Theoretical Foundations and Semantics of Logic Programming

Annalisa Bossi[1] and Maria Chiara Meo[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
bossi@dsi.unive.it
[2] Dipartimento di Scienze, Università di Chieti-Pescara
Viale Pindaro 42, 65127 Pescara, Italy
cmeo@unich.it

Abstract. The paper provides an overview of an approach to the semantics of (constraint) logic programs, whose aim is providing suitable theoretical bases for modeling observable properties of logic programs in a compositional way. The approach is based on the idea of choosing (either equivalence classes or abstractions of) sets of clauses as semantic domain and provides an uniform framework for defining different compositional semantics for logic programs, parametrically with respect to a given notion of observability. Since some observable properties have a natural definition which is dependent on the selection rule, the framework has been adapted to cope also with a suitable class of rules, which includes the leftmost selection rule. This provides a formal description of most of the observable properties of Prolog derivations and can therefore be viewed as reference semantics for Prolog transformation and analysis systems.

1 Introduction

The paper provides an overview of an approach of the semantics of (constraint) logic programming which push forward the *s-semantic* approach [26] developed about twenty years ago. The aim of such an approach was that of providing a suitable base for program analysis by means of a semantics which really captures the operational semantics of logic programs and thus permits to model properties which can be observed in an SLD-tree (observables). For instance, in [26] two programs are equivalent if for any goal G they return the same (up to renaming) computed answers. That doesn't hold for the least Herbrand model semantics, namely, there exist programs which have the same least Herbrand model, yet compute different answer substitutions. Several ad-hoc semantics modeling various observables have been defined. These include correct answer substitutions, computed answer substitutions, partial answers [25], OR-compositional correct answers [9,8], call patterns [33], proof trees and resultants [30].

In addition there are several semantics specifically designed for static program analysis, which can handle various observables such as types and groundness dependencies. The idea of this approach is to define a framework which collects all the informations on SLD-derivations (for example in terms of resultants) and that permits to define denotations modeling various observables (thus inheriting basic constructions and results).

The relevant information for specific applications can be extracted from such a *collecting semantics* by suitable abstractions.

The paper is organized as follows. In the next section we recall the basic notions and introduce the terminology used in the paper. In Section 3 we describe the observables and their associated equivalence relations considered in the paper. Sections 4 and 5 describe a first general semantics schema and its principal instances. In Section 6 we discuss how the previous results can be specialized for a suitable class of selection rules. In Section 7 we introduce a framework for constraint logic programs. Finally, in Section 8 we describe a framework for bottom-up abstract interpretation.

2 Preliminaries

2.1 Logic Programming

The reader is assumed to be familiar with the terminology of and the main results on the semantics of logic programs [43,1]. We briefly recall here few basic notions.

Throughout the paper we assume programs and goals defined on a first order language given by a signature consisting of a finite set F of *data constructors*, a finite set Π of *predicate symbols*, a denumerable set V of *variable symbols*. T denotes the set of terms built on F and V. Variable-free terms are called ground. If E is any syntactic object, $Var(E)$ and $Pred(E)$ denote the set of (free) variables and of predicates occurring in E, respectively. A substitution is a mapping $\vartheta : V \rightarrow T$ such that the set $dom(\vartheta) = \{X \mid \vartheta(X) \neq X\}$ (domain of ϑ) is finite; ε is the empty substitution: $dom(\varepsilon) = \emptyset$. If ϑ is a substitution and E is a syntactic expression, we denote by $\vartheta_{|E}$ the restriction of ϑ to the variables in $Var(E)$.

The composition $\vartheta\sigma$ of the substitutions ϑ and σ is defined as the functional composition. A substitution ϑ is idempotent if $\vartheta\vartheta = \vartheta$. A renaming is a (nonidempotent) substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} = \rho^{-1}\rho = \varepsilon$. The result of the application of the substitution ϑ to a term t is an *instance* of t denoted by $t\vartheta$. We define $t \preceq t'$ (t is more general than t') iff there exists ϑ such that $t\vartheta = t'$. A substitution ϑ is a grounding for t if $t\vartheta$ is ground and $Ground(t)$ denotes the set of ground instances of t. The relation $\preceq$ is a preorder and $\approx$ denotes the associated equivalence relation (variance). A substitution θ is a unifier of terms t_1 and t_2 if $t_1\theta = t_2\theta$ (where $=$ denotes syntactic equality). If two terms are unifiable then they have an idempotent most general unifier which is unique up to renaming. Therefore $\mathtt{mgu}(t_1, t_2)$ denotes any such an idempotent most general unifier of t_1 and t_2. All the above definitions can be extended to other syntactic objects in the obvious way.

We restrict our attention to idempotent substitutions, unless differently stated.

An *atom* A is an object of the form $p(t_1, \ldots, t_n)$, where $p \in \Pi$ and $t_1, \ldots t_n \in T$. A (definite) *clause* is a formula of the form $H :- A_1, \ldots, A_n$ with $n \geq 0$, where H (the *head*) and $A_1, \ldots, A_n$ (the *body*) are atoms. $:-$ and , denote logic implication and conjunction respectively, and all variables are universally quantified. If the body is empty the clause is a *unit clause*. A (positive) *program* is a finite set of definite clauses and a (positive) *goal* is a conjunction of atoms $A_1, \ldots, A_m$. The empty goal is denoted by $\Box$. $\mathcal{A}$ and $\mathcal{C}$ denote the sets of atoms and of clauses, respectively, while $\wp(S)$ denotes the powerset of a set S.

In the following $\mathbf{t}, \mathbf{X}$ denote tuples of terms and of *distinct* variables respectively, while $\mathbf{B}$ denotes a (possibly empty) conjunction of atoms.

The ordinal powers of a generic monotonic operator f on a complete lattice $(D, \leq)$ with bottom $\perp$ are defined as usual, namely $f \uparrow 0 = \perp$, $f \uparrow (\alpha + 1) = f(f \uparrow \alpha)$, for α successor ordinal and $f \uparrow \alpha = \mathit{lub}(\{f \uparrow \beta \mid \beta \leq \alpha\})$ if α is a limit ordinal.

The *Herbrand base* $\mathcal{B}_{\mathcal{P}}$ of a program P is the set of all ground atoms whose predicate symbols are in $\mathit{Pred}(P)$. An *Herbrand interpretation* I for a program P is any subset of the Herbrand base $\mathcal{B}_{\mathcal{P}}$. An Herbrand model for a program P is any Herbrand interpretation M which satisfies all the clauses of P. The intersection $\mathcal{M}(P)$ of all the Herbrand models of a (positive) program P is a model (least Herbrand model).

Definite clauses have a natural computational reading based on the resolution procedure. The specific resolution strategy called SLD can be described as follows. Let $\mathbf{G} = A_1, \ldots, A_m$ be a goal and $c = H : -\mathbf{B}$ be a (definite) clause. $\mathbf{G}'$ is derived from $\mathbf{G}$ and c by using ϑ iff there exists an atom A_j, $1 \leq j \leq m$ such that $\vartheta = \mathtt{mgu}(A_j, H)$ and $\mathbf{G}' = (A_1, \ldots, A_{j-1}, \mathbf{B}, A_{j+1}, \ldots, A_m)$. Given a goal $\mathbf{G}$ and a program P, an SLD-derivation (or simply a derivation) of $P \cup \mathbf{G}$ (of $\mathbf{G}$ in P) consists of a (possibly infinite) sequence of goals $\mathbf{G}_0, \mathbf{G}_1, \mathbf{G}_2, \ldots$ called resolvents, together with a sequence $c_1, c_2, \ldots$ of variants of clauses in P which are *renamed apart* (i.e. such that c_i does not share any variable with $\mathbf{G}_0, c_1, \ldots, c_{i-1}$) and a sequence $\vartheta_1, \vartheta_2, \ldots$ of idempotent mgu's such that $\mathbf{G} = \mathbf{G}_0$ and, for $i \geq 1$, each $\mathbf{G}_i$ is derived from $\mathbf{G}_{i-1}$ and c_i by using ϑ_i. An SLD-refutation of $P \cup \mathbf{G}$ is a finite SLD-derivation of $P \cup \mathbf{G}$ which has the empty clause $\Box$ as the last goal in the derivation.

Following [1], a *selection rule* R is a function which when applied to a "history" containing all the clauses and the mgu's used in the derivation $\mathbf{G}_0, \mathbf{G}_1, \ldots, \mathbf{G}_i$, returns an atom in $\mathbf{G}_i$ (the selected atom in $\mathbf{G}_i$). Given a selection rule R, an SLD-derivation is called via R if all the selections of atoms in the resolvents are performed according to R.

In the following $\mathbf{G} \overset{\vartheta}{\rightsquigarrow}_{P,R}^{*} \mathbf{B}$ denotes a finite SLD-derivation of $P \cup \mathbf{G}$ via selection rule R, which has length ≥ 0, where ϑ is the composition of the mgu's introduced and $\mathbf{B}$ is the last resolvent in the derivation. If R is omitted, we mean that any selection rule can be used (and the definition is independent from the selection rule). Moreover, when the length of the derivation is 0, we assume that $\vartheta = \varepsilon$ and $\mathbf{B} = \mathbf{G}$.

The computed answer substitution of a refutation $\mathbf{G} \overset{\vartheta}{\rightsquigarrow}_{P}^{*} \Box$ is the substitution obtained by the restriction of ϑ to the variables occurring in $\mathbf{G}$. $\mathbf{G} \xrightarrow{\vartheta}_P \Box$ will denote explicitly the refutation of $\mathbf{G}$ in P with computed answer substitution ϑ.

2.2 Galois Insertions and Abstract Interpretation

Abstract interpretation [19,20] is a theory developed to reason about the abstraction relation between two different semantics. The theory requires the two semantics to be defined on domains which are complete lattices. $(C, \preceq)$ (the concrete domain) is the domain of the concrete semantics, while $(A, \leq)$ (the abstract domain) is the domain of the abstract semantics. The partial order relations reflect an approximation relation. The two domains are related by a pair of functions α (abstraction) and γ (concretization), which form a Galois insertion.

(Galois insertion). Let $(C, \preceq)$ be the concrete domain and $(A, \leq)$ be the abstract domain. A Galois insertion $(\alpha, \gamma) : (C, \preceq) \rightarrow (A, \leq)$ is a pair of maps $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ such that α and γ are monotonic, for each $x \in C$, $x \preceq \gamma(\alpha(x))$ and for each $y \in A$, $\alpha(\gamma(y)) = y$.

Given a concrete semantics and a Galois insertion between the concrete and the abstract domain, we want to define an abstract semantics. The concrete semantics is the least fixpoint of a semantic function $F : C \rightarrow C$. The abstract semantic function $\tilde{F} : A \rightarrow A$ is correct if for all $x \in C$, $F(x) \preceq \gamma(\tilde{F}(\alpha(x)))$. F can be defined as composition of "primitive" operators. Let $f : C^n \rightarrow C$ be one such an operator and assume that $\tilde{f}$ is its abstract counterpart. Then $\tilde{f}$ is (locally) correct w.r.t. f if for all $x_1, ..., x_n \in C$, $f(x_1, ..., x_n) \preceq \gamma(\tilde{f}(\alpha(x_1), \ldots, \alpha(x_n)))$. The local correctness of all the primitive operators implies the global correctness. According to the theory, for each operator f, there exists an optimal (most precise) locally correct abstract operator $\tilde{f}$ defined as $\tilde{f}(y_1, \ldots, y_n) = \alpha(f(\gamma(y_1), \ldots, \gamma(y_n)))$. However the composition of optimal operators is not necessarily optimal. The abstract operator $\tilde{f}$ is precise if $\tilde{f}(\alpha(x_1), \ldots, \alpha(x_n)) = \alpha(f(x_1, \ldots, x_n))$. The above definitions are naturally extended to "'primitive" semantic operators from $\wp(C)$ to C.

3 Observables and Composition Operators

The concrete operational semantics of (logic) programs can be specified by means of a set of inference rules which specify how derivations are made and by defining which are the "observables" we are interested in. In pure logic programming, we can be interested in different observable properties such as successful derivations, finite failures, (partial) computed answer substitutions, etc. Therefore a program can have different concrete operational semantics depending on which properties are observed.

A given choice of the observable x induces an "observational" equivalence on programs. Namely $P \approx_x Q$ iff P and Q are observationally indistinguishable according to x. When also composition of programs is taken into account, for a given observable property we can obtain different equivalences depending on which kind of program composition we consider. Given an observable x and a syntactic program composition operator $\circ$, the induced equivalence $\approx_{(x,\circ)}$ is defined as follows. $P \approx_{(x,\circ)} Q$ iff for any program R, $P \circ R$ and $Q \circ R$ are observationally indistinguishable according to x (i.e. P and Q are observationally indistinguishable under any possible context allowed by the composition operator). A semantics $\mathcal{S}$ is correct wrt $(x, \circ)$, if $\mathcal{S}(P) = \mathcal{S}(Q)$ implies $P \approx_{(x,\circ)} Q$, for each logic programs P and Q. $\mathcal{S}(P)$ is fully abstract wrt $(x, \circ)$ when also the converse of the previous implication holds.

A semantic $\mathcal{S}$ is compositional wrt the program composition operator $\circ$, if the semantics of the composition of programs P and Q can be obtained from the semantics of P and the semantics of Q, i.e. if for a suitable composition operator f, $\mathcal{S}(P \circ Q) = f(\mathcal{S}(P), \mathcal{S}(Q))$.

If $\mathcal{S}$ is correct wrt x and is compositional wrt $\circ$, then $\mathcal{S}$ is also correct wrt $(x, \circ)$.

If we are concerned with the input/output behavior of programs we should just observe computed answers. However some semantic based techniques (such as program analysis, debugging and transformation), require to observe and take into account other features of the derivation, which make visible internal computation details. In principle, one could be interested in the complete information about the SLD-derivation, namely the sequences of goals, most general unifiers and variants of clauses. The resultants, introduced in [44] in the framework of partial evaluation, are a compact representation of the relation between the initial goal $\mathbf{G}$ and the current $\langle$*goal, substitution*$\rangle$ pair in a SLD-derivation of $\mathbf{G}$, where the *substitution* is the (restriction to $Var(\mathbf{G})$ of the) composition of the mgu's computed in the SLD-derivation from $\mathbf{G}$ to the current goal.

Definition 1. *Let P be a program and let R be a selection rule. $\mathbf{G}\vartheta :-\mathbf{B} \in \mathcal{C}$ is an R-computed resultant for $\mathbf{G}$ in P iff there exists a SLD-derivation via R such that $\mathbf{G} \overset{\vartheta}{\leadsto}_{P,R}{}^{*} \mathbf{B}$. Moreover Φ is a computed resultant of $\mathbf{G}$ in P if there exists a selection rule R such that Φ is an R-computed resultant for $\mathbf{G}$ in P.*

In the following, given the (R-)computed resultant $\mathbf{G}\vartheta :-\mathbf{B}$ for the goal $\mathbf{G}$, we will say that $\vartheta_{|\mathbf{G}}$ is the substitution associated to the resolvent $\mathbf{B}$.

Resultants are a logical representation, which is quite convenient to study transformation techniques of logic programs such as partial evaluation and Fold/Unfold [41,52]. In fact, since these transformations are based on unfolding, i.e. on the application of some SLD-derivation steps to the program clauses, their intermediate and final results and also their basic properties can be naturally expressed in terms of resultants. For example, in addition to the above mentioned use, resultants have been used in [4] to study loop checking mechanisms and in [24] to prove the correctness of a modular Unfold/Fold transformation system.

The resultants are the basic observables to introduce a semantic scheme in Section 4 which collects informations on SLD-derivations. We will then derive as instances of the scheme other semantics which model (in some cases compositionally) more abstract observables, formally defined in Definition 2. These observables are:

partial answers. (denoted by pa), which are the substitutions associated to a resolvent in any SLD-derivation, and correct partial answers (denoted by cpa), which are the substitutions associated to a resolvent in any SLD-refutation. The knowledge about partial answers is important for program analysis [11], to characterize the semantics of concurrent languages [25] and to characterize universal termination, which in turn is useful for the semantics of PROLOG [2,5]

call patterns. (denoted by pt), which are the atoms (procedure calls) selected in any SLD-derivation, and correct call patterns (denoted by cpt), which are the atoms (procedure calls) selected in any SLD-refutation. Call patterns make it possible to derive properties of procedure calls, which are clearly relevant to program optimization and play an important role in most program analysis frameworks based on abstract interpretation (see [22] for a broad overview).

computed answers. (denoted by ca), which are the substitutions associated to the last resolvent ($\Box$) in an SLD-refutation, and

successful derivations. (denoted by s), where we just observe successful termination.

In the following sections we will show, as instances of the general scheme, a semantics (in some cases compositional) for each one of the previous observables. Each semantics $\mathcal{F}_x$ is obtained by setting a parameter in the scheme in Section 4, according to the corresponding observational equivalence $\approx_x$. Moreover each $\mathcal{F}_x$ is correct wrt the corresponding $\approx_x$. In several cases also full abstraction is obtained.

We formally define now the observational equivalences that we will consider.

Computed answers and successful derivations are known to be independent from the selection rule. This property is based on the switching lemma [1] and on the fact that these observables are obtained from successful derivations, where all the atoms have been evaluated. This is not the case for partial answers and call patterns which therefore depend on the selection rule. We first consider only notions which are independent from the selection rule. Therefore, in the case partial answers and call patterns, we introduce the independence in the definition by considering any selection rule.

Definition 2. *Let P be a program, R be a selection rule and let $\mathbf{G}$ be a goal such that there exists a derivation $\mathbf{G} \overset{\gamma}{\leadsto}_{P,R}{}^* \mathbf{B}$.*

1. *ϑ is a R-partial answer for $\mathbf{G}$ in P iff $\vartheta = \gamma_{|\mathbf{G}}$,*
2. *ϑ is a correct R-partial answer for $\mathbf{G}$ in P iff $\vartheta = \gamma_{|\mathbf{G}}$ and $\mathbf{B}$ has a refutation in P,*
3. *A is a R-call pattern for $\mathbf{G}$ in P iff A is the atom selected by R in $\mathbf{B}$,*
4. *A is a correct R-call pattern for $\mathbf{G}$ in P iff A is the atom selected by R in $\mathbf{B}$ and $\mathbf{B}$ has a refutation in P.*

Moreover ϑ is a (correct) partial answer for $\mathbf{G}$ in P iff there exists a selection rule R such that ϑ is a (correct) R-partial answer for $\mathbf{G}$ in P. Analogously for (correct) call patterns.

Note that computed answers are a special case of (correct) partial answers.

The only notion of program composition (the *OR-composition*) we will consider in the following is a generalization of program union $\cup_\Omega$ defined in [8]. First an Ω-open program P is a (positive) program in which the predicate symbols belonging to the set Ω are considered partially defined in P. P can be composed with another program Q which may further specify the predicates in Ω. Such a composition is denoted by $\cup_\Omega$ and $P \cup_\Omega Q$ is defined only if the predicate symbols occurring in both P and Q are contained in Ω. When Ω contains all the predicate symbols of P and Q we get the standard $\cup$-composition, while if $\Omega = \emptyset$ the composition is allowed only on programs which do not share predicate symbols.

The combination of the above defined six observables with the composition operator gives six observational equivalences. We list below their definitions.

Definition 3. *Let P and Q be Ω-open programs, $\mathbf{G}$ be a goal and let W denote a program such that $P' = P \cup_\Omega W$ and $Q' = Q \cup_\Omega W$ are defined. Assume that $x \in \{s, ca, pa, cpa, pt, cpt\}$. Then $P \approx_{(\Omega,x)} Q$ iff i_x holds for any $\mathbf{G}$ and for any W, where the conditions i_x are defined as follows*

i_s**:** *$\mathbf{G}$ has a refutation in P' iff $\mathbf{G}$ has a refutation in Q',*
i_{ca}**:** *$\mathbf{G}$ has the same set of computed answers in P' and in Q',*
i_{pa} **(**i_{cpa}**):** *$\mathbf{G}$ has the same set of (correct) partial answers in P' and in Q',*
i_{pt} **(**i_{cpt}**):** *$\mathbf{G}$ has the same set of (correct) call patterns in P' and in Q'.*

The case $\Omega = \emptyset$ is equivalent to considering no composition at all and therefore in order to simplify the notation we will denote $\approx_{(\emptyset,x)}$ by $\approx_x$.

4 A General Semantic Scheme

The scheme which has been proposed in [30,31] is a generalization of the open semantics introduced in [9,8] to obtain compositionality wrt program union. The standard semantics based on atoms is not compositional wrt union of programs. Consider for instance the programs $P = \{q(a), p(X) : -r(X)\}$, $Q = \{q(a)\}$ and $R = \{r(a)\}$. The least Herbrand model semantics $\mathcal{M}(P)$ identifies P and Q, since $\mathcal{M}(P) = \mathcal{M}(Q) = \{q(a)\}$. However $\mathcal{M}(P \cup R) \neq \mathcal{M}(Q \cup R)$. In order to obtain the semantics of the union $P \cup R$ from those of the components, the semantics of P should contain also the information given by the clause $p(X) : -r(X)$. For this reason, the open semantics was then defined on domains containing equivalence classes of sets of clauses (called π-interpretations).

If we abstract from the specific equivalences in [9,8], the open semantics can be viewed as a semantic framework for correctly modeling $\approx_{(\circ,x)}$ equivalences. Similarly to what happens for least Herbrand model semantics [23] the semantics built on π-interpretations is a mathematical object which is defined in model-theoretic terms and which can be computed both by a top-down and a bottom-up construction. The link between the top-down and the bottom-up constructions is given by an unfolding operator [42], denoted by Γ.

In the following a π-interpretation is a $\sim$-equivalence class $[I]$ where $I \subseteq \mathcal{C}$. $\mathcal{I}$ is the set of all the π-interpretations and we define $\iota(I) = a$ where a is the renamed apart version of any element in $I \in \mathcal{I}$. All the definitions which use elements from $\mathcal{I}$ are parametric wrt an equivalence $\sim$. However, in the remaining of this section, we omit the $\sim$ index in order to simplify the notation.

The general semantics scheme in [31,30] is defined in terms of π-interpretations and hence parametrically wrt $\sim$. We give two equivalent characterizations. The top-down one has a definition in the style of an operational semantics, while the bottom-up one is based on the fixpoint of a general immediate consequences operator. Let us first define the top-down semantics $\mathcal{O}(P)$.

Definition 4 (Operational Semantic Scheme). *Let P be a program.* $\mathcal{O}(P) = [\{\Phi \in \mathcal{C} \mid \Phi$ *is a resultant for a goal of the form* $p(\mathbf{X})$ *in* $P\}] \in \mathcal{I}$.

Note that $\mathcal{O}(P)$ is a π-interpretation and it is the (equivalence class of the) set of all the resultants obtained from goals of the form $p(\mathbf{X})$ in P for any possible selection rule. In [7] the resultants are extended by collecting also sequences of clause identifiers in order to obtain the maximum amount of information on computations so to observe all the internal details of SLD-derivations. Moreover, by modifying $\mathcal{O}(P)$, it is possible to obtain semantics compositional w.r.t. other composition operators, as for example inheritance mechanisms [6].

The semantics $\mathcal{O}(P)$ can be obtained also by a fixpoint construction. The suitable immediate consequences operator can be defined in terms of an unfolding operator. To this aim, first it is necessary organize the set of π-interpretations in a lattice $(\mathcal{I}, \sqsubseteq)$ based on a suitable partial order relation $\sqsubseteq$. Second, an immediate consequences operator $\mathcal{T}_P$ is defined and proved monotonic and continuous on $(\mathcal{I}, \sqsubseteq)$. This allows us to define the fixpoint semantics $\mathcal{F}(P)$ for P as $\mathcal{F}(P) = \mathcal{T}_P \uparrow \omega$, which is proved equivalent to the operational semantics.

We require $\sim$ to be a congruence wrt infinite unions, i.e. if, for all $n \in N$, $I_n, J_n \subseteq \mathcal{C}$ and $I_n \sim J_n$, then $\bigcup_{n \in N} I_n \sim \bigcup_{n \in N} J_n$. Since $\sim$ is a congruence wrt infinite unions, given $X \subseteq \mathcal{I}$ we can define $\bigsqcup X = [\bigcup_{I \in X} \iota(I)]$ and for $I, J \in \mathcal{I}$, $I \sqsubseteq J$ if and only if $I \sqcup J = J$. The relation $\sqsubseteq$ is an ordering on $\mathcal{I}$ and $(\mathcal{I}, \sqsubseteq)$ is a complete lattice (with $\sqcup$ as lub and $[\emptyset]$ as the bottom element).

Let us introduce the basic syntactic operator Γ which will be used to construct the general immediate consequence operator $\mathcal{T}$. Given a program P and a set of clauses I, $\Gamma_P(I)$ generates all the clauses obtained by "partially" unfolding P wrt I, i.e. it generates also those clauses obtained by rewriting a (possibly empty) subset of the atoms in the bodies of clauses in P.

In the following Id_Ω be the set of clauses $\{p(\mathbf{X}) \text{ :-} p(\mathbf{X}) \mid p \in \Omega\}$.

$$\begin{aligned} \Gamma_P(I) = \{(A \text{ :-} \mathbf{D_1}, \ldots, \mathbf{D_n})\vartheta \mid\; & \exists \text{ a clause } A \text{ :-} B_1, \ldots, B_n \in P, \\ & \exists\, n \text{ renamed apart clauses in } I \cup Id_\Pi : \\ & H_1 \text{ :-} \mathbf{D_1}, \ldots, H_n \text{ :-} \mathbf{D_n}, \\ & \exists \vartheta = \mathtt{mgu}((B_1, \ldots, B_n), (H_1, \ldots, H_n))\}. \end{aligned}$$

Now, in order to define the fixpoint semantics we require that $\sim$ is a congruence wrt the Γ operator, i.e. if $I \sim J$, then for any program P, $\Gamma_P(I) \sim \Gamma_P(J)$. This restriction will guarantee the correctness of the definition of the general fixpoint semantics. $\mathcal{T}_P$ is defined simply as the semantic counterpart of the syntactic operator Γ_P.

Definition 5. *Let P be a program. Then $\mathcal{T}_P : \mathcal{I} \to \mathcal{I}$ is the function*

$$\mathcal{T}_P(I) = [\Gamma_P(\iota(I))].$$

$\mathcal{T}_P(I)$ is well defined, i.e. its definition is independent from the element chosen in the equivalence class I, because Γ is a congruence wrt $\sim$. Moreover $\mathcal{T}_P$ is continuous on $(\mathcal{I}, \sqsubseteq)$ and $\mathcal{T}_P \uparrow \omega$ is the least fixpoint of $\mathcal{T}_P$.

Definition 6 (Fixpoint Semantic Scheme). *Let P be a program.*

$$\mathcal{F}(P) = \mathcal{T}_P \uparrow \omega \in \mathcal{I}.$$

Because of the previously mentioned ability of Γ_P (and therefore of $\mathcal{T}_P$) to produce also the result of partial unfoldings, $\mathcal{F}(P)$ gives a bottom-up description of partial derivations, i.e. it contains also the intermediate results of non-terminated (and possibly non-terminating) computations. Indeed, no matter which specific $\sim$ equivalence is used, the equality of the top-down and the bottom-up constructions holds [30]. This general result simplifies the treatment in specific cases since it is usually easier proving the congruence requirements on $\sim$ rather than proving the stated equality.

Lemma 1 (Equivalence). *Let P be a program, $\sim$ be an equivalence on $\wp(\mathcal{C})$ which is a congruence wrt infinite unions and wrt the Γ operator. Then $\mathcal{F}(P) = \mathcal{O}(P)$.*

By instantiating $\sim$ to a specific equivalence $\sim_{(\circ,x)}$, which depends on the composition operator ($\circ$) and the observable (x), we can obtain suitable $\mathcal{T}_P$ operators and (equivalent operational and fixpoint) semantics for the corresponding $\approx_{(\circ,x)}$ equivalences.

When considering as $\sim$ the identity on $\wp(\mathcal{C})$ we obtain a kind of "collecting semantics" which correctly models resultants. The semantics modeling resultants is clearly correct wrt the equivalence induced by any notion of observability considered in the previous section. However, we are interested in defining, for specific observables, coarser $\sim$ equivalences in order to obtain a more (possibly fully) abstract semantics, while preserving the correctness.

In the following we will then introduce a suitable $\sim$-equivalence to obtain a correct (in some cases fully abstract) semantics for any $\approx$-equivalence considered in the previous section. The instances of the generic constructions $\mathcal{I}$, $\mathcal{T}$, $\mathcal{O}$ and $\mathcal{F}$, obtained by using a specific $\sim_i$-equivalence, will be denoted by $\mathcal{I}_i$, $\mathcal{T}^i$, $\mathcal{O}_i$ and $\mathcal{F}_i$, respectively. When the subscripts are omitted we mean that $\sim$ is the identity on $\wp(\mathcal{C})$.

5 Getting Instances from the General Schema

5.1 Computed Answers Substitutions and Successful Derivations

In this section we consider first the composition of programs which do not share predicates (i.e. $\Omega = \emptyset$). As previously discussed, this is the same as the case of no composition at all. Here the observables we are concerned with are computed answer substitutions and successful derivations. The induced equivalences on programs have been previously denoted by $\approx_{ca}$ and $\approx_s$. We first show that suitable definitions of $\sim_{ca}$ and $\sim_s$ allow us to obtain the s-semantics [26] and the least Herbrand model as instances of the scheme. Then we consider the relation of these semantics to $\approx_{ca}$ and $\approx_s$. Since here we are not concerned with compositions, it is sufficient to extract from each set of clauses I only the information given by the unit clauses contained in I. Two sets of clauses can then be considered equivalent if they contain the same unit clauses (up to variance). Moreover, in the case of successful derivations, we only need the information given by the ground instances of the clauses. We define then $\sim_{ca}$ and $\sim_s$ as follows.

Definition 7. *Let $I, J \subseteq \mathcal{C}$. $I \sim_{ca} J$ iff $I \cap \mathcal{A} = J \cap \mathcal{A}$. Moreover $I \sim_s J$ iff $Ground(I \cap \mathcal{A}) = Ground(J \cap \mathcal{A})$.*

$\sim_{ca}$ and $\sim_s$ are congruences wrt infinite unions and wrt the Γ operator and therefore, we obtain automatically from the scheme for any program P, $\mathcal{I}_{ca}$, $\mathcal{T}^{ca}$, $\mathcal{O}_{ca}$ and $\mathcal{F}_{ca}$, (analogously for $\sim_s$).

Let us first consider the instances of the general definitions obtained by using $\sim_{ca}$. For any $I \in \mathcal{I}_{ca}$, the set of unit clauses (modulo variance) of any element $\iota(I)$ can be considered the canonical representative of the equivalence class I. $\mathcal{T}_P^{ca}$ defined in terms of canonical representatives is essentially the immediate consequence operator T_P^{s-sem} originally defined in [26]. The s-semantics is the least fixpoint $T_P^{s-sem} \uparrow \omega$ of such an operator. As an obvious consequence, the s-semantics as originally defined is the canonical representative of $\mathcal{F}_{ca}(P)$ [31].
The strong completeness theorem in [26] shows that the s-semantics is fully abstract wrt $\approx_{ca}$. The mentioned correspondence with $\mathcal{F}_{ca}$ implies that $\mathcal{F}_{ca}(P)$ is fully abstract wrt $\approx_{ca}$ [31]. The same result was obtained in [35] using a proof theoretic approach.

Lemma 2. *Let P and Q be programs. Then $P \approx_{ca} Q$ iff $\mathcal{F}_{ca}(P) = \mathcal{F}_{ca}Q)$.*

Analogously, in the case of $\sim_s$, the canonical representative $\iota_s(J)$ of $J \in \mathcal{I}_s$ can be obtained by taking the ground instances of the unit clauses in $\iota(J)$. $\mathcal{T}_P^s$ defined in terms of canonical representatives is essentially the standard immediate consequence operator T_P [23]. Also in this case, the two formulations are equivalent and the least fixpoint of T_P (the least Herbrand model $\mathcal{M}(P)$) is the canonical representative of $\mathcal{F}_s(P)$ [31]. The mentioned correspondence between $\mathcal{M}(P)$ and $\mathcal{F}_s(P)$ implies that the latter semantics is fully abstract wrt $\approx_s$. More precisely the following holds [31].

Lemma 3. *Let P and Q be programs defined on a signature Σ which contains infinitely many constant symbols. Then $P \approx_s Q$ iff $\mathcal{F}_s(P) = \mathcal{F}_s(Q)$.*

5.2 Compositional Equivalences

We consider now equivalences obtained by considering $\cup_\Omega$ as composition operator. We first focus on computed answers as observable to obtain from the scheme the semantics which is correct wrt $\approx_{(\Omega,ca)}$. Finally we take into account successful derivations: by using an equivalence $\sim_{(\Omega,s)}$ based on weak subsumption equivalence [45], we obtain the semantics $\mathcal{F}_{(\Omega,s)}(P)$ which is fully abstract wrt $\approx_{(\Omega,s)}$.

A semantics correct wrt $\approx_{(\Omega,ca)}$. We show now the instance of the scheme $\mathcal{F}_{(\Omega,ca)}(P)$, which is compositional wrt $\cup_\Omega$ and correctly models computed answers, i.e. it is correct wrt $\approx_{(\Omega,ca)}$. A semantics with these features was already defined in [8] by using sets of clauses as interpretations. [31,30] show how such a semantics can be obtained from the general scheme.

We first define a syntactic equivalence $\simeq$ on (sets of) clauses which is correct wrt $\approx_{(\Omega,ca)}$ (for any Ω) and hence can be used to define π-interpretations for the compositional case when considering computed answers. A distinction can be made among the atoms in the body of a clause, by identifying those *relevant* atoms which can share variables with the head in a derivation, and those which cannot. Clearly, only the atoms of the first type can contribute to the answer computed in a derivation. The others can only be *tested* for their successful derivation, but their derivation cannot give any useful binding for the computed answer, since such an answer is always restricted to the variables in the goal. Hence the following.

Definition 8. *An atom B in the body of a clause c is called relevant if either it shares variables with the head of c or, inductively, it shares variables with another atom B' in the body of c which is relevant. The multiset of relevant atoms in c is denoted by $Rel(c)$.*

In the following $Set(M)$ denotes the set of the elements which appear in the multiset (or sequence) M. Moreover, when applied to multisets, $\subseteq$ denotes multiset inclusion.

Note that, in the following definitions relevant atoms in clause bodies are considered as multisets rather than sets. This is because in general a relevant atom in the body $\mathbf{B}$ of a clause cannot be deleted (even if a copy of the atom appear in $\mathbf{B}$) without changing the operational meaning of the clause in terms of computed answers. Recall that a clause $c_1 = H_1 :\!- \mathbf{A}$ subsumes a clause $c_2 = H_2 :\!- \mathbf{B}$ if there exists a substitution ϑ such that $H_1\vartheta = H_2$ and $Set(\mathbf{A})\vartheta \subseteq Set(\mathbf{B})$. Now, let c_1 and c_2 be two clauses which do not share variables and whose heads are H_1 and H_2, respectively. We say that $c_1 \leq_c c_2$ iff c_1 subsumes c_2 and there exists a renaming ρ such that $H_1\rho = H_2$, $Rel(c_2)\rho \subseteq Rel(c_1)$ and $Set(Rel(c_2)\rho) = Set(Rel(c_1))$.

The equivalence $\simeq$ is then defined as the symmetric closure of the Smith preordering induced on sets of clauses by $\leq_c$. It can be proved (see [31,30]) that $\simeq$ equivalent sets of clauses can be interchanged in any context while preserving the computed answer substitutions semantics. In fact, given $I, J \subseteq \mathcal{C}$, if $I \simeq J$ then the two sets of clauses are indistinguishable by $\approx_{(\Pi,ca)}$. We can then use $\simeq$ to define the equivalence $\approx_{(\Omega,ca)}$. Moreover, since $\cup_\Omega$ allows us to compose programs which share predicate symbols in Ω only, we only need the information given by clauses in C^Ω, where C^Ω denotes the set of clauses $H :- \mathbf{A}$ such that $Pred(\mathbf{A}) \subseteq \Omega$.

Definition 9. *Let $I, J \subseteq \mathcal{C}$. We define $I \simeq J$ iff for any $c \in I$ there exists $c' \in J$ such that $c' \leq_c c$ and vice versa. Moreover $I \sim_{(\Omega,ca)} J$ iff $I \cap C^\Omega \simeq J \cap C^\Omega$.*

It can be shown that $\sim_{(\Omega,ca)}$ is finer than (and hence correct wrt) $\approx_{(\Omega,ca)}$. $\sim_{(\Omega,ca)}$ is a congruence wrt infinite unions and wrt the Γ operator and therefore, we obtain automatically from the scheme for any program P, $\mathcal{I}_{(\Omega,ca)}$, $\mathcal{T}^{(\Omega,ca)}$, $\mathcal{O}_{(\Omega,ca)}$ and $\mathcal{F}_{(\Omega,ca)}$ by using $\sim_{(\Omega,ca)}$ as $\sim$.

Essentially the same results have been given in [9,8] by using the identity on $\wp(\mathcal{C})$ as $\sim_{(\Omega,ca)}$ equivalence.

Lemma 4. *Let P and Q be programs. If $\mathcal{F}_{(\Omega,ca)}(P) = \mathcal{F}_{(\Omega,ca)}Q$ then $P \approx_{(\Omega,ca)} Q$.*

The converse of the previous statement does not hold, i.e. the semantics $\mathcal{F}_{(\Omega,ca)}(P)$ is not fully abstract wrt $\approx_{(\Omega,ca)}$. The difficulty here is related to the use of clauses in the semantic domain (the full abstraction result in [34] was obtained using a domain not containing clauses).

A semantics correct and fully abstract wrt $\approx_{(\Omega,s)}$. Now we consider the usual program composition $\cup_\Omega$ but we will focus on successful derivations as observable. We will obtain from the general scheme a semantics $\mathcal{F}_{(\Omega,s)}(P)$ is fully abstract wrt $\approx_{(\Omega,s)}$.

According to the general construction, we have only to define a suitable equivalence $\sim_{(\Omega,s)}$ on clauses. First, note that the clause c is a tautology iff the body of c contains a copy of the head. Given $I, J \in \mathcal{C}$, we say that I and J are subsumption equivalent iff for any $c \in I$ there exists $c' \in J$ such that c' subsumes c and vice versa. I and J are weakly subsumption equivalent iff $I \backslash Taut(I)$ is subsumption equivalent to $J \backslash Taut(J)$, where $Taut(I)$ denotes the set of tautologies in I. Since here we are concerned only with successful derivations, $\sim_{(\Omega,s)}$ can simply be defined in terms of weak subsumption equivalence. Indeed, if c_1 subsumes c_2 then each successful derivation of a goal $\mathbf{G}$ can be performed by using c_1 instead of c_2. Moreover, if $\mathbf{G}$ has a successful derivation which uses the tautology c, $\mathbf{G}$ has also a derivation which does not use c. In other words, tautological clauses can be deleted. These remarks can be formalized as follows.

Definition 10. *Let $I, J \subseteq \mathcal{C}$. $I \sim_{(\Omega,s)} J$ iff $I \cap C^\Omega$ is weakly subsumption equivalent to $J \cap C^\Omega$.*

$\sim_{(\Omega,s)}$ is a congruence wrt infinite unions and wrt the Γ operator and therefore, we obtain automatically from the scheme for any program P, $\mathcal{F}_{(\Omega,s)}$ by using $\sim_{(\Omega,s)}$ as $\sim$. We have the following result.

Lemma 5. *Let P, Q be (finite) programs. $P \approx_{(\Omega,s)} Q$ iff $\mathcal{F}_{(\Omega,s)}(P) = \mathcal{F}_{(\Omega,s)}(Q)$.*

Note that the previous result holds also for infinite programs which contain only finitely many function symbols. It does not hold for generic infinite programs (for a counterexample consider the programs P and $Ground(P)$).

5.3 A Semantics for Partial Answers and Call Patterns

A fixpoint semantics for partial answers has been defined in [25]. [31,30] extend such a characterization by obtaining, from the general scheme, a fully abstract semantics for partial answers and a correct semantics for correct partial answers. Semantics for call patterns is also given.

We give just the intuition on how these semantics are obtained. More details can be found in the cited literature. For the sake of simplicity, we consider only the case $\Omega = \emptyset$. The compositional case can be obtained by using techniques similar to those used in the above section.

From the clauses in $\mathcal{F}(P)$ it is possible to extract the information needed to model partial answers and call patterns for any goal $\mathbf{G}$. For example, since each clause H :–$\mathbf{B}$ in $\mathcal{F}(P)$ corresponds to a derivation $p(\mathbf{X}) \overset{\beta}{\leadsto}_{P,R}{}^* \mathbf{B}$ (where $H = p(\mathbf{X})\beta$) ϑ is a partial answer for the goal $p(\mathbf{X})$ if there exists a clause H :–$\mathbf{B}$ in $\mathcal{F}(P)$ such that $\gamma = \mathtt{mgu}(p(\mathbf{X}), H)$ and $\vartheta = \gamma_{|p(\mathbf{X})}$. Moreover ϑ is a correct partial answer for $p(\mathbf{X})$ if there exists also a conjunction $\mathbf{C}$ containing atoms from $\mathcal{F}(P)$ such that $\mathbf{B}$ and $\mathbf{C}$ unify. This example can be extended to the general case in a obvious way.

Note that, when considering partial answers, we only need the information in the heads of the clauses in $\mathcal{F}(P)$, while for correct partial answers clearly we have to consider also bodies. In fact bodies contain the information needed to check if the partial derivation is part of a refutation. First of all, given $J \subseteq \mathcal{C}$, we define $Heads(J) = \{H \in \mathcal{A} \mid H$:–$\mathbf{B} \in J\}$ and therefore, according to the previous considerations the equivalences $\sim_{pa}$ and $\sim_{cpa}$ are defined as follows.

Definition 11. *Let $I, J \subseteq \mathcal{C}$. $I \sim_{pa} J$ iff $Heads(I \cup Id_{\Pi}) = Heads(J \cup Id_{\Pi})$. Moreover $I \sim_{cpa} J$ iff $I \sim_{(\Pi,ca)} J$.*

$\sim_{pa}$ and $\sim_{cpa}$ are congruences wrt infinite unions and wrt the Γ operator and therefore, the semantics for partial answers and correct partial answers can be automatically obtained as usual from the general scheme for any program P, by using $\sim_{pa}$ and $\sim_{cpa}$ as $\sim$, respectively. Moreover $\mathcal{F}_{pa}(P)$ is fully abstract wrt $\approx_{pa}$.

Lemma 6. *Let P, Q be programs. Then $P \approx_{pa} Q$ iff $\mathcal{F}_{pa}(P) = \mathcal{F}_{pa}(Q)$.*

For $\mathcal{F}_{cpa}(P)$ we have only the following correctness result. The problems for obtaining full abstraction here are the same as those mentioned for compositional computed answers.

Lemma 7. *Let P, Q be programs. If $\mathcal{F}_{cpa}(P) = \mathcal{F}_{cpa}(Q)$ then $P \approx_{cpa} Q$.*

The information needed to model call patterns can be obtained from the clauses in $\mathcal{F}(P)$ as well. For example, if H :–$B_1, \ldots, B_n \in \mathcal{F}(P)$ and $\vartheta = \mathtt{mgu}(A, H)$ then $B_i\vartheta$ is a call pattern for the goal A. Since we are not considering a specific selection rule, we only need the information on the relation between the head and the various atoms in the body. In other words, the clause $H :\!- B_1, \ldots, B_n$ is equivalent to the set of clauses $\{H :\!- B_1, \ldots, H :\!- B_n\}$. Therefore the following.

Definition 12. *Let* $c = H :\!- B_1, \ldots, B_n \in \mathcal{C}$. $Krom(c) = \{H :\!- B_1, \ldots, H :\!- B_n\}$.

The Krom operator, which transforms (equivalence classes of) clauses into sets of binary clauses, is extended in the obvious way to subsets of $\mathcal{C}$.

Definition 13. *Let* $I, J \subseteq \mathcal{C}$. $I \sim_{pt} J$ *iff* $Krom(I) = Krom(J)$.

$\sim_{pt}$ is a congruence wrt infinite unions and the operator Γ, therefore we have the usual definition of the semantics as instance of the scheme.

Definition 14. *(Call patterns semantics) Let* P *be a program. The semantics* $\mathcal{F}_{pt}(P)$ *for call pattern is defined as the instance of* $\mathcal{F}(P)$ *obtained by using* $\sim_{pt}$.

From the previous observations, we have the correctness results for the call pattern semantics.

Lemma 8. *Let* P, Q *be programs. If* $\mathcal{F}_{pt}(P) = \mathcal{F}_{pt}(Q)$ *then* $P \approx_{pt} Q$.

6 Introducing the Selection Rule

[32] shows how all the previous results can be specialized for a suitable class of selection rules. We discuss the idea of the specialization and give as an example the definition of the R-partial answer semantics. For the sake of simplicity, we consider only the case $\Omega = \emptyset$. The compositional case can be obtained by using techniques similar to those used in the Section 5.2.

First we focus on R-computed resultants, i.e. on those resultants which describe derivations which use the selection rule R. This provides a sort of collecting semantics which describes most of the observable properties of R-derivations. As in Section 4 a π-interpretation is a $\sim$-equivalence class $[I]$ where $I \subseteq \mathcal{C}$. $\mathcal{I}$ is the set of all the π-interpretations and we define $\iota(I) = a$ where a is the renamed apart version of any element in $I \in \mathcal{I}$. All the definitions which use elements from $\mathcal{I}$ are parametric wrt an equivalence $\sim$. However, in the remaining of this section, we omit the $\sim$ index in order to simplify the notation.

Definition 15 (Operational Semantic Scheme). *Let* P *be a program.* $\mathcal{O}_R(P) = [\{\Phi \in \mathcal{C} \mid \Phi$ *is a* R*-computed resultant for a goal of the form* $p(\mathbf{X})$ *in* $P\}] \in \mathcal{I}$.

The problems arise with the fixpoint definition. If we consider a generic selection rule, we cannot obtain a fixpoint (bottom-up) semantics equivalent to the operational one [32]. Therefore, in order to be able to reconstruct exactly the derivation from the bottom, [32] introduces the local rules, as specified by the following definition.

Definition 16 (Local rule). *Let* ϕ *be a given bijection on the set of integer numbers. A selection rule* R *is local, if it satisfies the following conditions:*

1. *if* $\mathbf{G} = A_1, \ldots, A_n$ *is the initial goal, then the atom selected by* R *in* $\mathbf{G}$ *is the atom* A_s*, such that* $\phi(s) < \phi(i)$ *for any* $i \in [1, n]$, $i \neq s$,

2. *if* $\mathbf{G}$ *is a generic resolvent, assume that* $A_1, \ldots, A_n$ *is the sequence of atoms in* $\mathbf{G}$ *introduced by the last derivation step. Then, as before, the atom selected is* A_s*, such that* $\phi(s) < \phi(i)$ *for any* $i \in [1, n]$*,* $i \neq s$*.*

Rules which select one of the most recently introduced atoms were called local in [40] and were studied since they produce SLD-trees with a simple structure, suitable for efficient searching techniques. Clearly the rules that we consider are also local in the sense of [40]. Note also that the PROLOG leftmost rule is local by defining ϕ as follows: $\phi(i) = i$.

It is possible to define a fixpoint semantics for R-computed resultants, where R is a local rule. Moreover, since the leftmost selection rule is a local rule, this semantics can therefore be viewed as a reference semantics for Prolog transformation and analysis systems, by setting R equal to the leftmost selection rule. Suitable abstractions of this semantics allow the characterization of observables useful for specific applications. We will consider explicitly the abstraction which gives a (fully abstract) semantics for partial answers.

The intuition behind the definition of the bottom-up semantics is the following. According to the previous definition, if A_j is the atom selected by a local rule R in the resolvent $A_1, \ldots, A_n$, then all the atoms derived from A_j are fully evaluated before the selection of the atoms A_i, $i \neq j$. Moreover a function ϕ is used to establish an ordering on the atoms of the query and of the clauses used in the derivation.

The ordering ϕ can then be used *locally* on the bodies of clauses in P, to establish how to rewrite the bodies (by using clauses in I in $\mathit{\Gamma}_{P,R}(I)$, see Definition 17). Namely, when considering a clause H :– $B_1, \ldots, B_n \in P$ in the definition of $\mathit{\Gamma}_{P,R}(I)$, we take any partition K, J of the indexes $\{1, \ldots, n\}$ such that $\phi(k) < \phi(j)$ for any $k \in K$ and $j \in J$. This means that any atom B_k, with $k \in K$, is fully evaluated before any B_j with $j \in J$, in any derivation which uses the clause H :– $B_1, \ldots, B_n$. Accordingly, the B_k's are unified with atoms in I. Moreover we consider an atom B_s such that $s \in J$ and the value of $\phi(s)$ is the minimum among the $\phi(j)$'s for $j \in J$. This means that B_s is the first atom selected after the evaluation of the B_k's has been completed. Since the evaluation of (the atoms derived by) B_s can also be not completed, B_s is unified with the head of a generic clause in I.

In order to simplify the notation, given a query $\mathbf{G} = A_1, \ldots, A_n$ and a set of indexes $K = \{k_1, \ldots, k_m\} \subseteq \{1, \ldots, n\}$, in the following we denote by $\mathbf{G}_K$ the query $A_{k_1}, \ldots, A_{k_m}$ and by $\mathbf{G}_{-K}$ the query obtained from $\mathbf{G}$ by deleting A_k for any $k \in K$.

Definition 17. *Let* P *be a program,* R *be a local selection rule and let* I *be a set of clauses.*

$$\begin{array}{ll} \mathit{\Gamma}_{P,R}(I) = \{(A :\!- \mathbf{D})\vartheta & \mid \exists\, A :\!- \mathbf{B} \in P \text{ with } \mathbf{B} = B_1, \ldots, B_n, \\ & \exists\, K \subseteq \{1, \ldots, n\},\ J = \{1, \ldots, n\} \setminus K, \\ & \exists s \in J, \text{ such that for any } k \in K \text{ and for any } j \in J \\ & \phi(k) < \phi(s) \leq \phi(j) \\ & \exists \text{ a sequence } \mathbf{H} \text{ of atoms in } I \text{ and} \\ & \exists \text{ a clause } H' :\!- \mathbf{B}' \text{ in } I \cup Id_{\Pi} \text{ such that} \\ & \vartheta = \mathtt{mgu}((\mathbf{B}_K, B_s), (\mathbf{H}, H')) \text{ and} \\ & \mathbf{D} \text{ is obtained from } \mathbf{B}_{-K} \text{ by replacing } B_s \text{ with } \mathbf{B}'\}. \end{array}$$

All the results shown in Section 4 hold also for this specialized version of the immediate consequence operator. In particular if $\sim$ is a congruence wrt infinite unions and the Γ operator, then $\mathcal{T}_{P,R} = [\Gamma_{P,R}(\iota(I))]$ (the semantic counterpart of the syntactic operator $\Gamma_{P,R}$) is well defined. Moreover $\mathcal{T}_{P,R}$ is continuous on $(\mathcal{I}, \sqsubseteq)$ and $\mathcal{F}_R(P) = \mathcal{T}_{P,R} \uparrow \omega$ (the least fixpoint of $\mathcal{T}_{P,R}$) is equal to the operational semantics $\mathcal{O}_R(P)$.

Now, we show as it is possible to model the R-partial answer semantics. For all the other observables it is possible to follows a similar construction.

Definition 18. *Let P and Q be programs. $P \approx_{pa,R} Q$ iff for any goal $\mathbf{G}$, $\mathbf{G}$ has the same set of R-partial answers in P and in Q.*

From $\mathcal{F}_R(P)$ it is possible to extract the R-partial answers as follows. Analogously to the case of partial answers (without considering the selection rule) in Section 5.3, since each clause H :$-\mathbf{B}$ in $\mathcal{F}_R(P)$ corresponds to a derivation $p(\mathbf{X}) \overset{\beta}{\rightsquigarrow}_{P,R}{}^{*}\ \mathbf{B}$ (where $H = p(\mathbf{X})\beta$) in order to model R-partial answer we only need keep the heads of the resultants and therefore, in the definition of $\sim_{pa,R}$, we can abstract from the bodies. However, we need to distinguish among partial answers those which are also computed answers, i.e. we need to distinguish between heads of non unit clauses and heads of unit clauses in $\mathcal{F}_R(P)$. Consider for example the goal $q(X), r(Y)$ and assume that R is the leftmost selection rule. If $X = a$ is a computed answer for $q(X)$ in the program P (i.e. if $\mathcal{F}_R(P)$ contains the unit clause $q(a)$) and $Y = b$ is a leftmost partial answer for $r(Y)$ in P, then $\{X = a, Y = b\}$ is a leftmost partial answer for $q(X), r(Y)$ in P. This in general is not the case if $X = a$ is a leftmost partial answer (and not a computed answer) for $q(X)$ (i.e. if $\mathcal{F}_R(P)$ contains a non unit clause $q(a)\ :-\mathbf{B}'$ and does not contain a unit clause $q(a)$).

According to the above considerations, the equivalences $\sim_{pa,R}$ is defined as follows.

Definition 19. *Let $I, J \subseteq \mathcal{C}$. $I \sim_{pa,R} J$ iff $Heads(I) = Heads(J)$ and $I \cap \mathcal{A} = J \cap \mathcal{A}$.*

$\sim_{pa,R}$ is a congruence wrt infinite unions and wrt the $\Gamma_{P,R}$ operator and therefore, we obtain automatically from the scheme $\mathcal{F}_{pa,R}$ by using $\sim_{pa,R}$ as $\sim$. We have the following result.

Lemma 9. *Let P, Q be programs and let R be a local selection rule. $P \approx_{pa,R} Q$ iff $\mathcal{F}_{pa,R}(P) = \mathcal{F}_{pa,R}(Q)$.*

7 A Semantic Scheme for Constraint Logic Programs

The *Constraint Logic Programming* paradigm CLP($\mathcal{X}$) (CLP for short) has been proposed by Jaffar and Lassez [38,37] in order to integrate a generic computational mechanism based on constraints with the logic programming framework. The benefits of such an integration are several. From a pragmatic point of view, CLP($\mathcal{X}$) allows one to use a specific constraints domain $\mathcal{X}$ and a related constraint solver within the declarative paradigm of logic programming. From the theoretical viewpoint, CLP provides a unified view of several extensions to pure logic programming (arithmetics, equational programming, object-oriented features, taxonomies) within a framework which preserves the unique semantic properties of logic programs, in particular the existence of equivalent operational, model theoretic and fixpoint semantics [38]. Moreover, since the computation is performed over the specific domain of computation $\mathcal{X}$, CLP($\mathcal{X}$) programs have an equivalent "algebraic" semantics [38] directly defined on the algebraic structure of $\mathcal{X}$.

[28] introduces a framework for defining various semantics, each corresponding to a specific observable property of computations, thus applying to the CLP case the methodology proposed in [7,31]. Analogously to the case of (standard) Logic Programming in Section 4, each semantics can be equivalently defined either operationally (top-down) or declaratively (bottom-up) as the least fixpoint of a suitable operator. The construction is based on a new notion of interpretation (which is a modified version of that given in Section 4), on a natural extension of the standard notion of truth and on the definition of various immediate consequences operators, whose least fixpoints on the lattice of interpretations are models corresponding to various observable properties. All the semantics defined in [38] can be reconstructed within the framework proposed in [28]. The main issue however is the definition of some new semantics and the investigation of their relation, in terms of correctness and full abstraction, wrt the program equivalences induced by various observable properties.

Some of the semantics considered in [28] are the generalization to the CLP case of the non-ground semantics for (positive) logic programs in [26] and of the compositional semantics in [8]. Indeed, most semantic constructions and results lift directly from logic programming to CLP. Moving to a non-ground semantics is even more natural in the case of CLP, since the computation structure may not even include constants so that there might be no "ground" objects.

In particular, [28] first defines a fully abstract semantics which characterizes computed answer constraints for constraint logic programs and then a semantics which models answer constraints and which is compositional wrt programs union. Such a semantics is the natural extension of the previous one obtained by using a semantic domain based on clauses.

Since the compositional semantics contains the "maximum" amount of information on computations, it can also be used to model other non-standard observable properties. Indeed suitable abstractions of this compositional semantics allow us to obtain a correct (in one case fully abstract) semantics for partial answer constraints and call patterns for constraint logic programs.

The definitions of the semantics are mainly interesting for their applications. Thus, the answer constraint semantics can be taken as the basis of a correct notion of program equivalence to be preserved by program transformation techniques. Suitable abstract versions of the immediate consequence operators introduced in [28] can be used for bottom-up abstract interpretation (i.e. fixpoint computation of the abstract model). More interestingly, the compositional semantics was used in [24] to develop a framework for the modular analysis of CLP programs. This is particularly relevant for practical applications where modularity can help to reduce the size and the complexity of the analysis. The semantics for partial answers and call patterns was used for the analysis of constraint logic programs too. For example, informations on partially computed constraints can be used to detect "independence" of (sub)goals [21], thus providing the conditions for optimizations of CLP programs based on AND-parallelism and intelligent backtracking.

8 A Semantic Scheme for Static Program Analysis

Static program analysis aims at determining properties of the behavior of a program without actually executing it. Static analysis is founded on the theory of abstract interpretation ([18]) for showing the correctness of analysis with respect to a given semantics. Thus, it is essentially a semantic-based technique and different semantic definition styles lead to different approaches to program analysis. In the field of logic programs we find two main approaches which correspond to the two main possible constructions of the semantics: top-down and bottom-up. The main difference between them is related to goal dependency. In particular, a top-down analysis starts with an abstract goal (see [10,39]), while the bottom-up approach (see [46,47]) determines an approximation of the success set which is goal independent. It propagates the information "bottom-up" as in the computation of the least fixpoint of the immediate consequences operator T_P.

Thanks to the equivalence between top-down and bottom-up constructions of the concrete semantics, by using an approach analogous to that given in Section 4, it is possible to get a goal independent top-down and bottom-up construction of the abstract model. This was the leading principle in the development of the framework for bottom-up abstract interpretation proposed in [3]. An instance of the framework consists in the specialization of a set of basic abstract operators like abstract unification, abstract substitution application and abstract union. By means of these abstract operators, [3] gives a bottom-up definition of an abstract model, i.e. a goal independent approximation of the concrete denotation. Different instances produce different analysis.

The concrete semantics considered in [3] is the semantics of computed answer substitutions. It is worth noticing that previous attempts [46,47], based on concrete semantics which do not contain enough information on the program behavior, failed on non-trivial analysis (like mode analysis). The problem was that they were too abstract to be useful to capture program properties like variable sharing or ground dependencies.

The ability to determine call patterns was also usually associated to goal dependent top-down methods. [11,29] showed that the choice of an adequate (concrete) semantics allows us to determine goal independent information on both partial answer substitutions and call patterns and that this information can be computed both top-down and bottom-up. This facilitates the analysis of concurrent logic programs (ignoring synchronization) and provides a collecting semantics which characterizes both successes and call patterns. Many other analysis had been defined based on a "non-ground Tp" semantics like groundness dependency analysis, depth-k analysis, and a "pattern" analysis to establish most specific generalizations of calls and success sets (see [12]). A similar methodology has been applied also to CLP programs [36], leading to a framework where abstraction simply means abstraction of the constraint system.

[14] builds upon the idea in [13] of providing an algebraic characterization of the observables. [14] extends the approach, by taking two basic semantics: a denotational semantics and a transition system which define SLD-derivations. In addition, the semantic properties of the observables are expressed as compositionality properties. This

leads to a more flexible classification of the observables, where it is possible reason about properties such as OR-compositionality and existence of abstract transition systems. Using abstract interpretation techniques to model abstraction allows us to state very simple conditions on the observables which guarantee the validity of several general theorems.

The idea is to define the denotational semantics and the transition system for SLD-derivations in terms of four semantic operators, directly related to the syntactic structure of the language. The observables are defined as Galois insertions and it is possible to characterize various classes of observables in terms of simple properties of the Galois insertion and of the basic semantic operators.

The reconstruction of an existing semantics or the construction of a new semantics in the framework requires just a few very simple steps.

1. First of all, we define an observable property domain, namely, a set of properties of derivations with an ordering relation which can be viewed as an approximation structure. An observation consists of looking at an SLD-derivation and extracting some property (abstraction). The formalization of the property o we want to model is a Galois insertion $\langle \alpha_o, \gamma_o \rangle$ between SLD-derivations and the property domain.
2. Once we have an observable o, we want to systematically derive the abstract semantics. The idea is to define the optimal abstract versions of the various semantic operators and then check under which conditions (on $\langle \alpha_o, \gamma_o \rangle$) we obtain the optimal abstract semantics. This will allow us to identify some interesting classes of observables and to assign the observable property to the right class of observables.
3. Depending on the class, we automatically obtain the new denotational semantics, transition system, top-down ($\mathcal{O}_{\alpha_o}(P)$) and bottom-up ($\mathcal{F}_{\alpha_o}(P)$) denotations (simply replacing the concrete semantic operators by their optimal abstract versions), together with several interesting theorems (equivalence, compositionality w.r.t. the various syntactic operators, correctness and minimality of the denotations).

Since it is based on standard operational and denotational semantic definitions, the framework can be adapted to other programming languages.

Finally [14] considers two classes of observables, *complete* and *approximate*. For every complete or approximate observable, the abstract operational semantics and the abstract denotational semantics are equivalent. This will allow us to define equivalent top-down and bottom-up analysis algorithms. The above equivalence property requires the observable to be condensing. Condensing is a compositionality property which tells that the abstract semantics of a procedure call can be derived (without losing precision) from the abstract semantics of the procedure declaration. This property is needed in abstract diagnosis [17,15,16] where the specification is a post-condition describing a (goal-independent) property of a set of procedure declarations. It is worth noting that the observables corresponding to the declarative semantics are condensing and that the declarative semantics do indeed characterize procedure declarations. Note also that several observables used in program analysis (for mode, type and groundness analysis) are also condensing and that a non-condensing observable can systematically be transformed into a (more concrete) condensing observable, by using domain refinement operators (see, for example, how the condensing domain $\mathcal{POS}$, for groundness analysis can be derived from the non-condensing domain $\mathcal{DEF}$ [50]). The results of the diagnosis for approximate observables are also valid for non-condensing domains, which are sometimes convenient to use in practice for efficiency reasons.

As expected from abstract interpretation theory, the difference between complete and approximate observables is related to precision. Namely, the abstract semantics coincides with the abstraction of the collecting semantics, in the case of complete observables, while it is just a correct approximation, in the case of approximate observables. On the other side, approximate observables correspond to noetherian domains. Hence their abstract semantics is finite, while (in general) it is infinite for complete observables. The class of complete observables includes the observables (ground instances of) computed answers and correct answers which allow us to reconstruct the declarative semantics used in declarative debugging, i.e., the least Herbrand model used in [51] and the least term model (atomic logical consequences or c-semantics) used in [27]. Moreover includes all the observable introduced in Section 3. On the other hand, the class of approximate observables includes $depth(k)$ [49] and several domains proposed for type, mode and groundness analysis (for example the domain $\mathcal{POS}$ [48] for groundness analysis).

Note that the AND-compositionality property (i.e., the compositionality with respect to the conjunction of atoms) of all the semantics defined by this approach, including their abstract versions, allows us to proceed in a goal independent way since we can obtain the result for any specific goal $\mathbf{G}$ just by executing $\mathbf{G}$ in $\mathcal{O}_{\alpha(o)}(P)$.

9 Conclusions

In the last twenty years, several semantics for logic programs had been developed according to an approach which push forward the s-semantics introduced by Moreno Falaschi, Giorgio Levi, Maurizio Martelli and Catuscia Palamidessi in [26]. The common aim was that of providing suitable theoretical bases for program analysis of different operational behaviors of logic programs. Each semantics captures properties which can be observed in an SLD-tree and is correct (in some cases fully abstract) wrt an equivalence relation induced by the considered property. We provided an overview of these semantics emphasizing their mutual relations and characteristics.

Acknowledgment

First of all we wish to thank Giorgio Levi, the main promoter of the s-semantics approach and together with him we wish to thank all those who have contributed to the development of this approach.

References

1. Apt, K.R.: Logic programming. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 493–574 (1990)
2. Barbuti, R., Codish, M., Giacobazzi, R., Maher, M.J.: Oracle semantics for Prolog. Information and Computation 122(2), 178–200 (1995)

3. Barbuti, R., Giacobazzi, R., Levi, G.: A general framework for semantics-based bottom-up abstract interpretation of logic programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 15(1), 133–181 (1993)
4. Bol, R.N., Apt, K.R., Klop, J.W.: An analysis of loop checking mechanisms for logic programs. Theor. Comput. Sci. 86(1), 35–79 (1991)
5. Bossi, A., Bugliesi, M., Fabris, M.: A new fixpoint semantics for Prolog. In: ICLP 1993: Proceedings of the Tenth Int'l Conference on Logic Programming, pp. 374–389. MIT Press, Cambridge (1993)
6. Bossi, A., Bugliesi, M., Gabbrielli, M., Levi, G., Meo, M.C.: Differential logic programming. In: POPL 1993: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 359–370 (1993)
7. Bossi, A., Gabbrielli, M., Levi, G., Martelli, M.: The s-semantics approach: theory and applications. Journal of Logic Programming 19(20), 149–197 (1994)
8. Bossi, A., Gabbrielli, M., Levi, G., Meo, M.C.: A compositional semantics for logic programs. Theoretical Computer Science 122(1-2), 3–47 (1994)
9. Bossi, A., Menegus, M.: Una semantica composizionale per programmi logici aperti. In: Sesto convegno sulla programmazione logica, pp. 95–109 (1991)
10. Bruynooghe, M.: A practical framework for the abstract interpretation of logic programs. Journal of Logic Programming 10(2), 91–124 (1991)
11. Codish, M., Dams, D., Yardeni, E.: Bottom-up abstract interpretation of logic programs. Theoretical Computer Science 124(1), 93–125 (1994)
12. Codish, M., Søndergaard, H.: Meta-circular abstract interpretation in Prolog, pp. 109–134 (2002)
13. Comini, M., Levi, G.: An algebraic theory of observables. In: SLP, pp. 172–186 (1994)
14. Comini, M., Levi, G., Meo, M.C.: Compositionality in sld-derivations and their abstractions. In: ILPS, pp. 561–575 (1995)
15. Comini, M., Levi, G., Meo, M.C., Vitiello, G.: Proving properties of logic programs by abstract diagnosis. In: Dam, M. (ed.) LOMAPS-WS 1996. LNCS, vol. 1192, pp. 22–50. Springer, Heidelberg (1997)
16. Comini, M., Levi, G., Meo, M.C., Vitiello, G.: Abstract diagnosis. Journal of Logic Programming 39(1-3), 43–93 (1999)
17. Comini, M., Levi, G., Vitiello, G.: Abstract debugging of logic program. In: Fribourg, L., Turini, F. (eds.) LOPSTR 1994 and META 1994. LNCS, vol. 883, pp. 440–450. Springer, Heidelberg (1994)
18. Cousot, P.: Program analysis: the abstract interpretation perspective. ACM Computing Surveys 28(4es), 165 (1996)
19. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
20. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
21. García de la Banda, M.J., Hermenegildo, M.V., Marriott, K.: Independence in constraint logic programs. In: ILPS, pp. 130–146 (1993)
22. Debray, S.K.: Formal bases for dataflow analysis of logic programs, pp. 115–182 (1994)
23. Van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. Journal of the ACM 23(4), 733–742 (1976)
24. Etalle, S., Gabbrielli, M.: Transformations of clp modules. Theor. Comput. Sci. 166(1&2), 101–146 (1996)
25. Falaschi, M., Levi, G.: Finite failures and partial computations in concurrent logic languages. Theor. Comput. Sci. 75(1&2), 45–66 (1990)

26. Falaschi, M., Levi, G., Martelli, M., Palamidessi, C.: Declarative Modeling of the Operational Behaviour of Logic Languages. Theoretical Computer Science 69, 289–318 (1989)
27. Ferrand, G.: Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method. Journal of Logic Programming 4(3), 177–198 (1987)
28. Gabbrielli, M., Dore, G.M., Levi, G.: Observable semantics for constraint logic programs. J. Log. Comput. 5(2), 133–171 (1995)
29. Gabbrielli, M., Giacobazzi, R.: Goal independency and call patterns in the analysis of logic programs. In: SAC, pp. 394–399 (1994)
30. Gabbrielli, M., Levi, G., Meo, M.C.: Observational equivalences for logic programs. In: Proceedings of the Joint Int'l Conference and Symposium on Logic Programming, pp. 131–145 (1992)
31. Gabbrielli, M., Levi, G., Meo, M.C.: Observable behaviors and equivalences of logic programs. Inf. Comput. 122(1), 1–29 (1995)
32. Gabbrielli, M., Levi, G., Meo, M.C.: Resultants semantics for prolog. J. Log. Comput. 6(4), 491–521 (1996)
33. Gabbrielli, M., Meo, M.C.: Fixpoint semantics for partial computed answer substitutions and call patterns. In: Kirchner, H., Levi, G. (eds.) ALP 1992. LNCS, vol. 632, pp. 84–99. Springer, Heidelberg (1992)
34. Gaifman, H., Shapiro, E.: Fully abstract compositional semantics for logic programs. In: POPL 1989: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 134–142. ACM Press, New York (1989)
35. Gaifman, H., Shapiro, E.: Proof theory and semantics of logic programs. In: Proceedings of the Fourth Annual Symposium on Logic in computer science, pp. 50–62. IEEE Press, Los Alamitos (1989)
36. Giacobazzi, R., Debray, S.K., Levi, G.: A generalized semantics for constraint logic programs. In: Proceedings of the Int'l Conference on Fifth Generation Computer Systems, pp. 581–591. ACM Press, New York (1992)
37. Jaffar, J., Lassez, J.-L.: Constraint logic programming. Technical report, Department of Computer Science, Monash University (June 1986)
38. Jaffar, J., Lassez, J.-L.: Constraint logic programming. In: POPL, pp. 111–119 (1987)
39. Janssens, G., Bruynooghe, M.: Deriving descriptions of possible values of program variables by means of abstract interpretation. Journal of Logic Programming 13(2-3), 205–258 (1992)
40. Kawamura, T., Kanamori, T.: Preservation of stronger equivalence in unfold/fold logic program transformation. Theor. Comput. Sci. 75(1&2), 139–156 (1990)
41. Komorowski, H.J.: A specification of an Abstract Prolog Machine and Its Applications to Partial Evaluation. Phd thesis, Linköping University (1981)
42. Levi, G.: Models, unfolding rules and fixpoint semantics. In: Proc. of the Fifth Int'l Conference and Symposium on Logic Programming, vol. 2, pp. 1649–1665. MIT Press, Cambridge (1991)
43. Lloyd, J.W.: Foundations of logic programming. Springer, New York (1984)
44. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. J. Log. Program. 11(3&4), 217–242 (1991)
45. Maher, M.J.: Equivalences of logic programs. In: Foundations of Deductive Databases and Logic Programming, pp. 627–658 (1988)
46. Marriott, K., Søndergaard, H.: Bottom-up abstract interpretation of logic programs. In: Proc. Fifth Int'l Conf. on Logic Programming, pp. 733–748. MIT Press, Cambridge (1988)
47. Marriott, K., Søndergaard, H.: Semantics-based dataflow analysis of logic programs. In: IFIP Congress, pp. 601–606. North-Holland, Amsterdam (1989)
48. Marriott, K., Søndergaard, H.: Precise and efficient groundness analysis for logic programs. LOPLAS 2(1-4), 181–196 (1993)

49. Sato, T., Tamaki, H.: Enumeration of success patterns in logic programs. Theor. Comput. Sci. 34, 227–240 (1984)
50. Scozzari, F.: Logical optimality of groundness analysis. Theor. Comput. Sci. 277(1-2), 149–184 (2002)
51. Shapiro, E.Y.: Algorithmic Program Debugging. MIT Press, Cambridge (1983)
52. Tamaki, H., Sato, T.: Unfold/fold transformation of logic programs. In: ICLP, pp. 127–138 (1984)