Theoretical Computer Science 410 (2009) 4692–4703

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

S-semantics for logic programming: A retrospective look

Annalisa Bossi

Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy

ARTICLE INFO

Keywords:
Logic programming
Semantics

ABSTRACT

The paper provides an overview of the s-semantic approach to the semantics of logic programs which had been 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 behavior of logic programs, and thus offers useful notions of observable program equivalences. The semantics is given in terms of extended interpretations, which are more expressive than Herbrand interpretations, extends the standard Herbrand semantics, and can be obtained as a result of both top-down and bottom-up constructions. The approach has been applied to several extensions of positive logic programs and used to develop semantic-based techniques for program analysis, verification and transformation.

1. Introduction

In the springtime of 1990 Giorgio Levi visited my Department in Padova. We met many times before, but never worked together. We were sitting in my room when I started saying: "I am working on the definition of an OR-compositional semantics for logic programs. It turns out to be an extension of your s-semantics, all the results can be easily …". I could not finish the sentence. "I know exactly what you are doing", he said immediately, "yes, I like it". That was the beginning of a productive and very stimulating collaboration on *the s-semantic approach to the semantics of logic programs*.

This terminology had been coined in [8] to identify many different semantics for logic programs developed between the end of 1980s and the beginning of 1990s. All these proposals are ruled by the same convincements: a semantics should help understanding the meaning of programs by providing useful notions of observable program equivalences. Each semantics in the approach captures some observable properties of logic programs and allows us to detect when two programs cannot be distinguished by observing their behaviors. That provides a suitable base for program analysis and transformation.

The first example of a semantic construction in this class is the *s-semantics* introduced by Falaschi, Levi, Martelli and Palamidessi in [26] to model the computed answer observational equivalence. According to this observable two programs are equivalent if for any goal G they return the same (up to renaming) computed answers. That does not hold for the least Herbrand model semantics, namely, there exist programs which have the same least Herbrand model, yet compute different answer substitutions.

There exists another problem with the least Herbrand model semantics: it is not compositional with respect to the union operation. More specifically, we cannot obtain the semantics of the (syntactic) union of two programs by (semantically) composing the least Herbrand models of the two components. The compositional semantics for positive logic programs introduced in [9,10] extends the s-semantics and provides a refined notion of observational equivalence which takes into account both computed answers and program composition, i.e., two programs P_1 and P_2 are equivalent if for any goal G and any program Q, the programs $P_1 \cup Q$ and $P_2 \cup Q$ return the same (up to renaming) computed answers. Other observable properties can be considered like *call patterns* [34] or *resultants* [32], as well as different composition operators. Observable and composition are the two dimensions which establish different equivalences and different semantics.

E-mail address: bossi@dsi.unive.it.

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.07.039

All these semantics are built on syntactic domains consisting of (equivalence classes of) syntactic objects. Syntactic domains make it possible to define a unique denotation also when there exists no unique representative Herbrand model. These program denotations capture various computational aspects in a goal independent way. Goal independence is a key issue. It means that denotations are defined by collecting the observable properties starting with the most general atomic goals and that they give a complete characterization of the program behavior for any goal.

The paper is organized as follows. In the next section we recall the basic notions, introduce the terminology used in the paper, and describe the general approach. In Section 3, we introduce its more significant instances: the original s-semantics [26,27], its compositional extension [9] and the collecting semantics of resultants [34,32] which gives the maximum amount of information on computations. Finally, Section 5 shows some applications of the approach in the areas of semantics-based analysis and program transformation.

2. A general overview

Similarly to what happens for standard Herbrand model semantics [23], all the semantics in the s-semantics approach admit three equivalent constructions: operational, fixpoint and model-theoretic. In this section we present these constructions in a general language independent way. First we recall some basic notions of logic languages and introduce the terminology used in the paper.

2.1. Basic notions

The reader is assumed to be familiar with the terminology of and the main results in the semantics of logic programs [47,1]. We briefly recall here few basic notions.

Let $\mathcal{L}$ be a first order language whose signature consists of a set C of data constructors, a finite set $\mathcal{P}$ of predicate symbols, a denumerable set V of variable symbols. Let $\mathcal{T}$ be the set of terms built on C and V. Variable-free terms are called ground. A substitution is a mapping $\vartheta : V \to \mathcal{T}$ such that the set $\mathcal{D}(\vartheta) = \{X \mid \vartheta(X) \neq X\}$ (domain of ϑ) is finite; ε denotes the empty substitution: $\mathcal{D}(\varepsilon) = \emptyset$. If W is a set of variables, we denote by $\vartheta_{|W}$ the restriction of $\mathcal{D}(\vartheta)$ to the variables in W, i.e., $\vartheta_{|W}(X) = \vartheta(X)$ if $X \in W$, and $\vartheta_{|W}(X) = X$ if $X \notin W$. Moreover if E is any syntactic object, *Var(E)* denotes the set of variables occurring in E and we use the abbreviation $\vartheta_{|E}$ to denote $\vartheta_{|Var(E)}$. The composition $\vartheta\gamma$ of the substitutions ϑ and γ is defined as the functional composition. Therefore a substitution ϑ is idempotent if $\vartheta\vartheta = \vartheta$. A renaming is a substitution ρ for which there exists the inverse ρ^{-1} such that $\rho\rho^{-1} = \rho^{-1}\rho = \varepsilon$. The pre-ordering $\preccurlyeq$ (more general than) on substitutions is such that $\vartheta \preccurlyeq \sigma$ iff there exists γ such that $\vartheta\gamma = \sigma$. The result of the application of the substitution ϑ to a term t is an instance of t denoted by $t\vartheta$. We define $t \preccurlyeq t'$ (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. The relation $\preccurlyeq$ is a pre-order and $\approx$ denotes the associated equivalence relation (variance). A substitution θ is a unifier of terms t_1 and t_2 if $t_1\theta = t_2\theta$. $\mathrm{mgu}(t_1, t_2)$ denotes any 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.

A literal L is an object of the form $p(t_1, \ldots, t_n)$ (atom or positive literal) or $\neg p(t_1, \ldots, t_n)$(negative literal), where $p \in \mathcal{P}$, $t_1, \ldots t_n \in \mathcal{T}$ and $\neg$ denotes negation. A clause is a formula of the form $H : -L_1, \ldots, L_n$ with $n \geq 0$, where H (the head) is an atom and $L_1, \ldots, L_n$ (the body) are literals. A definite clause is a clause whose body contains atoms only. If the body is empty the clause is a unit clause. A normal program is a finite set of clauses. A positive program is a finite set of definite clauses. A normal (positive) goal is a formula $L_1, \ldots, L_m$, where each L_i is a literal (atom). If the signature is not specified, we assume the one defined by the symbols occurring in the program P and denote by $\mathrm{pred}(P)$ the set of predicate symbols occurring in P. The Herbrand base $\mathcal{B}_{\mathcal{P}}$ of a program P is the set of all ground atoms whose predicate symbols are in $\mathrm{pred}(P)$.

An Herbrand interpretation I for a program P is any subset of the Herbrand base $\mathcal{B}_{\mathcal{P}}$. For ground atoms, the satisfiability relation is defined as $I \models A$ if and only if $A \in I$. 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). $\mathcal{M}(P)$ is also the least fixpoint $T_P \uparrow \omega$ of a continuous transformation T_P (immediate consequences operator) on Herbrand interpretations. The ordinal powers of a generic monotonic operator T_P on a complete lattice $(D, \leq)$ with bottom $\perp$ are defined as usual, namely $T_P \uparrow 0 = \perp$, $T_P \uparrow (\alpha + 1) = T_P(T_P \uparrow \alpha)$, for α successor ordinal and $T_P \uparrow \alpha = lub(\{T_P \uparrow \beta \mid \beta \leq \alpha\})$ if α is a limit ordinal. If G is a positive goal, $G \overset{\theta}{\rightsquigarrow}_{P,R} B_1, \ldots, B_n$ denotes an SLD-derivation of $B_1, \ldots, B_n$ from the goal G in the program P which uses the selection rule R and such that θ is the composition of the mgu's used in the derivation. $G \xrightarrow{\theta}_P \Box$ denotes the SLD-refutation of G in the program P with computed answer substitution θ and computed instance $G\theta$. A computed answer substitution is always restricted to the variables occurring in G. We will denote by $\tilde{X}$ and $\tilde{t}$ a tuple of distinct variables and a tuple of terms respectively, while $\tilde{B}$ will denote a (possibly empty) conjunction of atoms.

2.2. Operational semantics

The s-semantics approach is based on the idea of choosing (equivalence classes of) sets of clauses as semantic domains. The denotations are defined by syntactic objects, as in the case of Herbrand interpretations, and are called π-interpretations. It is worth noting that the aim of the approach is not defining a new notion of model and that π-interpretations are not

interpretations in the conventional sense. The aim is that of providing new notions of program denotations useful from the programming point of view. The amount of syntax which is needed in the semantic domains depends on the observable property and on the composition operator we are considering. Consider for example positive logic programs and computed answer substitutions as observables. Since variables are essential in the description of these observables, the syntactic construct of variables is added to the Herbrand domain. Thus, the denotation of a program is a set of non-ground atoms, which can be viewed as a possibly infinite program. This is just an instance of a more general property of denotations within the approach. Namely denotations are possibly infinite programs and semantic domains are made of syntactic objects.

When we consider also union of programs, non-ground unit clauses are no longer sufficient to describe different status, and thus denotations are formed by general clauses. Since we have syntactic objects in the semantic domain, we need an equivalence relation in order to abstract from irrelevant syntactic differences. For instance, in the case of computed answer substitutions, this relation is variance.

The operational semantics of a program P is then a π-interpretation I, which has the following property. P and I are observationally equivalent with respect to any goal G. This is the property which allows us to state that the semantics does indeed capture the observable behavior. If the equivalence is accurate enough the semantics is fully abstract.

2.3. Fixpoint semantics and unfolding

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 [44]. The equivalence proofs can be stated in terms of simple properties of the unfolding and immediate consequences operators [21].

Given an operational semantics, the following steps are usually needed in order to define a fixpoint semantics equivalent to it. First, it is necessary to 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 T_P^π 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 $\mathcal{F}(P) = T_P^\pi \uparrow \omega$. Finally, the fixpoint semantics $\mathcal{F}$ is proved equivalent to the operational semantics. If this equivalence holds, the immediate consequences operator T_P^π models the observable properties and may be used for bottom-up program analysis.

An intermediate notion of unfolding semantics $\mathcal{U}(P)$ [44] helps in the construction of the equivalence proofs. Unfolding is a well-known program transformation operation which allows us to partially evaluate a program by expanding procedure definitions at the calling point. If the language syntax is powerful enough to express its own semantics, we can repetitively apply the unfolding operator and construct a collection of programs and an associate collection of π-interpretations. The unfolding semantics $\mathcal{U}(P)$ is obtained as the limit of this process. It is strongly related to the operational semantics, since they are based on the same inference rule. Indeed, if the unfolding rule preserves the observable properties, then $\mathcal{U}(P)$ is equivalent to the operational semantics $\mathcal{O}(P)$. Moreover, the definition of the immediate consequences operator T_P^π can be based on that of the unfolding operator, which is easier to define because of its strong relation to the operational semantics, and the equivalence proofs can exploit this relation.

2.4. π-models

As already noticed, a π-interpretation I is not an interpretation in the conventional sense. It has to be seen as just a syntactic notation for a set $\mathcal{H}(I)$ of Herbrand interpretations.[1] For instance, when computer answer substitutions and positive logic programs are considered, the operational semantics $\mathcal{O}(P)$ is a set of non-ground atoms and $\mathcal{H}(\mathcal{O}(P))$ is the set containing only one interpretation: the least Herbrand model of P. The following relation holds in all the instances of the s-semantics approach.

A π-interpretation I is a π-model of a program P if and only if every Herbrand interpretation in $\mathcal{H}(I)$ is a model of P.

A partial order $\preceq$ is usually defined on π-interpretations so that $I_1 \preceq I_2$ means that I_1 conveys more information than I_2 and that $(\mathcal{I}, \preceq)$ is a complete lattice. Indeed, it is worth noticing that the information of a π-interpretation I_1 may be contained in I_2 without I_1 being a subset of I_2. In the lattice $(\mathcal{I}, \preceq)$ the greatest lower bound of a set of π-models is a π-model and there exists a least π-model, which is the least Herbrand model. It is worth noting that the most expressive π-model is a non-minimal π-model.

3. The principal instances

In this section we consider the three principal instances of the s-semantics approach, namely the original computed answer semantics (s-semantics) for positive programs [26,27], its compositional extension [9] and the collecting semantics of resultants [34,32] which gives the maximum amount of information on computations.

[1] This view, first introduced in [27], prevails that based on the original notions of s-truth and s-model of [26].

3.1. The computed answer semantics

In the seminal paper [26], the authors motivate their proposal by showing that the standard declarative semantics is not correct w.r.t. to the observational equivalence based on computed answer substitutions. Namely, there exist programs which have the same least Herbrand model, yet compute different answer substitutions. Consider for instance the following two programs:

$$P_1 = \{ \quad \texttt{r(a).} \quad \texttt{p(a).} \quad \texttt{p(X).}\} \qquad P_2 = \{ \quad \texttt{r(a).} \quad \texttt{p(X).}\}$$

They have the same least Herbrand model but different computational behavior since the goal `p(X)` produces different answers if queried in P_1 or P_2.

The problem is that the equivalence based on the logical view considers two programs P_1 and P_2 equivalent if and only if any goal G is refutable in P_1 if and only if it is refutable in P_2. But the equivalence induced by this view is definitely too abstract to capture the essence of logic programming, i.e. the ability to compute answers. If we observe computed answers two programs are equivalent if and only if any positive goal G has the same computed instances (up to renaming) in the two programs. Recall that $\approx$ denotes the variance relation on terms.

Let P_1 and P_2 be positive logic programs. Then, $P_1 \simeq P_2$ if for every positive goal G, $G \xrightarrow{\vartheta}_{P_1} \Box \iff G \xrightarrow{\vartheta'}_{P_2} \Box$ and $G\vartheta \approx G\vartheta'$.

Before defining the operational semantics which captures such equivalence we have to introduce the suitable π-interpretations. They are subsets of the quotient set of all the (possibly non-ground) atoms w.r.t. variance.

Let $\mathcal{A}/_{\approx}$ the quotient set of all the atoms w.r.t the variance relation $\approx$. A π-interpretations is any subset of $\mathcal{A}/_{\approx}$.

We denote by $\mathcal{I}_\pi$ the set of all π-interpretations. The equivalence class containing all the variants of the atom A is denoted by $A^{\approx}$ or by A itself [26], if no ambiguity arises. The operational semantics $\mathcal{O}(P)$ is defined as follows.

$$\begin{aligned}\mathcal{O}(P) = \{A^{\approx} \mid\ & \exists p \text{ of arity } n \geq 0, \exists X_1, \ldots, X_n \text{ distinct variables},\\ & \exists \vartheta \text{ such that } p(X_1, \ldots, X_n) \xrightarrow{\vartheta}_P \Box,\\ & A = p(X_1, \ldots, X_n)\vartheta\}.\end{aligned}$$

Notice that we can associate a program $\wp(I)$ to any π-interpretation I as follows: $\wp(I) = \{A \mid A^{\approx} \in I\}$. Therefore $\mathcal{O}(P)$ can be seen as a (possibly infinite) program containing only facts.

In [26,27] many important results on $\mathcal{O}(P)$ are proved. First of all the fact that it correctly models the equivalence relation $\simeq$ on programs, i.e., if two programs have the same semantics then they are equivalent, and that it is fully abstract, i.e., two equivalent programs have the same operational semantics.

Let P_1 and P_2 be two positive logic programs. Then, $P_1 \simeq P_2$ if and only if $\mathcal{O}(P_1) = \mathcal{O}(P_2)$.

The previous result derives from the fact that the semantics, even if operational, is goal independent, and that it is AND-compositional, i.e., compositional with respect to the conjunction of atoms.

Let P be a positive program and $G = G_1 \ldots G_n$ be a positive goal. Then $G \xrightarrow{\vartheta}_P \Box$ if and only if there exist $A_1, \ldots, A_n \in \mathcal{O}(P)$ and γ such that $\gamma = mgu((A_1, \ldots, A_n), (G_1, \ldots, G_n))$, $G\theta \approx G\gamma$ and $A_1, \ldots, A_n$ are renamed apart w.r.t. $G_1, \ldots, G_n$.

Similar results hold for all the semantics defined according to the s-semantics style. This is also the key property which allows us to use abstractions of the semantics for goal independent abstract interpretation. The semantics $\mathcal{O}(P)$ can be viewed as a (possibly infinite) program containing only facts and the answer substitutions for a goal G can be determined by *executing* G *in* $\mathcal{O}(P)$.

Since the existence of computed answers and successes are strongly related, there must exist a relation between $\mathcal{O}(P)$ and the standard least Herbrand model semantics $\mathcal{M}(P)$. Given a π-interpretation I we denote by $[I]$ the set of all possible ground instances of the atoms in (the equivalent classes of) I; $[I]$ is clearly an Herbrand interpretation. The correspondence between the two semantics is the following.

Let P be a positive program, then $\mathcal{M}(P) = [\mathcal{O}(P)]$.

Another useful property of the s-semantics is its independence from the language [52,46]. More precisely, if we extend the language $\mathcal{L}_P$ (whose signature is the one defined by the symbols occurring in the program P), we do not obtain a different denotation, even if the set $\mathcal{A}/_{\approx}$ changes. It is worth noticing that there are other proposals of semantics which contain also non-ground atoms (e.g. in [14,28,35]) but they are not correct w.r.t the equivalence $\simeq$ and language independent. Indeed, in [46] it is showed that language independence is a key property to correctly model the computed answers.

The computed answer substitutions semantics can be obtained both by a bottom-up and a top-down construction. The first one is obtained as a fixpoint of an immediate consequences operator T_P^π on the complete lattice, $(\mathcal{I}_\pi, \subseteq)$, of π-interpretations ordered by set inclusion. Let P be a positive program and I be a π-interpretation.

$$T_P^\pi(I) = \{A^{\approx} \in \mathcal{A}/_{\approx} \mid \begin{array}{l} \exists \text{ a clause } c : H \coloneq B_1, \ldots, B_n \in P, \\ \exists\, C_1^{\approx}, \ldots, C_n^{\approx} \in I, \text{ where } C_1, \ldots, C_n \\ \text{are renamed apart w.r.t. the clause c,} \\ \exists \vartheta = \mathsf{mgu}((B_1, \ldots, B_n), (C_1, \ldots, C_n)), \\ A = H\vartheta \ \}. \end{array}$$

Note that T_P^π is different from the standard T_P operator [23] in that it derives instances of the clause heads by unifying the clause bodies with atoms in the current π-interpretation, rather than by taking all the possible ground instances.

In [26] we find the proof that the T_P^π operator is continuous on $(\mathcal{I}, \subseteq)$ and the following definition.

The fixpoint semantics of a positive program P is defined as $\mathcal{F}(P) = T_P^\pi \uparrow \omega$.

It is worth noticing that all the finite approximations $T_P^\pi \uparrow i$ are finite, since any program P is a finite set of clauses. The T_P^π operator can then effectively be used for the construction of bottom-up proofs.

The equivalence between $\mathcal{F}(P)$ and $\mathcal{O}(\mathcal{P})$ was first given in [26]. An alternative proof is given in [21], based on the following unfolding operator unf_P [44]. Let P and Q be positive programs. Then the unfolding of P w.r.t. Q is defined as:

$$unf_P(Q) = \{(A \coloneq \tilde{D_1}, \ldots, \tilde{D_n})\vartheta \mid \begin{array}{l} \exists \text{ a clause } A \coloneq B_1, .., B_n \in P, \\ \exists\, n \text{ renamed apart clauses :} \\ \quad C_1 \coloneq \tilde{D_1} \in Q, \\ \quad \vdots \\ \quad C_n \coloneq \tilde{D_n} \in Q, \\ \exists\, \vartheta, \ \mathsf{mgu} \ \text{of :} \\ (B_1, .., B_n) = (C_1, .., C_n) \ \}. \end{array}$$

Since the language is closed under unfolding, i.e., if P and Q are programs also $unf_P(Q)$ is a program, it is possible to define a collection of programs.

$$P_0 = P$$
$$P_i = unf_P(P_{i-1}), \quad i = 1, 2, \ldots.$$

The unfolding semantics $\mathcal{U}(P)$ of the positive program P is defined as

$$\mathcal{U}(P) = \bigcup_{i=0,1,\ldots} P_i^\pi$$

where $P_i^\pi = \{A^{\approx} \mid A \in P_i\}$ is the collection of π-interpretations associated to the unfoldings of P. We can prove [8] that:

Let P be a positive program. Then $\mathcal{F}(P) = \mathcal{U}(P) = \mathcal{O}(P)$.

In [27], in order to introduce the notion of π-model, a function $\mathcal{H}$ from π-interpretations to sets of Herbrand interpretations is first defined.

Let I be a π*-interpretations, then* $\mathcal{H}(I) = [I]$ *where* $[I]$ *is the set containing all the ground instances of all the atoms A such that* $A^{\approx} \in I$*. Then, I is a* π*-model of P if* $\mathcal{H}(I)$ *is an Herbrand model of P.*

Thus $\mathcal{H}(I)$ contains just one Herbrand interpretation $[I]$ the set of ground instances of the atoms in (the equivalent classes of) I. Notice that it is exactly the least Herbrand model of the associated program $\wp(I)$. On the other hand, every Herbrand model of P can be seen as a π-model of P. Therefore one could wonder if the least Herbrand model of P is also the least π-model of P. This is not true since the intersection property does not hold on π-models. Consider for instance the following two programs:

$$P_1 = \{ \begin{array}{l} \texttt{p(a).} \\ \texttt{p(X).} \\ \texttt{q(b).}\} \end{array} \qquad P_2 = \{ \begin{array}{l} \texttt{p(X).} \\ \texttt{q(b).}\} \end{array}$$

They have the same least Herbrand model which is a π-model of both programs. It is easy to check that also $\{q(b), p(X)\}$ is a π-model of P_2, but $\{q(b)\} = \{q(b), p(a), p(b)\} \cap \{q(b), p(X)\}$ is not a π-model of P_2.

This is not surprising, since π-interpretations *represent* sets of non-ground atoms. A more adequate partial order relation $\sqsubseteq$ on π-interpretations is defined in [27]. Let I_1 , I_2 be π-interpretations. We define:

$I_1 \preceq I_2$ *if and only if* $\forall A_1^{\approx} \in I_1\ \exists A_2^{\approx} \in I_2$ *such that* $A_2 \preceq A_1$.
$I_1 \sqsubseteq I_2$ *if and only if* $I_1 \preceq I_2$ *and* ($I_2 \preceq I_1$ *implies* $I_1 \subseteq I_2$).

We can prove that $(\mathcal{I}_\pi, \sqsubseteq)$ is a complete lattice; $\mathcal{A}/_\approx$ is the top element and $\emptyset$ is the bottom element. Moreover, the greatest lower bound of any set of π-models is a π-model and:

The least Herbrand model $\mathcal{M}(P)$ of a positive program P is the least π-model of P in the lattice $(\mathcal{I}_\pi, \sqsubseteq)$.

It is worth noticing that the semantics of computed answer substitutions $\mathcal{O}(P)$ is simply a non-ground representation of the least Herbrand model of P. From the Herbrand models viewpoint the two semantics are therefore equivalent. However $\mathcal{O}(P)$ contains more useful information.

3.2. The ∪-compositional computed answer substitutions semantics

In this section we consider an extension of the computed answer substitutions semantics which models a finer observational equality on positive logic programs.

Let P_1 and P_2 be two positive logic programs. Then, $P_1 \simeq_\cup P_2$ if for every positive goal G and positive program Q, $G \xrightarrow{\vartheta}_{P_1 \cup Q} \square \iff G \xrightarrow{\vartheta'}_{P_2 \cup Q} \square$ and $G\vartheta \approx G\vartheta'$.

It is easy to see that both the least Herbrand model semantics and the computed answer substitutions semantics do not model such an observable. Consider for instance the following two programs:

$$P_1 = \{ \quad \texttt{p(a).} \qquad\qquad P_2 = \{ \quad \texttt{p(a).}\}$$
$$\texttt{q(b) :-p(b).}\}$$

They have the same least Herbrand model semantics which is also the same computed answer substitutions semantics but when we consider their union with the program $Q = \{\texttt{p(b).}\}$ and the goal `?- q(X).` we obtain different computed answers.

We say that a semantics $\mathcal{S}$ is ∪-compositional w.r.t. computed answer substitutions if it correctly models the observational equivalence $\simeq_\cup$ and if for any pair of programs P_1, P_2, $\mathcal{S}(P_1 \cup P_2)$ can be obtained from $\mathcal{S}(P_1)$ and $\mathcal{S}(P_2)$.

In the literature we find different proposals of semantics which are correct with respect to $\simeq_\cup$ and compositional (e.g. [35,43,36]) but we consider here the Ω-semantics defined in [10,9] according to the general s-semantics approach. It was originally defined for a more general composition operator $\cup_\Omega$, defined on Ω-open programs. An Ω-open program [9] 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 $P_1 \cup_\Omega P_2$ is defined only if the predicate symbols occurring in both P_1 and P_2 are contained in Ω. When Ω contains all the predicate symbols of P_1 and P_2 we get the standard ∪-composition. The equivalence $\simeq_{\cup_\Omega}$ is defined as:

Let P_1 and P_2 be two positive logic programs. Then, $P_1 \simeq_{\cup_\Omega} P_2$ if for every positive goal G and positive program Q, such that both $P_1 \cup_\Omega Q$ and $P_2 \cup_\Omega Q$ are defined: $G \xrightarrow{\vartheta}_{P_1 \cup_\Omega Q} \square \iff G \xrightarrow{\vartheta'}_{P_2 \cup_\Omega Q} \square$ and $G\vartheta \approx G\vartheta'$.

Similarly to what we did for the semantics $\mathcal{O}$, before defining the operational semantics which captures such equivalence we have to introduce the suitable π-interpretations. First we define the following equivalence relation on clauses.

Let $c_1 = A_1 :- B_1, \ldots, B_n$ and $c_2 = A_2 :- D_1, \ldots, D_n$ be two clauses. Then, $c_1 \simeq_c c_2$ if there exists a permutation $D_{i_1}, \ldots, D_{i_n}$ of $D_1, \ldots, D_n$ such that $(A_1, B_1, \ldots, B_n) \approx (A_2, D_{i_1}, \ldots, D_{i_n})$.

The π_Ω-interpretations for Ω-programs are equivalence classes of sets of clauses whose body contains only predicates in Ω.

Let $\mathcal{C}_\Omega$ be the quotient set of all the clauses whose body contains only predicates in Ω. w.r.t the relation $\simeq_c$. A π_Ω-interpretation is any subset of $\mathcal{C}_\Omega$.

Since there is a 1–1 correspondence between π_Ω-interpretations and sets of (renamed apart) syntactic clauses, we use the same notation for programs and π_Ω-interpretations.

Let $\mathtt{Id}_\Omega$ be the set of clauses $\{\mathtt{p}(\tilde{X}) :- \mathtt{p}(\tilde{X}) \mid \mathtt{p} \in \Omega\}$. The operational semantics which models $\simeq_{\cup_\Omega}$ is then define as follows. Recall that we consider bodies of clauses as multisets.

Let P be a positive program, Ω be a set of predicate symbols, P^+ be the augmented program $P \cup \mathtt{Id}_\Omega$ and R be a fair selection rule. Then,

$$\begin{aligned}\mathcal{O}_\Omega(P) = \{c \mid \quad & \exists p \text{ of arity } n \geq 0, \exists X_1, \ldots, X_n \text{ distinct variables},\\ & \exists \gamma, \vartheta \text{ such that } p(X_1, \ldots, X_n) \overset{\gamma}{\rightsquigarrow}_{P,R} D_1, \ldots, D_m\\ & \qquad \text{and } D_1, \ldots, D_m \overset{\vartheta}{\rightsquigarrow}_{P^+,R} B_1, \ldots, B_s,\\ & \text{the predicate symbols of } B_1, \ldots, B_s \text{ are all in } \Omega,\\ & c = p(X_1, \ldots, X_n)\gamma\vartheta :- B_1, \ldots, B_s\ \}.\end{aligned}$$

In the above definition, the set of tautologies $\mathtt{Id}_\Omega$ is used to delay the evaluation of open atoms. This is a trick which allows us to obtain a denotation which is independent from the (fair) selection rule.

As an example consider the following Ω-open program with $\Omega = \{q\}$.

$$P = \{\ \begin{array}{ll} \texttt{p1(X)} \coloneq \texttt{q(X).} & \texttt{p2(X)} \coloneq \texttt{r(X).} \\ \texttt{q(a).} & \texttt{r(b).}\} \end{array}$$

Then, $\mathcal{O}_\Omega(P) = \{\texttt{p1(X)} \coloneq \texttt{q(X)},\ \texttt{p1(a)},\ \texttt{p2(b)},\ \texttt{r(b)},\ \texttt{q(a)}\}$

In [9] we find the proof that $\mathcal{O}_\Omega$ actually models computed answer substitutions in a compositional way.

Let P_1, P_2 be programs and $\mathrm{Pred}(P_1) \cap \mathrm{Pred}(P_2) \subseteq \Omega$. *Then,*

$$\mathcal{O}_\Omega(\mathcal{O}_\Omega(P_1) \cup_\Omega \mathcal{O}_\Omega(P_2)) = \mathcal{O}_\Omega(P_1 \cup_\Omega P_2).$$

The semantics $\mathcal{O}_\Omega(P)$ can be obtained also by a fixpoint construction. The suitable immediate consequences operator can be defined in terms of the unfolding operator, defined in the previous section. Actually, we are considering here its "semantic" version, which is well defined since clauses are always renamed apart and logical conjunction is commutative.

Let P be a positive Ω-program and $I \subseteq \mathcal{C}_\Omega$. Then $T_P^{\pi_\Omega}(I) = unf_P(I \cup Id_\Omega)$.

Since $T_P^{\pi_\Omega}$ is continuous on $(\mathcal{I}, \subseteq)$, there exists its least fixpoint which is the fixpoint semantics $\mathcal{F}_\pi^\Omega(P)$ of P: $\mathcal{F}_\pi^\Omega(P) = T_P^{\pi_\Omega} \uparrow \omega$. The equivalence between the operational and the fixpoint semantics can also be proved [10,9].

Let P be an Ω-program. Then $\mathcal{F}_\pi^\Omega(P) = \mathcal{O}_\Omega(P)$.

To introduce the notion of π_Ω-models we can associate to the π_Ω-interpretation I the set of the least Herbrand models of all the programs which can be obtained by "completing" the denotation I, considered as a program. This completion has to be formed in all the possible ways that we consider as a possible definition of the Ω-predicates.

Let I be a π_Ω-interpretation. Then $\mathcal{H}_\Omega(I) = \{\mathcal{M}(I \cup_\Omega J) \mid J \subseteq [\mathsf{open}(I)]\}$, where $\mathsf{open}(I)$ is the set of all atoms in the body of a clause in I.

Consider for instance the previous program P whose Ω-semantics is $\mathcal{O}_\Omega(P) = \{\texttt{p1(X)} \coloneq \texttt{q(X)},\ \texttt{p1(a)},\ \texttt{p2(b)},\ \texttt{r(b)},\ \texttt{q(a)}\}$. We have:

$$\mathcal{H}_\Omega(\mathcal{O}_\Omega(P)) = \{\ \begin{array}{l} \{\texttt{p1(a)},\ \texttt{p2(b)},\ \texttt{r(b)},\ \texttt{q(a)}\}, \\ \{\texttt{p1(b)},\ \texttt{p1(a)},\ \texttt{p2(b)},\ \texttt{r(b)},\ \texttt{q(a)},\ \texttt{q(b)}\}\ \} \end{array}$$

Then, according to the general schema of the s-semantics approach, a π_Ω-interpretation I is a π_Ω-model if all the Herbrand interpretations in $\mathcal{H}_\Omega(I)$ are Herbrand models of P. We can prove [9] that

- *Every Herbrand model of P is a π_Ω-model of P;*
- *$\mathcal{O}_\Omega(P)$ is a π_Ω-model of P*

The main idea behind the compositional semantics is the use of sets of clauses as semantic domain. This is the syntactic device which allows us to obtain a unique representation for a possibly infinite set of Herbrand models when a unique representative Herbrand model does not exist. Similar domains consisting of clauses have been used to model other non-standard observable properties like partial computed answer substitutions, call patterns or resultants and to characterize logic programs with negation [42,33,22], with the aim of delaying the evaluation of negative literals. Moreover, by modifying $\mathcal{O}_\Omega(P)$ we can obtain semantics compositional w.r.t. other composition operators, as for example inheritance mechanisms [6].

3.3. The resultants semantics and its abstractions

There are tasks, such as program analysis and optimization, where we are forced to observe and take into account more concrete observable properties which make visible internal computation details, like the sequences of goals, most general unifiers and variants of clauses involved in a computation. The basic notion which allows us to capture these properties is that of *resultants*, introduced in [49] in the framework of partial evaluation. They are a compact representation of the relation between the initial goal and the current (*goal*, *mgu*) pair and are useful (see [1]) to formalize the properties of SLD-resolution. They are the basic observables used by Gabbrielli, Levi and Meo in [32,34] to introduce a semantics which collects information on SLD-derivations. In [8] they are extended by collecting also sequences of clause identifiers to obtain the maximum amount of information on computations so to observe all the internal details of SLD-derivations.

Let P be a positive program, G be a goal and R be a selection rule. If there exists an SLD-derivation $G \overset{\vartheta}{\leadsto}_{P,R} B_1, \ldots, B_n$ obtained by using the sequence of clauses $c_1, \ldots, c_k$, $k \geq 0$, then the pair $\langle G\vartheta \leftarrow B_1, \ldots, B_n, (c_1, \ldots, c_k)\rangle$ is the associated resultant with clauses.

Note that if we take into account the selection rule, then the order of atoms is relevant; hence both G and $B_1, \ldots, B_n$ has to be considered as ordered sequences. If the initial goal G is atomic then the resultants for G are a definite clause, with the body viewed as a sequence of atoms. Resultants which are variants of each other are equivalent. We denote by $\mathcal{R}^G_{(P,R)}$ the set of all the (equivalence classes of) resultants with clauses for the goal G in P via R.

Several semantics useful for program analysis can be obtained by abstraction from $\mathcal{R}^G_{(P,R)}$. The basic equivalence is the following.

Let P_1, P_2 be positive programs and R be a selection rule. Then $P_1 \approx_{\mathcal{R}} P_2$ if for every goal G, $\mathcal{R}^G_{(P_1,R)} = \mathcal{R}^G_{(P_2,R)}$.

The s-semantics technique can be used to obtain the top-down definition of a semantics $\mathcal{O}^R_{\mathcal{R}}(P)$ correct w.r.t. $\approx_{\mathcal{R}}$. The first step is the definition of $\pi_{\mathcal{R}}$-interpretations.

Let $\mathcal{G}_P = \{p(\tilde{X}) \mid p \in pred(P)\}$ be the set of all most general atomic goals for the program P. A $\pi_{\mathcal{R}}$-interpretation is any subset of $\bigcup_{G\in\mathcal{G}_P} \mathcal{R}^G_{(P,R)}$.

Since we are considering atomic goals, any element of a $\pi_{\mathcal{R}}$-interpretation is a pair $\langle c, c_s\rangle$ composed of a clause (up to variance) and a sequence of clause identifiers.

Let P be a positive program and R be a selection rule. Then,

$$\begin{aligned}\mathcal{O}^R_{\mathcal{R}}(P) = \{\langle c, cs\rangle \mid\ & \exists p \in \mathsf{pred}(P),\ \exists \tilde{X} \text{ distinct variables},\\ & \exists \text{ the derivation } p(\tilde{X}) \overset{\vartheta}{\leadsto}_{P,R} \tilde{B}\\ & \text{obtained by using the clauses } c_1, \ldots, c_k,\\ & c = p(\tilde{X})\vartheta \leftarrow \tilde{B},\\ & cs = (c_1, \ldots, c_k)\ \}.\end{aligned}$$

A bottom-up definition equivalent to the top-down one can be derived only for *l*ocal selection rules. These are rules which always selects one of the most recently introduced atoms in the derivation from the initial goal. An example is the leftmost rule of PROLOG. The fixpoint semantics equivalent to $\mathcal{O}^R_{\mathcal{R}}(P)$ specialized to the leftmost rule is the least fixpoint of the following operator defined in [8].

Let P be a positive program and left *the leftmost selection rule. Then,*

$$\begin{aligned}T^{left}_{\mathcal{R}}(I) = \mathtt{Id}_P \cup \{\langle res, cs\rangle \mid\ & \exists c = A \coloneq B_1, .., B_k, .., B_n \in P,\\ & \exists \langle B'_1, cs_1\rangle, \ldots, \langle B'_{k-1}, cs_{k-1}\rangle \in I,\\ & \exists \langle B'_k \coloneq D_1, .., D_m, cs_k\rangle \in I \cup \mathtt{Id}_P,\\ & \exists \vartheta = \mathsf{mgu}((B_1, .., B_k), (B'_1, .., B'_k)),\\ & res = (A \coloneq D_1, .., D_m, B_{k+1}, .., B_n)\vartheta,\\ & cs = (c \circ cs_1 \circ cs_2 \ldots \circ cs_k)\ \},\end{aligned}$$

where $Id_P = \{\langle p(\tilde{X}) \coloneq p(\tilde{X}), \epsilon\rangle \mid p \in pred(P)\}$ and $\circ$ denotes the concatenation of sequences.

As an example consider the following program P.

$c_1 = \mathtt{p(X,Y)} \coloneq \mathtt{q(X),\ r(X,Y).}$ $\qquad$ $c_2 = \mathtt{r(a,b).}$
$c_3 = \mathtt{q(a).}$ $\qquad$ $c_4 = \mathtt{q(b).}$

Let ϵ denotes the empty sequence, the semantics $\mathcal{O}^{\text{left}}_{\mathcal{R}}(P)$ contains the pairs:

$\langle \mathtt{p(X,Y)} \coloneq \mathtt{p(X,Y)}, \epsilon\rangle.$ $\qquad$ $\langle \mathtt{r(X,Y)} \coloneq \mathtt{r(X,Y)}, \epsilon\rangle$
$\langle \mathtt{q(X)} \coloneq \mathtt{q(X)}, \epsilon\rangle$
$\langle \mathtt{p(X,Y)} \coloneq \mathtt{q(X),r(X,Y)}, (c_1)\rangle$ $\qquad$ $\langle \mathtt{r(a,b)}, (c_2)\rangle$
$\langle \mathtt{q(a)}, (c_3)\rangle$ $\qquad$ $\langle \mathtt{q(b)}, (c_4)\rangle$
$\langle \mathtt{p(a,Y)} \coloneq \mathtt{r(a,Y)}, (c_1, c_3)\rangle.$ $\qquad$ $\langle \mathtt{p(b,Y)} \coloneq \mathtt{r(b,Y)}, (c_1, c_4)\rangle$
$\langle \mathtt{p(a,b)}, (c_1, c_3, c_2)\rangle.$

and $T^{\text{left}}_{\mathcal{R}}$ applied to $\{\langle \mathtt{q(a)}, (c_3)\rangle, \langle \mathtt{r(a,b)}, (c_2)\rangle\}$ returns:

$\{$ $\langle \mathtt{p(X,Y)} \coloneq \mathtt{p(X,Y)}, \epsilon\rangle.$ $\qquad$ $\langle \mathtt{r(X,Y)} \coloneq \mathtt{r(X,Y)}, \epsilon\rangle.$
$\langle \mathtt{q(X)} \coloneq \mathtt{q(X)}, \epsilon\rangle$
$\langle \mathtt{p(X,Y)} \coloneq \mathtt{q(X),r(X,Y)}, (c_1)\rangle$ $\qquad$ $\langle \mathtt{r(a,b)}, (c_2)\rangle$
$\langle \mathtt{q(a)}, (c_3)\rangle$ $\qquad$ $\langle \mathtt{q(b)}, (c_4)\rangle$
$\langle \mathtt{p(a,Y)} \coloneq \mathtt{r(a,Y)}, (c_1, c_3)\rangle.$ $\qquad$ $\langle \mathtt{p(a,b)}, (c_1, c_3, c_2)\rangle\}.$

We refer to [34] for the proofs of the equivalence between the top-down and bottom-up semantics, as well as the proof that this denotation allows us to determine the observable for any goal, according to the s-semantics approach.

From the model theory point of view, one can define the following function from $\pi_{\mathcal{R}}$-interpretations to Herbrand interpretations.

Let I be a $\pi_{\mathcal{R}}$-interpretation. Then $\mathcal{H}_{\mathcal{R}}(I)$ is the set consisting of the set of ground instances of the unit resultants in I.

It is easy to see that the set of $\pi_{\mathcal{R}}$-interpretations contains the set of π-interpretations and that (see [34]) $\mathcal{M}(P)$ (the least Herbrand model of P), $\mathcal{O}(P)$ (the computed answers semantics of P), as well as $\mathcal{O}^R_{\mathcal{R}}(P)$ (the resultants semantics of P) are all $\pi_{\mathcal{R}}$-models of P.

The semantics $\mathcal{O}^R_{\mathcal{R}}(P)$ yields a lot of information on the computational behavior of P, possibly more than demanded. In this case we can derive new semantics just by *abstracting* from the redundant details. Following this idea we obtain many different abstractions of $\mathcal{O}^R_{\mathcal{R}}(P)$ which correctly model different observable properties. We recall here few of them.

- The finite success semantics [20], where unit resultants (representing successful derivations) are taken distinct from the non-unit ones (representing possible non-terminating computations).
- The resultants semantics with depth [3], where a sequence of clauses is abstracted by its length.
- The partial answers semantics [32,34], where we only keep the heads of the resultants.
- The (leftmost) call patterns semantics [32,34], where we delete all the atoms in the clause bodies but the first.

4. Applications

The main motivation of the s-semantics approach is to provide semantics useful in the development of semantics-based methods for program analysis, verification and transformation. We recall here the principal results obtained by using the s-semantics approach.

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 ([19]) 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 [12,41]), while the bottom-up approach (see [50,51]) 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, with the s-semantics approach we 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 by Barbuti, Giacobazzi and Levi in [4]. 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 we get 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 [4] is the semantics of computed answer substitutions and thus the bottom-up construction is based on the operator $\mathcal{T}_P^{\pi}$ introduced in Section 3.1. It is worth noticing that previous attempts [50,51], 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 are 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. Again, the s-semantics approach showed that the choice of an adequate (concrete) semantics allows us to determine goal independent information on the call patterns and that this information can be computed both top-down and bottom-up. In [13] the bottom-up approach is extended to provide approximations of both partial answer substitutions and call patterns. 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 [15]). A similar methodology has been applied also to CLP programs [37], leading to a framework where abstraction simply means abstraction of the constraint system.

The overall abstract interpretation methodology based on the s-semantic approach consists of the following three steps.

1. Select an observable o such that the property to be considered by the analysis is an abstraction $\alpha(o)$ of it.
2. Select a concrete semantics $\mathcal{O}$ correct w.r.t. the observable o and a fixpoint construction, either top-down or bottom-up.
3. Define a suitable abstraction $\mathcal{O}_{\alpha(o)}$ of $\mathcal{O}$, by providing the abstract versions of the operator used to construct the concrete semantics.

If the abstraction satisfies suitable properties [40,37], the analysis is correct. Note that the AND-compositionality property of all the semantics defined by the s-semantics approach, including their abstract versions, allows us to proceed in a goal independent way since we can obtain the result for any specific goal G just by executing G in $\mathcal{O}_{\alpha(o)}(P)$.

Clearly, if we are interested in properties of the answer substitutions (such as aliasing and sharing) we have to choose a concrete semantics correct w.r.t. answer substitutions. Thus a semantics at least as detailed as the computed answer

substitution semantics (which is fully abstract with respect to that observable), has to be considered. On the other hand, if we want to perform analysis of program components in a modular way, we need a semantics compositional w.r.t. program union (see [14]). AS a matter of fact, $\mathcal{O}_\Omega(P)$ can be considered as the semantic basis for modular program analysis, since by using suitable abstractions we can analyze program components and then combine the results to obtain the analysis of the whole program. Thus, a semantics at least as detailed as the compositional computed answer substitutions semantics has to be considered. Similarly, if we want to determine abstract properties of the procedure calls we should use a concrete semantics which gives more information on the computation than just the computed answers. In [30] the call patterns semantics, as defined in [34,32], is considered.

The s-semantics approach had been successfully applied also to (declarative) debugging [53,48]. The combination of the s-semantics approach, algorithmic (declarative) debugging and abstract interpretation produced a very powerful technique for the error diagnosis of logic programs called abstract debugging [18,16,17]. The main advantages of this method over the previous ones derive from the relation between concrete and abstract semantics in the s-semantics approach. In fact, declarative debugging systems compare the results of sub-computations with what the programmer intended. This is expressed by an intended semantics which is an abstraction of the concrete semantics, usually represented by an oracle [53] which tells us whether a given object belongs to the semantics. It is an abstraction of the concrete semantics. Since abstract denotations are finite, they can explicitly be used as oracles. Then we can test a program in a uniform way w.r.t. different specifications of the program properties.

Program transformation

Any transformation technique like partial evaluation [49] or unfold/fold transformation [54] is defined so as to preserve semantic properties while improving some computational behavior. To prove that the process does not alter the semantics of the initial program is never an easy task; but it becomes rather complex when we are not just interested in preserving the least Herbrand model semantics but other observational equivalences which are not captured by such a semantics, like computed answer substitutions or finite failures or the compositional program equivalence. In these cases we need a reference semantics which correctly models the considered observables, so that the proof can be based on general results of the considered semantics and its associated immediate consequences operators. For instance, the semantics modeling computed answer substitutions of Section 3.1 has been the reference semantics used in [7] to prove that some basic transformation operations used in unfold/fold transformations and in partial deduction preserve computed answer substitutions of logic programs. A suitable abstract version of the immediate consequence operator $T_P^{\pi\Omega}$ introduced in Section 3.2 is instead at the basis of the specialization process used in the partial deduction framework described in [57] (see also [56]).

Indeed, both the resultants semantics and the compositional computed answer substitution semantics are strongly related to partial evaluation. The compositional semantics $\mathcal{O}_\Omega$ is essentially the result of the partial evaluation, where derivations terminate at open predicates (i.e. predicates in Ω). Similar reasoning had been applied in the context of normal logic programs by Aravindan and Dung in [2], where the proofs are based on the properties of the semantic kernel defined in [42]. The idea of the semantic kernel construction is to evaluate all the positive atoms in the clause bodies by unfolding them until there are no more positive atoms left. The semantic kernel is then a (possibly infinite) program consisting of clauses in the form $A :\!-\neg B_1, \ldots, \neg B_n$. which can be viewed as a π-interpretation (called quasi-interpretation in [42]). Its semantics can be derived by a fixpoint construction similar to the one used for the compositional semantics. In [22] it has been proved that every Herbrand model of the completion of the semantic kernel is a stable model of P and in [2] we find a very elegant proof of the correctness of unfold/fold w.r.t. several non-monotonic semantics (as, for example, the stable model and the well-founded model semantics), by showing that it preserves the semantic kernel. A resultants semantics for Constraint Logic Programming (CLP) is instead the reference semantics used to prove that by using the transformation system proposed in [24], the original and the transformed programs have the same computational behavior, in terms of answer constraints, also when they are composed with other modules.

5. Conclusions

Several semantics for logic programs had been developed according to the same methodology known as "the s-semantics approach". In this paper we briefly recall the main characteristics of the approach and the properties which are common to all the semantics developed following it, namely: AND-compositionality, correctness w.r.t. the considered observable property, equivalence of the top-down and the bottom-up two definitions. As shown in [32,29], the various semantics are mutually related by means of abstractions.

We focused our attention on the three basic proposals: the semantics for computed answers substitutions [26,27], its compositional extension [9] and the collecting semantics of resultants with clauses [8,34,32]. They have been successfully applied to solve real problems, mainly in the field of program analysis and transformations. We have reported the main characteristics of the semantic-based frameworks for bottom-up abstract interpretation and bottom-up partial deduction of logic programs.

Many other observable properties for positive logic programs had been considered and suitable semantics proposed. The approach has also been applied to several extensions of positive logic programs, including concurrent constraint logic

programs [31] and normal constraint logic programs [25], constructive negation [55,33,11], structured logic programs with inheritance [6] and Prolog programs [5].

More difficult is the problem of modeling finite failures. In fact, even if the non-ground finite failure set defined in [45] is indeed correct w.r.t. this observable, and-compositionality, one of the basic properties of the s-semantics approach, does not hold. For many years the problem of verifying a program w.r.t. the observational equivalence based on finite failure had remained an open, challenging problem. Finally, in [38], a new fixpoint semantics, based on a co-continuous operator, which correctly models finite failure and is compositional w.r.t. the syntactic operators had been defined. Based on this fixpoint semantics a first inductive method able to verify a program w.r.t, the property of finite failure had been proposed. The method was not effective, but Gori and Levi in [39] show how top-down and bottom-up finite approximations can be used to make the method effective for the verification of program equivalence w.r.t. finite failures.

References

[1] K.R. Apt, Introduction to logic programming, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, Elsevier, Amsterdam and The MIT Press, Cambridge, 1990, pp. 495–574.
[2] Chandrabose Aravindan, Phan Minh Dung, On the correctness of unfold/fold transformation of normal and extended logic programs, Journal of Logic Programming 24 (3) (1995) 201–217.
[3] Roberto Barbuti, Michael Codish, Roberto Giacobazzi, Michael J. Maher, Oracle semantics for Prolog, Information and Computation 122 (2) (1995) 178–200.
[4] Roberto Barbuti, Roberto Giacobazzi, Giorgio Levi, A general framework for semantics-based bottom-up abstract interpretation of logic programs, ACM Transactions on Programming Languages and Systems (TOPLAS) 15 (1) (1993) 133–181.
[5] Annalisa Bossi, Michele Bugliesi, Massimo Fabris, A new fixpoint semantics for Prolog, in: D.S. Warren (Ed.), ICLP'93: Proceedings of the Tenth Int'l Conference on Logic Programming, Cambridge, MA, USA, The MIT Press, 1993, pp. 374–389.
[6] Annalisa Bossi, Michele Bugliesi, Maurizio Gabbrielli, Giorgio Levi, Maria Chiara Meo, Differential logic programming, in: POPL'93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, ACM Press, 1993, pp. 359–370.
[7] Annalisa Bossi, Nicoletta Cocco, Basic transformation operations which preserve computed answer substitutions of logic programs, Journal of Logic Programming 16 (1–2) (1993) 47–87.
[8] Annalisa Bossi, Maurizio Gabbrielli, Giorgio Levi, Maurizio Martelli, The s-semantics approach: Theory and applications, Journal of Logic Programming 19/20 (1994) 149–197.
[9] Annalisa Bossi, Maurizio Gabbrielli, Giorgio Levi, Maria Chiara Meo, A compositional semantics for logic programs, Theoretical Computer Science 122 (1–2) (1994) 3–47.
[10] Annalisa Bossi, Marina Menegus, Una semantica composizionale per programmi logici aperti, in: P. Asirelli (Ed.), Sesto convegno sulla programmazione logica, in: Proc. Sixth Italian Conference on Logic Programming, GULP, Pisa, Italy, June 1991, pp. 95–109.
[11] Paola Bruscoli, Francesca Levi, Giorgio Levi, Maria Chiara Meo, Compilative constructive negation in constraint logic programs, in: CAAP'94: Proc. of the 19th Int'l Colloquium on Trees in Algebra and Programming, London, UK, Springer-Verlag, 1994, pp. 52–67.
[12] Maurice Bruynooghe, A practical framework for the abstract interpretation of logic programs, Journal of Logic Programming 10 (2) (1991) 91–124.
[13] Michael Codish, Dennis Dams, Eyal Yardeni, Bottom-up abstract interpretation of logic programs, Theoretical Computer Science 124 (1) (1994) 93–125.
[14] Michael Codish, Saumya K. Debray, Roberto Giacobazzi, Compositional analysis of modular logic programs, in: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, 1993, pp. 451–464.
[15] Michael Codish, Harald Søndergaard, Meta-Circular Abstract Interpretation in Prolog, 2002, pp. 109–134.
[16] Marco Comini, Giorgio Levi, Maria Chiara Meo, Giuliana Vitiello, Proving properties of logic programs by abstract diagnosis, in: Selected Papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, London, UK, Springer-Verlag, 1997, pp. 22–50.
[17] Marco Comini, Giorgio Levi, Maria Chiara Meo, Giuliana Vitiello, Abstract diagnosis, Journal of Logic Programming 39 (1–3) (1999) 43–93.
[18] Marco Comini, Giorgio Levi, Giuliana Vitiello, Abstract debugging of logic program, in: LOPSTR'94/META'94: Proceedings of the 4th Int'l Workshops on Logic Programming Synthesis and Transformation — Meta-Programming in Logic, London, UK, Springer-Verlag, 1994, pp. 440–450.
[19] Patrick Cousot, Program analysis: The abstract interpretation perspective, ACM Computing Surveys 28 (4es) (1996) 165.
[20] Giorgio Delzanno, Maurizio Martelli, A bottom-up characterization of finite success, in: ILPS'93: Proceedings of the 1993 Int'l Symposium on Logic Programming, Cambridge, MA, USA, The MIT Press, 1993, p. 676.
[21] Francois Denis, Jean-Paul Delahaye, Unfolding, procedural and fixpoint semantics of logic programs, in: Eighth Annual Symposium on Theoretical Aspects of Computer Science, Hamburg, Germany, in: Lecture Notes in Computer Science, February 1991, pp. 511–522.
[22] P.M. Dung, K. Kanchanasut, A fixpoint approach to declarative semantics of logic programs, in: E.L. Lusk, R.A. Overbeek (Eds.), Logic Programming: Proc. of the North American Conference 1989, vol. 1, The MIT Press, Cambridge, MA, 1989, pp. 604–625.
[23] M.H. Van Emden, R.A. Kowalski, The semantics of predicate logic as a programming language, Journal of the ACM 23 (4) (1976) 733–742.
[24] Sandro Etalle, Maurizio Gabbrielli, Transformations of CLP modules, Theoretical Computer Science 166 (1–2) (1996) 101–146.
[25] Francois Fages, Roberta Gori, A hierarchy of semantics for normal constraint logic programs, in: Algebraic and Logic Programming, 1996, pp. 77–91.
[26] Moreno Falaschi, Giorgio Levi, Maurizio Martelli, Catuscia Palamidessi, Declarative modeling of the operational behaviour of logic languages, Theoretical Computer Science 69 (1989) 289–318.
[27] Moreno Falaschi, Giorgio Levi, Maurizio Martelli, Catuscia Palamidessi, A model-theoretic reconstruction of the operational semantics of logic programs, Information and Computation 102 (1) (1993) 86–113.
[28] Gèrard Ferrand, Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method, Journal of Logic Programming 4 (3) (1987) 177–198.
[29] Maurizio Gabbrielli, The semantics of logic programming as a programming language, Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa, 1992.
[30] Maurizio Gabbrielli, Roberto Giacobazzi, Goal independency and call patterns in the analysis of logic programs, in: ACM Symposium on Applied Computing, 1994, pp. 394–399.
[31] Maurizio Gabbrielli, Giorgio Levi, Unfolding and fixpoint semantics of concurrent constraint logic programs, in: Proc. of the Second Int'l Conference on Algebraic and Logic Programming, London, UK, Springer-Verlag, 1990, pp. 204–216.
[32] Maurizio Gabbrielli, Giorgio Levi, Maria Chiara Meo, Observational equivalences for logic programs, in: K. Apt (Ed.), Logic Programming, Proceedings of the Joint Int'l Conference and Symposium on Logic Programming, Washington, DC, The MIT Press, November 1992, pp. 131–145.
[33] Maurizio Gabbrielli, Giorgio Levi, Daniele Turi, A two steps semantics for logic programs with negation, in: LPAR'92: Proceedings of the Int'l Conference on Logic Programming and Automated Reasoning, 1992, pp. 297–308.
[34] Maurizio Gabbrielli, Maria Chiara Meo, Fixpoint semantics for partial computed answer substitutions and call patterns, in: H. Kirchner, G. Levi (Eds.), Algebraic and Logic Programming: Proc. of the Third Int'l Conference, Springer, Berlin, Heidelberg, 1992, pp. 84–99.
[35] H. Gaifman, E. Shapiro, Fully abstract compositional semantics for logic programs, in: POPL'89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, ACM Press, 1989, pp. 134–142.
[36] H. Gaifman, E. Shapiro, Proof theory and semantics of logic programs, in: Proceedings of the Fourth Annual Symposium on Logic in Computer Science, Piscataway, NJ, USA, IEEE Press, 1989, pp. 50–62.

[37] Roberto Giacobazzi, Saumya K. Debray, Giorgio Levi, A generalized semantics for constraint logic programs, in: Proceedings of the Int'l Conference on Fifth Generation Computer Systems, ICOT, Japan, Association for Computing Machinery, 1992, pp. 581–591.
[38] Roberta Gori, An abstract interpretation framework to reason on finite failure and other properties of finite and infinite computations, Theoretical Computer Science 290 (1) (2003) 863–936.
[39] Roberta Gori, Giorgio Levi, On the verification of finite failure, Journal of Computer and System Sciences 71 (4) (2005) 535–575.
[40] Dean Jacobs, Anno Langen, Static analysis of logic programs for independent AND parallelism, Journal of Logic Programming 13 (2–3) (1992) 291–314.
[41] G. Janssens, M. Bruynooghe, Deriving descriptions of possible values of program variables by means of abstract interpretation, Journal of Logic Programming 13 (2–3) (1992) 205–258.
[42] Kanchana Kanchanasut, Peter J. Stuckey, Transforming normal logic programs to constraint logic programs, Theoretical Computer Science 105 (1) (1992) 27–56.
[43] Jean-Louis Lassez, Michael J. Maher, Closures and fairness in the semantics of programming logic, Theoretical Computer Science 29 (1984) 167–184.
[44] Giorgio Levi, Models, unfolding rules and fixpoint semantics, in: R.A. Kowalski, K.A. Bowen (Eds.), in: Logic Programming: Proc. of the Fifth Int'l Conference and Symposium, vol. 2, The MIT Press, Cambridge, MA, 1991, pp. 1649–1665.
[45] Giorgio Levi, Maurizio Martelli, Catuscia Palamidessi, Failure and success made symmetric, in: S. Debray, M. Hermenegildo (Eds.), Logic Programming: Proc. of the 1990 North American Conference, The MIT Press, Cambridge, MA, 1990, pp. 3–22.
[46] Giorgio Levi, Davide Ramundo, Formalization of metaprogramming for real, in: ICLP'93: Proceedings of the Tenth Int'l Conference on Logic Programming, Cambridge, MA, USA, The MIT Press, 1993, pp. 354–373.
[47] J.W. Lloyd, Foundations of Logic Programming, Springer-Verlag New York, Inc., New York, NY, USA, 1984.
[48] John W. Lloyd, Declarative error diagnosis, New Generation Computing 5 (2) (1987) 133–154.
[49] J.W. Lloyd, J.C. Shepherdson, Partial evaluation in logic programming, Journal of Logic Programming 11 (1991) 217–242.
[50] Kim Marriott, Harald Søndergaard, Bottom-up abstract interpretation of logic programs, in: R.A. Kowalski, K.A. Bowen (Eds.), Proc. Fifth Int'l Conf. on Logic Programming, Cambridge, MA, The MIT Press, 1988, pp. 733–748.
[51] Kim Marriott, Harald Søndergaard, Semantics-based dataflow analysis of logic programs, in: G. Ritter (Ed.), IFIP Congress, North-Holland, 1989, pp. 601–606.
[52] Danny De Schreye, Bern Martens, A sensible least herbrand semantics for untyped vanilla meta-programming and its extension to a limited form of amalgamation, in: A. Pettorossi (Ed.), Meta-Programming in Logic: Proc. of the Third Int'l Workshop META-92, Springer, Berlin, Heidelberg, 1992, pp. 192–204.
[53] Ehud Y. Shapiro, Algorithmic Program Debugging, The MIT Press, Cambridge, MA, USA, 1983.
[54] Hisao Tamaki, Taisuke Sato, Unfold/fold transformation of logic programs, in: Second Int. Conf. on Logic Programming, 1984, pp. 127–138.
[55] Daniele Turi, Extending s-models to logic programs with negation, in: Koichi Furukawa (Ed.), Logic Programming, Proceedings of the Eighth International Conference, Cambridge, MA, USA, The MIT Press, June 1991, pp. 397–411.
[56] Wim Vanhoof, Techniques for online and offline specialisation of logic programs, Ph.D. Thesis, Katholieke Universiteit Leuven, June 2001.
[57] Wim Vanhoof, Danny De Schreye, Bern Martens, Bottom-up partial deduction of logic programs, Journal Functional and of Logic Programming (2) (1999) (special issue).