow G_1\wedge G_2} = \mathcal{B}(\mathcal{L}_{hc})_{P\twoheadrightarrow G_1} \wedge \mathcal{B}(\mathcal{L}_{hc})_{P\twoheadrightarrow G_2} \ . \tag{27}$$

However, the classical result of and-compositionality for definite logic programs (w.r.t. correct answers or other observables) says that the semantics of $\mathtt{G_1} \wedge \mathtt{G_2}$ can be derived from the semantics of $\mathtt{G_1}$ and $\mathtt{G_2}$. Since goals in definite programs become existentially quantified in our setting, we would like a relationship between $P \twoheadrightarrow \vec{\exists}.G_1 \wedge G_2$, $P \twoheadrightarrow \vec{\exists}.G_1$ and $P \twoheadrightarrow \vec{\exists}.G_2$. Unfortunately, this cannot be derived directly from (26). □

Note that $U_\mathcal{L}$ works with proofs with hypotheses. For this reason, it is not possible to retrieve only terminated computations using this fixpoint operator. This is not a flaw in the definition of the operator, but an intrinsic limit of all the kinds of top-down semantic refinements.

4 Abstraction Framework

The previous semantics are by far too detailed for most of the needs. However, it is now possible to use the techniques of abstract interpretation [11] to develop a range of abstract semantics for sequent calculi. We begin by defining the fundamental concept of *observable*.

Definition 6 (Observable). *An observable is a triple (D, α, γ) where D (the* abstract domain*) is an ordered set w.r.t. the relation $\sqsubseteq$ and $\alpha : \mathsf{Int} \to D$ (the* abstraction function*) is a monotonic function with γ as right adjoint.*

Since α and γ in (D, α, γ) uniquely determine each other [12], we will often refer to an observable just by the abstraction function.

An abstract interpretation for a logic $\mathcal{L}$ is an element of the abstract domain D. Given an interpretation I, it is possible to define an abstract counterpart $\alpha(I)$. Hence, it is possible to define abstract denotational, operational and fix-point semantics as the abstractions of the corresponding concrete semantics. The question is whether it is possible to derive such abstract semantics working entirely in the abstract domain.

Example 7. Given a logic $\mathcal{L}$, take as abstract domain D_s the powerset of all the sequents with the standard ordering, and as abstraction function the following

$$\alpha_s(I) = \{S \mid \exists p \in I.\ \mathrm{th}(p) = S \text{ and } \mathrm{hyp}(S) = \emptyset\} \ . \tag{28}$$

The right adjoint of α is the function

$$\gamma_s(A) = \{p \mid \mathrm{hyp}(S) \neq \emptyset \text{ or } \mathrm{th}(p) \in A\} \ . \tag{29}$$

We call $(D_s, \alpha_s, \gamma_s)$ the observable of *success sets*, since it abstracts a set of proofs in the set of the theorems they prove. □

4.1 Abstract Semantic Operators

The only two operators we use in the specification of the concrete semantics are union and gluing. Once we define an abstraction, we have an abstract operator $\cup_\alpha$ correct w.r.t. $\cup$, defined as

$$\bigcup\nolimits_\alpha \{A_j \mid j \in J\} = \alpha\left(\bigcup\{\gamma(A_j) \mid j \in J\}\right) \ . \tag{30}$$

In general, $\cup_\alpha$ is the least upper bound of those elements in D which are the image of some interpretation I. Moreover, it is a complete operator, i.e.

$$\bigcup\nolimits_\alpha \{\alpha(I_j) \mid j \in J\} = \alpha\left(\bigcup\{I_j \mid j \in J\}\right)$$

for each collection $\{I_j\}_{j \in J}$ of interpretations.

We could define an abstract operator $\rhd_\alpha$ correct w.r.t. $\rhd$ as done for $\cup_\alpha$ in (30). However, $\rhd$ is never used in all its generality. Hence we prefer to consider the optimal abstract counterparts of the two unary operators $I \mapsto \mathcal{R} \rhd I$ and $I \mapsto I \rhd (R \cup \epsilon)$. We define

$$\mathcal{R} \rhd_\alpha A = \alpha(\mathcal{R} \rhd \gamma(A)) \ , \tag{31}$$

$$A \rhd_\alpha (\mathcal{R} \cup \epsilon) = \alpha(\gamma(A) \rhd (\mathcal{R} \cup \epsilon)) \ . \tag{32}$$

When either $\mathcal{R} \rhd_\alpha A$ or $A \rhd_\alpha (R \cup \epsilon)$ is complete, we say that the observable is respectively *denotational* or *operational*, following the terminology introduced in [2]. If, for each inference rule $r \in \mathcal{R}$, there is an abstract operator $\tilde{r}$ correct

w.r.t. r as defined in (6), a correct abstract operator for $\mathcal{R} \rhd A$ can be defined as

$$\mathcal{R} \tilde{\rhd} A = \bigcup_{r \in \mathcal{R}}^{\alpha} \tilde{r}(A) \ . \tag{33}$$

Moreover, if all the $\tilde{r}$'s are optimal or complete, the same holds for (33).

Example 8. With respect to the observable α_s, consider an inference rule $r : S_1, \ldots, S_n \vdash S$. The corresponding optimal abstract operator r_{α_s} is given by

$$r_{\alpha_s}(X_1, \ldots, X_n) = \begin{cases} \{S\} & \text{if } S_i \in X_i \text{ for each } i = 1 \ldots n \\ \emptyset & \text{otherwise} \end{cases} \tag{34}$$

and it can be proved to be complete. Then, it turns out that the observable of success sets is denotational. An observable which is both operational and denotational is that of *plain resultants*, defined as

$$\alpha_r(I) = \{\langle S, (S_1, \ldots, S_n)\rangle \mid \exists p \in I.\ p : S_1, \ldots, S_n \vdash S\} \ . \tag{35}$$

with the obvious ordering by subsets. Note that what is generally called resultant is the reduced product [12] of α_r and the observable of computed answers. □

4.2 Pre-interpretations and Observables

By means of the observables we want to recover the great generality given by the use of pre-interpretations, but in a more controlled way, in order to simplify the definition and comparison of different semantics.

Given a logic $\mathcal{L}$ and an observable (D, α, γ), we have a corresponding pre-interpretation $\mathcal{I}_\alpha$ given by

- $\mathcal{I}_\alpha(S) = \langle \{x \in D \mid x \sqsubseteq \alpha(\mathsf{Sch}_S)\}, \sqsubseteq \rangle$, where $\sqsubseteq$ is the ordering for D;
- $\mathcal{I}_\alpha(r) = \alpha \circ r \circ \gamma$.

The idea is that, with the use of pre-interpretations, we break an abstract interpretation in pieces, each one relative to a single sequent. If A is an abstract interpretation, a corresponding interpretation $[\![\_]\!]$ w.r.t. $\mathcal{I}_\alpha$ is

$$[\![S]\!]_\alpha = A \cap_\alpha \alpha(\mathsf{Sch}_S) \ , \tag{36}$$

for each sequent S, where $\cap_\alpha$ is the optimal abstract operator which is correct w.r.t. $\cap$. On the other side, given $[\![\_]\!]_\alpha$, we have the abstract interpretation

$$A = \bigcup\nolimits_\alpha \{[\![S]\!]_\alpha \mid S \text{ is a sequent}\} \ . \tag{37}$$

However, in general, (36) and (37) do not form a bijection. Actually, an interpretation w.r.t. $\mathcal{I}_\alpha$ always keeps separate the semantics for different sequents, while the same does not happen for abstract interpretations.

Example 9. Consider the observable (D, α, γ) where $D = \{\mathsf{true}, \mathsf{false}\}$, with $\mathsf{false} \sqsubseteq \mathsf{true}$ and

$$\alpha(I) = \begin{cases} \mathsf{true} & \text{if } \exists p \in I.\ \mathrm{hyp}(p) = \emptyset \\ \mathsf{false} & \text{otherwise} \end{cases} \tag{38}$$

The corresponding pre-interpretation $\mathcal{I}_\alpha$ is the same as the one defined in Example 4. Given the interpretation $[\![\_]\!]$ such that $[\![\bar{S}]\!] = \mathsf{true}$ for a given sequent $\bar{S}$ and $[\![S]\!] = \mathsf{false}$ for each $S \neq \bar{S}$, the composition of (36) and (37) is the interpretation $[\![\_]\!]'$ such that

$$[\![S]\!]' = \left(\bigcup\nolimits_\alpha \{ [\![S']\!] \mid S' \text{ is a sequent} \} \right) \cap_\alpha \mathsf{true} = \mathsf{true} \tag{39}$$

for each sequent S. □

Given an observable α, we say that it *separates sequents* when

- $\gamma(\alpha(\mathsf{Sch}_S)) = \mathsf{Sch}_S$ for each sequent S;
- $\gamma(\alpha(\bigcup_S X_S)) = \bigcup_S \gamma(\alpha(X_S))$ if $X_S \subseteq \mathsf{Sch}_S$ for each sequent S.

If α separates sequents, (36) and (37) form a bijection between the abstract interpretations which are in the image of α and the interpretations $[\![\_]\!]$ such that $[\![S]\!]$ is in the image of α for each sequent S. From this point of view, it seems that observables are even more general than pre-interpretations. On the other side, abstractions only cover a subset of all the pre-interpretations, those whose abstraction function has a right adjoint.

Example 10. It is easy to prove that α_s separates sequents. The corresponding pre-interpretation $\mathcal{I}_{\alpha_s}$ is isomorphic to the pre-interpretation $\mathcal{I}$ given in Example 4. Note that, thanks to abstract interpretation theory, we automatically obtain an optimal candidate for the abstract semantic functions from the choice of the abstract domain.

4.3 Abstract Semantics

We say that an abstract interpretation A is an *abstract model* when the corresponding interpretation $[\![\_]\!]_\alpha$ for $\mathcal{I}_\alpha$ given by (36) is a model. In formulas, this means that, for each inference rule $r : S_1, \ldots, S_n \vdash S$,

$$\alpha\left(r\left(\gamma(A \cap_\alpha \alpha(\mathsf{Sch}_{S_1})), \ldots, \gamma(A \cap_\alpha \alpha(\mathsf{Sch}_{S_n}))\right)\right) \sqsubseteq A \cap_\alpha \alpha(\mathsf{Sch}_S)\ . \tag{40}$$

In turn, this is equivalent to say that $\gamma(A)$ is a syntactic model.

We would like to define the *abstract declarative semantics* $\mathcal{D}_\alpha(\mathcal{L})$ as the least abstract model for $\mathcal{L}$. However, since our abstract domain is a poset, we are not guaranteed that such an element exists. Nevertheless, when we work with a denotational observable, we have:

- $\mathcal{D}_\alpha(\mathcal{L}) = \alpha(\mathcal{D}(\mathcal{L}))$, where $\mathcal{D}_\alpha(\mathcal{L})$ is the least abstract model;

– $\mathcal{D}_{c,\alpha}(\mathcal{L}) = \alpha(\mathcal{D}_c(\mathcal{L}))$, where $\mathcal{D}_{c,\alpha}(\mathcal{L})$ is the least abstract model greater than $\alpha(\epsilon)$.

Other conditions, such as surjectivity of α, imply the existence of $\mathcal{D}_\alpha(\mathcal{L})$, whether or not α is denotational. However, in this case, we cannot be sure of the stated correspondence with $\alpha(\mathcal{D}(\mathcal{L}))$.

As in the concrete case, we want to recover $\mathcal{D}_\alpha(\mathcal{L})$ as the least fixpoint of a continuous operator. If the observable is denotational, we define by

$$T_{\mathcal{L},\alpha}(A) = A \cup_\alpha (\mathcal{R} \rhd_\alpha A) \ , \tag{41}$$

an abstract operator which is complete w.r.t. $T_{\mathcal{L}}$. Then, by well known results of abstract interpretation theory [12],

$$T_{\mathcal{L},\alpha} \uparrow \omega = \alpha(T_{\mathcal{L}} \uparrow \omega) = \mathcal{D}_\alpha(\mathcal{L}) \ , \tag{42}$$

$$T^\omega_{\mathcal{L},\alpha}(\alpha(\epsilon)) = \alpha(T^\omega_{\mathcal{L}}(\epsilon)) = \mathcal{D}_{c,\alpha}(\mathcal{L}) \ , \tag{43}$$

which are the required equalities.

Finally, let us come to the abstract operational semantics. In general, since we do not have an abstraction on the level of the single derivation, we can only abstract the collecting operational semantics given by $U_{\mathcal{L}}$. If $\rhd$ is operational, we define

$$U_{\mathcal{L},\alpha}(A) = A \rhd_\alpha (\mathcal{R} \cup \epsilon) \ , \tag{44}$$

which is a complete abstract operator w.r.t. $U_{\mathcal{L}}$. It is a well known result of abstract interpretation theory that

$$U^\omega_{\mathcal{L},\alpha}(\alpha(\epsilon)) = \alpha(U^\omega_{\mathcal{L}}(\epsilon)) = \alpha(D_c(\mathcal{L})) \ , \tag{45}$$

$$U^\omega_{\mathcal{L},\alpha}(\alpha(\{\epsilon_Q\})) = \alpha(U^\omega_{\mathcal{L}}(\{\epsilon_S\})) = \alpha(\mathcal{B}(\mathcal{L})_Q) \ . \tag{46}$$

Therefore, we have a top-down collecting construction of the abstract declarative semantics and of the operational behavior of $\mathcal{L}$.

Generally, if we replace the first equality with a "greater than" disequality in the equations (42), (43), (45) and (46), they become true for every observable α. In this case, the semantics computed in the abstract domain are correct w.r.t. the real abstract semantics.

5 Examples

Now that the theory is well established, we can focus our attention on its applications. We will recover two of the most common abstractions used in the field of logic programming, but working within the framework of the sequent calculus. The advantage is that our definitions do not depend on any computational procedure used to interpret the language. In turn, this makes it easier to extend the abstractions to different logic languages. Actually, the observables we are going to discuss are general enough to be applied to the full first-order intuitionistic

logic. The drawback of this approach is that observables like computed answers, which rely on a specific computational mechanism, are not directly expressible.

In the following, we will assume to work in the domain of first-order intuitionistic logic. This means that $\langle \mathcal{D}, \mathcal{G} \rangle$ is a first-order language, while Term and Var are the corresponding sets of first-order terms and variables. To simplify the notation, in the forthcoming discussions we assume that, in each sequent, there is at most one quantification for each bound variable.

Here is a summary of the inference rule schemas we use for the sequent calculus of first-order intuitionistic logic.

$$\frac{\Gamma_1, B, \Gamma_2, C \twoheadrightarrow D}{\Gamma_1, C, \Gamma_2, B \twoheadrightarrow D}\ \textit{interchange} \qquad \frac{\Gamma_1, B, B \twoheadrightarrow C}{\Gamma_1, B \twoheadrightarrow C}\ \textit{contraction}$$

$$\frac{}{\Gamma, B \twoheadrightarrow B}\ id \qquad \frac{}{\Gamma, \bot \twoheadrightarrow \bot}\ \mathsf{true}R \qquad \frac{\Gamma \twoheadrightarrow \bot}{\Gamma \twoheadrightarrow B}\ \bot R$$

$$\frac{\Gamma \twoheadrightarrow B}{\Gamma \twoheadrightarrow B \vee C}\ \vee R_1 \qquad \frac{\Gamma \twoheadrightarrow B}{\Gamma \twoheadrightarrow C \vee B}\ \vee R_2 \qquad \frac{\Gamma, B \twoheadrightarrow D \qquad \Gamma, C \twoheadrightarrow D}{\Gamma, B \vee C \twoheadrightarrow D}\ \vee L$$

$$\frac{\Gamma, B_1, B_2 \twoheadrightarrow C}{\Gamma, B_1 \wedge B_2 \twoheadrightarrow C}\ \wedge L \qquad \frac{\Gamma \twoheadrightarrow B \quad \Gamma \twoheadrightarrow C}{\Gamma \twoheadrightarrow B \wedge C}\ \wedge R$$

$$\frac{\Gamma \twoheadrightarrow B \quad \Gamma, C \twoheadrightarrow E}{\Gamma, B \supset C \twoheadrightarrow E}\ \supset L \qquad \frac{\Gamma, B \twoheadrightarrow C}{\Gamma \twoheadrightarrow B \supset C}\ \supset R$$

$$\frac{\Gamma, B[x/t] \twoheadrightarrow C}{\Gamma, \forall x.B \twoheadrightarrow C}\ \forall L \qquad \frac{\Gamma \twoheadrightarrow B[x/v]}{\Gamma \twoheadrightarrow \forall x.B}\ \forall R$$

$$\frac{\Gamma, B[x/v] \twoheadrightarrow C}{\Gamma, \exists x.B \twoheadrightarrow C}\ \exists L \qquad \frac{\Gamma \twoheadrightarrow B[x/t]}{\Gamma \twoheadrightarrow \exists x.B}\ \exists R$$

provided that the variable v does not occur in the lower sequents of the $\exists L$ and $\forall R$ schemas and B is an atomic formula in the id schema. When we want to denote a well defined inference rule, which is an instance of one of these schemas, we append appropriate indexes to the name of the schemas, like in $\exists R_{\Gamma, \exists z.\varphi, t(a)}$ for

$$\frac{\Gamma \twoheadrightarrow \varphi[z/t(a)]}{\Gamma \twoheadrightarrow \exists z.\varphi}\ .$$

5.1 Correct Answers

First of all, we want to extend the standard notion of correct answer for Horn clauses to the general case of first order intuitionistic logic. Given a goal `G` and a program `P` in pure logic programming, a correct answer θ is a function (substitution) from the variables in `G` to terms, with the interesting property that

$\vec{\forall}\mathtt{P} \twoheadrightarrow \mathtt{G}\theta$ is provable. Since the real logical meaning of evaluating $\mathtt{G}$ in a program $\mathtt{P}$ is that of proving the closed sequent $\vec{\forall}\mathtt{P} \twoheadrightarrow \vec{\exists}\mathtt{G}$, we can think of an extension of the concept of correct answer to generic sequents $\Gamma \twoheadrightarrow \varphi$ as a mapping from existentially quantified variable in φ to terms. We require that $\Gamma \twoheadrightarrow \varphi\{\theta\}$ is provable for an appropriate notion of substitution $\{\theta\}$.

However, note the following facts:

- if we only work with Horn clauses and a sequent $\Gamma \twoheadrightarrow \exists x.\varphi$ is provable, we know that there exists a term t such that $\Gamma \twoheadrightarrow \varphi[x/t]$ is provable. This is not true in the general case. Therefore, we can think of using partial functions mapping variables to terms, so that we can choose not to give an instance for some of the variables;
- consider the two sequents $S = \Gamma \twoheadrightarrow \exists x.\varphi$ and $S' = \Gamma \twoheadrightarrow (\exists x.\varphi) \supset \psi$. The role of the two existential quantifiers is completely different. In the first case we are actually looking for a term t to substitute into the x. In the second case, we are producing a new object a, forcing the fact that $\varphi[x/a]$ holds. To be more precise, in a proof for S, we introduce the formula $\exists x.\varphi$ with the rule $\exists R$ or $\bot R$, while in a proof for S' we introduce it by $\exists L$. As a consequence, we want to restrict our attention to the first kind of existential quantifiers.

Given a formula φ, a variable x is said to be a *query variable* for φ if the subformula $\exists x.\varphi'$ positively occurs in φ for some φ'. A *(candidate) answer* θ for φ is a function from the query variables of φ to Term such that

- if $\exists x.\varphi'$ positively occurs in φ, $\theta(x)$ does not contain any variable which is quantified in φ';
- θ is idempotent, i.e. its domain (the set of variables for which θ is defined) and range (the set of variables which occur in its image) are disjoint.

Let us point out that, when φ has no positive existentially quantified variables, it has only a trivial candidate answer.

Given an answer θ for φ, we define the *instantiation* $\varphi\{\theta\}$ of φ via θ by induction on the structure of the goals, as follows:

$$
\begin{array}{rll}
\bot\{\theta\} = \bot & & \\
A\{\theta\} = A & & \text{if } A \text{ is an atomic goal} \\
(\varphi' \oplus \varphi'')\{\theta\} = \varphi'\{\theta\} \oplus \varphi''\{\theta\} & & \text{for each binary logical symbol } \oplus \\
(\forall x.\varphi)\{\theta\} = \forall x.(\varphi\{\theta\}) & & \\
(\exists x.\varphi)\{\theta\} = \varphi[x/\theta(x)]\{\theta\} & & \text{if } \theta(x) \text{ is defined} \\
(\exists x.\varphi)\{\theta\} = \exists x.\varphi\{\theta\} & & \text{if } \theta(x) \text{ is undefined.}
\end{array}
$$

In other words, $\varphi\{\theta\}$ is obtained by replacing every existentially quantified subformula $\exists x.\varphi'$ in φ such that $\theta(x) \neq \bot$ with $\varphi'[x/\theta(x)]$.

An answer for φ is said to be a *correct answer* for the sequent $\Gamma \twoheadrightarrow \varphi$ when $\Gamma \twoheadrightarrow \varphi\{\theta\}$ is provable. Given the standard functional ordering $\leq$ for candidate answers, it is easy to check that, if θ is a correct answer for the sequent S and $\theta' \leq \theta$, then θ' is a correct answer, too. A correct answer for φ is *total* when its domain coincides with the set of query variables for φ.

Example 11. Given the goal $G = \forall x.\exists y.p(x,y)$, the answers $\theta = \{y \rightsquigarrow f(x)\}$, $\theta' = \{y \rightsquigarrow a\}$ and $\theta'' = \{\}$ give origin to the instantiated goals $G\{\theta\} = \forall x.p(x,f(x))$, $G\{\theta'\} = \forall x.p(x,a)$ and $G\{\theta''\} = G$. It turns out that θ and θ'' are correct answers for the sequent $\forall x.p(x,f(x)) \twoheadrightarrow G$.

Note that $\theta = \{x \rightsquigarrow y\}$ is not a candidate answer for $G = \exists x.\forall y.p(x,y)$, since y is a bound variable in $\forall y.p(x,y)$.

Assume we want to restrict ourselves to the fragment of Horn clauses. Let `P` be a pure logic program and let `G` be a definite goal. A correct answer θ (in the classical framework) for `G` in `P` is said *total* when it is idempotent and $\mathrm{dom}(\theta) = \mathrm{vars}(\texttt{G})$. Then, the two different definitions of total correct answers do coincide.

For example, let us consider the program `p(X,X)` and the goal `p(X,Y)`. The substitution {`X/Y`} is a (non total) correct answer in the classical setting, but $\{x/y\}$ is not a candidate answer for the sequent $\forall x.p(x,x) \twoheadrightarrow \exists x.\exists y.p(x,y)$. However, the equivalent correct answer {`X/Z,Y/Z`} is total, and corresponds to a correct answer in our setting, too.

5.2 Groundness

A first order term t is *ground* when it contains no variables. If θ is a candidate answer for φ, a variable x is ground in θ if $\theta(x)$ is ground. We also say that θ is *grounding* for x. A typical problem of static analysis is to establish which variables are forced to be ground in all the correct answers for a sequent S. There are many studies on this subject for the case of Horn clauses (see, for example, [4]), and some works for hereditary Harrop formulas, too (see [15]).

Given the set $\mathsf{Gr} = \{g, ng\}$, a *groundness answer* for a formula φ is a partial function β from the query variables of φ to Gr. Note that we do not assume any ordering between g and ng. Given a candidate answer θ for φ, we define a corresponding groundness answer $\alpha_{\mathrm{g}}(\theta)$, according to the following:

$$\alpha_{\mathrm{g}}(\theta)(x) = \begin{cases} \bot & \text{if } \theta \text{ is undefined in } x, \\ g & \text{if } \theta \text{ is grounding for } x, \\ ng & \text{otherwise} \end{cases} \tag{47}$$

If θ is a correct answer for S, then $\alpha_{\mathrm{g}}(\theta)$ is called a *correct groundness answer* for S. Given the obvious functional ordering for groundness answers, it turns out that if β is correct for S and $\beta' \leq \beta$, then β' is correct.

Example 12. Let us give some examples of sequents and their corresponding correct groundness answers:

sequent	**groundness answers**
$\forall y.p(y) \twoheadrightarrow \exists x.p(x)$	$\{x/g\}\ \{x/ng\}$
$\forall y.p(a,y) \wedge p(y,b) \twoheadrightarrow \exists x.p(x,x)$	$\{x/g\}$
$p(a) \vee r(b) \twoheadrightarrow \exists x.p(x) \vee r(x)$	$\{x/g\}$
$\bot \twoheadrightarrow \exists x.p(x)$	$\{x/g\}\ \{x/ng\}$
$\forall y.p(y,y) \twoheadrightarrow \forall x_1.\exists x_2.p(x_1,x_2)$	$\{x_1/ng\}$
$\forall y.p(y,y) \twoheadrightarrow \exists x_1.\exists x_2.p(x_1,x_2)$	$\{x_1/g, x_2/g\}, \{x_1/ng, x_2/ng\}$
$\exists y.p(y) \twoheadrightarrow \exists x.p(x)$	$\{x/\bot\}$
$p(t(a)) \twoheadrightarrow \exists x.p(r(x))$	$\emptyset$

Note that we only give the maximal correct groundness answers, according with the functional ordering.

We are interested in effectively computing the set of correct groundness answers for a given input sequent. Using the theory presented in this paper, we have developed a top-down analyzer for groundness, which works for the full intuitionistic first-order logic. It is based on the idea that, given a proof of the sequent S, it is possible to derive a groundness answer for S by just examining the structure of the proof. In particular if $p = r(p1, \ldots, p_n)$, it is:

$$\text{ganswer}(p)(x) = \begin{cases} g & \text{if } r = \exists R_{\Gamma,\exists x.\varphi,t} \text{ and } t \text{ is ground,} \\ ng & \text{if } r = \exists R_{\Gamma,\exists x.\varphi,t} \text{ and } t \text{ is not ground,} \\ \{\text{ganswer}(p_i)(x)\} & \text{if } r \neq R_{\Gamma,\exists x.\varphi,t} \text{ and } x \text{ appears in } p_i, \\ \bot & \text{otherwise.} \end{cases} \tag{48}$$

In general, if p is a final proof for S, we are not guaranteed that ganswer(p) is a correct groundness answer for S. For example, if $S = \exists x.t(x) \twoheadrightarrow \exists y.t(y)$ and p is the obvious corresponding final proof, it is ganswer$(p) = \{x/ng\}$, while the only correct answer is $\{x/\bot\}$. However, if β is a correct groundness answer for S, we can find a final proof p of S such that ganswer$(p) \geq \beta$. As a result, if I is the set of final proofs for S, then $\downarrow \{\text{ganswer}(p) \mid p \in I\}$ contains all the correct groundness answers for S. In the language of the theory of abstract interpretation, it means that $\downarrow \{\text{ganswer}(p) \mid p \in I\}$ is a correct approximation of the set of correct groundness answers.

Now, let us consider the function α_{t} which abstracts a formula φ with the same formula, where terms have been replaced by the set of variables occurring in them. We can trivially lift α_{t} to work with sequents.

If we name by $\langle \mathcal{D}', \mathcal{G}' \rangle$ the new language image of $\langle \mathcal{D}, \mathcal{G} \rangle$ via α_{t}, we can define a domain of *groundness with set resultants* D_{rg} such as

$$\begin{aligned} D_{rg} = \mathcal{P}_{\downarrow}\{\langle S, \beta, R\rangle \mid\ & S \text{ is a sequent in } \langle \mathcal{D}', \mathcal{G}' \rangle, \\ & R = \{S_1, \ldots, S_n\} \text{ is a finite set of sequents in } \langle \mathcal{D}', \mathcal{G}' \rangle, \\ & \beta \text{ is a groundness answer for } S\}\ . \end{aligned} \tag{49}$$

where $\mathcal{P}_{\downarrow}(X)$ is the set of downward closed subsets of X, ordered by

$$\langle S, \beta, R\rangle \leq \langle S', \beta', R'\rangle \text{ iff } S = S' \wedge \beta \leq \beta' \wedge R \supseteq R' \tag{50}$$

We can define an abstraction from syntactical interpretations to the domain of groundness with resultants as

$$\alpha_{\mathrm{rg}}(I) = \{\langle \alpha_{\mathrm{t}}(S), \beta, \{\alpha_{\mathrm{t}}(S_1), \ldots, \alpha_{\mathrm{t}}(S_n)\}\rangle \mid \text{there exists } p : S_1, \ldots, S_n \vdash S \text{ in } I \text{ with ganswer}(p) = \beta\} \tag{51}$$

We obtain an observable which can be effectively used for top-down analysis of groundness. The analyzer we have developed in `PROLOG` and which can be found at the URL `http://www.di.unipi.it/~amato/papers/sas2000final.pl` is an implementation of this observable, with some minor optimizations.

Example 13. By applying our analyzer to the sequents in the Example 12 we obtain precisely the same set of correct groundness answers, with the following exceptions:

sequent	**groundness answers**
$\exists y.p(y) \twoheadrightarrow \exists x.p(x)$	$\{x/g\}$
$p(t(a)) \twoheadrightarrow \exists x.p(r(x))$	$\{x/ng\}$

The previous example shows two different situations in which we lose precision. The first one is due to the fact that we abstract a term with the set of its variables, loosing the information about the functors. To solve this problem, the only solution is to improve our domain. The second situation arises from the interaction between positively and negatively occurring existential quantifiers, and can be addressed by improving the precision of the ganswer function. It should be possible to define a complete ganswer function, such that if p is a final proof for S, then ganswer(p) is a correct groundness answer for S. However, this involves examining the interaction between different quantifiers, and can possibly lead to a further generalization of the notion of correct answers, as a graph, linking quantifiers which produce "objects", introduced by $\forall R$ and $\exists L$, and quantifiers which consume "objects", introduced by $\forall L$ and $\exists R$.

If we restrict ourselves to Horn clauses logic, the abstraction function is quite precise, and we obtain a domain which, although expressed with a different formalism, has the same precision of $\mathcal{P}os$ [16, 10].

6 Conclusions and Future Works

The usefulness of a general semantic framework strictly depends on its ability to be easily instantiated to well known cases while suggesting natural extensions to them. In the case of a framework which we want to use as a reference for the development of procedures for static analyses, we also require that theoretical descriptions can be implemented in a straightforward way.

In this paper we presented a semantic framework for sequent calculi modeled around the idea of the three semantics of Horn clauses and around abstract interpretation theory. With particular reference to groundness and correct answers, we have shown that well known concepts in the case of Horn clauses can be obtained as simple instances of more general definitions valid for much broader logics. This has two main advantages. First of all, we can instantiate the general concepts to computational logics other then Horn clauses, such as hereditary Harrop formulas. Moreover, the general definitions often make explicit the logical meaning of several constructions (such as correct answers), which are otherwise obscured by the use of small logical fragments. We think that, following this framework as a sort of guideline, it is possible to export most of the results for positive logic programs to the new logic languages developed following proof-theoretic methods.

Regarding the implementation of static analyzers from the theoretical description of the domains, not all the issues have been tackled. While a top-down analyzer can often be implemented straightforwardly, like our interpreter for groundness, the same definitely does not hold for bottom-up analyzers. Since for a bottom-up analysis we have to build the entire abstract semantics of a logic, we need a way to isolate a finite number of "representative sequents" from which the semantics of all the others can easily be inferred: it is essentially a problem of compositionality.

We are actually studying this problem and we think that extending the notion of a logic $\mathcal{L}$ with the introduction of some *rules for the decomposition of sequents* will add to the theoretical framework the power needed to easily derive compositional $T_{\mathcal{L}}$ operators, thus greatly simplifying the implementation of bottom-up analyzers.

Moreover, the problem of groundness analysis for intuitionistic logic could be further addressed. The precision we can reach with the proposed domain can be improved by refining the abstraction function, and the implementation of the analyzer could be reconsidered to make it faster. Finally, it should be possible to adapt the domain to work with intuitionistic linear logic.

We think that our approach to the problem of static analyses of logic programs is new. There are several papers focusing on logic languages other than Horn clauses [15] but, to the best of our knowledge, the problem has never been tackled before from the proof-theoretic point of view. An exception is [20], which, however, is limited to hereditary Harrop formulas and does not come out with any real implementation of the theoretical framework.

References

1. G. Amato. Uniform Proofs and Fixpoint Semantics of Sequent Calculi. DRAFT. Available at the following URL: `http://www.di.unipi.it/~amato/papers/`, 1999.
2. Gianluca Amato and Giorgio Levi. Properties of the lattice of observables in logic programming. In M. Falaschi and M. Navarro, editors, *Proceedings of the APPIA-GULP-PRODE'97 Joint Conference on Declarative Programming*, 1997.

3. J. M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.
4. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Proc. Static Analysis Symposium, SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 266–280. Springer-Verlag, 1994.
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. Antonio Brogi, Paolo Mancarella, Dino Pedreschi, and Franco Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, July 1994.
7. M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Information and Computation*, 1999. To appear.
8. M. Comini, G. Levi, and G. Vitiello. Modular abstract diagnosis. In *International Workshop on Tools and Environments for (Constraint) Logic Programming, ILPS'97 Postconference Workshop*, 1997.
9. M. Comini and M. C. Meo. Compositionality properties of *SLD*-derivations. *Theoretical Computer Science*, 211(1 & 2):275–309, 1999.
10. A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.
11. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.
12. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
13. S. K. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, editor, *Advances in logic programming theory*, pages 115–182. Clarendon Press, Oxford, 1994.
14. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987. Second edition.
15. F. Malésieux, O. Ridoux, and P. Boizumault. Abstract compilation of λProlog. In J. Jaffar, editor, *Joint International Conference and Symposium on Logic Programming*, pages 130–144, Manchester, United Kingdom, June 1998. MIT Press.
16. K. Marriott and H. Sondergaard. Abstract Interpretation of Logic Programs: the Denotational Approach. In A. Bossi, editor, *Proc. Fifth Italian Conference on Logic Programming*, pages 399–425, 1990.
17. D. Miller, F. Pfenning, G. Nadathur, and A. Scedrov. Uniform proofs as a foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
18. B. Möller. On the Algebraic Specification of Infinite Objects – Ordered and Cntinuous Models of Algebraic Types. *Acta Informatica*, 22:537–578, 1985.
19. G. Nadathur and D. Miller. An Overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programmiong Conference*, pages 810–827. MIT Press, 1988.
20. P. Volpe. Abstractions of uniform proofs. In M. Hanus and M. Rodriguez-Artalejo, editors, *Algebraic and Logic Programming, Proc. 5th International Conference, ALP '96*, volume 1139 of *Lecture Notes in Computer Science*, pages 224–237. Springer-Verlag, 1996.