abstract standard semantics for the language. This constraint leads to a powerful framework of collecting semantics where abstract interpretation is used both to relate collecting semantics at different levels of abstraction, and to systematically construct them by composition. We introduce: (1) a hierarchy of collecting semantics for logic programs where standard semantic notions (e.g., model theory in logic programming) can be extended to any collecting and abstract semantics for analysis; (2) a systematic approach to collecting semantics design by abstraction, and composition of more abstract semantics, where collecting semantics are all constrained between a *standard* and an *operational* semantics of the language; (3) a constructive characterization for the *"optimal"* collecting semantics for program analysis, by combining the property to analyze and the standard semantics (see Theorem 6.4). This semantics characterizes precisely the least amount of information about program behaviour which is essential to analyze a given program property, and it can be considered as the simplest (most abstract) concrete semantics in semantic-based program analyses. This view of collecting semantics is particularly appealing in logic programming where for instance the distinction between declarative and procedural reading of programs identifies precisely this space of collecting semantics.

2 Preliminaries

In the following, we will assume familiarity with the standard notions of *logic programming* (e.g. see [2]) and *abstract interpretation* ([12, 13]).

Let A and B be sets. Set isomorphism is denoted $\cong$. The isomorphism $\imath : A \rightarrow B$ is denoted $A \cong_\imath B$. The powerset of A is denoted by $\wp(A)$. Sequences in A^* are typically denoted by $\langle a_1, \ldots, a_n\rangle$ or simply $a_1, \ldots, a_n$, for a_i's symbols in A. The empty sequence is denoted Λ. Concatenation of sequences $s_1, s_2 \in A^*$ is denoted $s_1 :: s_2$. The sequence of symbols with first element a followed by the sequence s is denoted $a \mid s$. A equipped with a partial order $\sqsubseteq$ is denoted $A_\sqsubseteq$. If A is a poset, we usually denote $\sqsubseteq_A$ the corresponding partial order. This notation is extended to arbitrary algebraic structures like lattices *etc.* Function composition is $\circ$ and sometimes is omitted. The set of fixpoints of a function f is denoted $fp(f)$, and the least fixpoint (if it exists) is denoted $lfp(f)$. The ordinal power of a function f is denoted $f\!\uparrow\!\alpha$ where α is an ordinal. ω denotes the first limit ordinal. Let $\langle A, \leq, \wedge, \vee, \top, \bot\rangle$ be a complete lattice, by an *(upper) closure operator* on A we mean an operator $\rho : A \rightarrow A$ such that for every $x, y \in A$: if $x \leq y$ then $\rho(x) \leq \rho(y)$, $x \leq \rho(x)$, and $\rho(\rho(x)) = \rho(x)$. We denote $uco(A)$ the set of all closure operators on A. It is a complete lattice $\langle uco(A), \sqsubseteq_{uco}, \sqcap, \sqcup, \lambda x.\top, \lambda x.x\rangle$ where for every $\rho, \eta \in uco(A)$ and $x \in A$: $\rho \sqsubseteq_{uco} \eta$ iff for each $x \in L$, $\rho(x) \leq \eta(x)$. $(B_{\sqsubseteq_B}, \alpha, A_{\sqsubseteq_A}, \gamma)$ is a Galois insertion iff $\alpha : B \rightarrow A$ and $\gamma : A \rightarrow B$ are monotonic mappings,

Throughout, Σ, Π and *Var* will respectively denote a set of function symbols, a set of predicate symbols and a denumerable set of variables, defining a *first-order language* $\mathcal{L}$. The set of terms, atoms, clauses and programs on the language $\mathcal{L}$ are denoted respectively *Term*, *Atom*, *Clause* and *Program*. $Term_\emptyset$, $Atom_\emptyset$ denote the sets of corresponding *ground* objects. Atoms and unit clauses will be considered equivalent notions. Tuples of syntactic objects of the same type (like variables, terms *etc.*) are sometimes denoted $\bar{s}$. In the following the set of variables (predicates) that occur in a syntactic object s is denoted $var(s)$ ($pred(s)$). If A is a set of syntactic objects and $\pi \subseteq \Pi$, then $A|_\pi$ denotes the restriction of A to the elements with predicate symbol in π. In the following we restrict our interest to idempotent substitutions ranging in *Sub*, unless explicitly stated otherwise. If $\Theta \subseteq Sub$, then $s\Theta = \{s\theta \mid \theta \in \Theta\}$. $t' \leq t$ iff there exists a substitution θ such that $t' = t\theta$. Syntactic objects t_1 and t_2 are *equivalent up to renaming*, denoted $t_1 \sim t_2$, iff $t_1 \leq t_2$ and $t_2 \leq t_1$. $Term/\!\sim$, $Atom/\!\sim$ and $Clause/\!\sim$ are complete lattices with respect to $\leq$, and will be often denoted *Term*, *Atom* and *Clause*. Since all the definitions in the paper are clearly independent on syntactic variable names, we will let a syntactic object denote its equivalence class by renaming. For a syntactic object s and a set of (equivalence classes by renaming) of objects I, we denote by $\langle c_1, \ldots, c_n\rangle \ll_s I$, $n \geq 0$ that $c_1, \ldots, c_n$ are representatives of elements of I renamed apart from s and from each other.

[illegible]

3 Systems of collecting semantics

A *semantic definition* (or *semantics*) is a pair $\langle C, T\rangle$, where $C_\sqsubseteq$ is a complete lattice (the semantic domain) and $T : Program \rightarrow (C \rightarrow C)$ is a mapping such that given a program P: $T(P)$ a continuous operator on C. In the following, when clear from the context, we abuse and let T denote $T(P)$ for a program P.

3.1 Collecting semantics

We formalize the notion of *collecting semantics* in the standard framework of abstract interpretation [12]. Given a Galois insertion (B, α, A, γ), Cousot & Cousot in [13] proved that it is always possible to associate with any operator $T : B \rightarrow B$, an operator which is the *best correct approximation* of T in A: namely the function $\alpha \circ T \circ \gamma$. In the following given any two semantics $\langle C, T\rangle$ and $\langle C', T'\rangle$, then $\langle C, T\rangle \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \langle C', T'\rangle$ will denote a Galois insertion (C', α, C, γ) such that $T = \alpha \circ T' \circ \gamma$.

Definition 3.1 *[soundness & completeness]*
Let $\mathcal{S}$ and $\mathcal{X}$ be semantic definitions. We say that $\mathcal{S}$ is sound *(*complete*) with respect to $\mathcal{X}$ if $\mathcal{S} \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \mathcal{X}$ (resp. $\mathcal{X} \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \mathcal{S}$).*

A *collecting semantics* (with respect to a semantics $\mathcal{S}$) is a semantics $\mathcal{X}$ which is complete with respect to $\mathcal{S}$. In this case, $\mathcal{S}$ will be called the *core semantics*. The core semantics is then an abstract interpretation of any collecting semantics, yet providing the *best* approximating operator with respect to the given abstraction. It provides a *lower bound* with respect to abstraction for collecting semantics. Therefore, $\mathcal{S} \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \mathcal{X}$ will denote also that $\mathcal{X}$ is a collecting semantics with respect to the core semantics $\mathcal{S}$. When the core semantics is fixed, a collecting semantics will be simply denoted by $\mathcal{X}_\alpha^\gamma$.

Proposition 3.1
Let $\langle C, T\rangle$ be a semantics and (C, α, A, γ) be a Galois insertion. Then $\langle \gamma\circ\alpha(C), \gamma\circ\alpha\circ T\rangle$ and $\langle A, \alpha\circ T\circ\gamma\rangle$ are isomorphic.

Given a semantics $\mathcal{S}$, a *system of collecting semantics* is a set of collecting semantics with respect to $\mathcal{S}$. Systems of collecting semantics with respect to $\mathcal{S}$ are usually denoted $\mho_{\mathcal{S}}$. A system is *generated by* $\mathcal{S}$ if it contains all the collecting semantics with respect to $\mathcal{S}$. Let $\mho$ be a system of collecting semantics, and $\propto \subseteq \mho \times \mho$ such that: $\mathcal{S} \propto \mathcal{X}$ iff $\exists\alpha\exists\gamma.\ \mathcal{S} \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \mathcal{X}$. From a similar result in [12], Galois insertions can be composed such that if $\mathcal{S}_\alpha^\gamma$ is a collecting semantics and $\mathcal{X}$ is such that $\mathcal{S} \overset{\tilde{\gamma}}{\underset{\tilde{\alpha}}{\rightleftharpoons}} \mathcal{X}$, then $\mathcal{X}_{\alpha\circ\tilde{\alpha}}^{\tilde{\gamma}\circ\gamma}$ is a collecting semantics. Notice that $\propto$ is a pre-order on $\mho$ and naturally defines an *observational equivalence* on collecting semantics: $\mathcal{S} \simeq \mathcal{X}$ iff $\mathcal{S} \propto \mathcal{X}$ and $\mathcal{X} \propto \mathcal{S}$. Note that if $\langle C, T\rangle \simeq \langle C', T'\rangle$ then $C \cong C'$. Moreover, let $\langle C, T\rangle$ be a collecting semantics, we follow [30][1] by defining an observation as an element in $\mathcal{O}(\mathcal{S}) = \{\mathcal{X} \mid \mathcal{X} \in \mho \ \&\ \mathcal{X} \propto \mathcal{S}\}$. It is immediate to prove that $\mathcal{S} \simeq \mathcal{X}$ iff $\mathcal{O}(\mathcal{S}) = \mathcal{O}(\mathcal{X})$. Equivalent collecting semantics allow the same set of possible observations (in the sense of [30]). Therefore, they are called *observationally equivalent*. In the following we abuse by denoting $\mho$ the set $\mho/_\simeq$. Moreover we assume $\langle C, T\rangle \not\simeq \langle C, T'\rangle$ while $\langle C, T\rangle \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \langle C', T'\rangle$. It is worth noting that any system $\mho_\propto$ is indeed a poset where the core semantics is fixed to be the most abstract object (i.e., the *least collecting semantics*).

3.2 Model-theoretic collecting semantics

Our notion of collecting semantics is suitable to provide a formal basis for the definition of a *model-theoretic collecting semantics* of logic programs. It is inspired by the s semantics approach in [21] and generalizes that results to any system of collecting semantics. We introduce the notions of *model* and *collecting model* for a program as a generalization on collecting semantics of the standard notion of Herbrand model. Models are T-closed interpretations. A collecting interpretation is a collecting model if, when abstracted, it provides a model for the program.

[1] [30] defines an *observation* for a semantics $\langle C, T\rangle$ as a complete lattice isomorphic to an upper closure of C. In our case, this naturally induces a (more abstract) semantics, i.e., an object in $\mathcal{O}(\langle C, T\rangle)$.

[illegible]

relationship between closure operators and abstract interpretations is in Cousot-Cousot 79

Definition 3.2 *[models & collecting models]*
Let P be a logic program and $\langle C', T'\rangle_\alpha^\gamma$ be a collecting semantics in $\mho_S$, where $S = \langle C, T\rangle$. An interpretation $M \in C$ is a S-model *for P if $T(M) \sqsubseteq_C M$. $M' \in C'$ is a* collecting model *for P if $\alpha(M')$ is a S-model for P.*

Proposition 3.2
Let P be a logic program and $\langle C, T\rangle_\alpha^\gamma$ be a collecting semantics in $\mho_S$. Then (1) M is a collecting model for a program P iff $\gamma \circ \alpha(M)$ is a collecting model for P; and (2) if $M \in C$ is a collecting model for P, then $T(M) \sqsubseteq_C \gamma\circ\alpha(M)$.

$T(M) \sqsubseteq_C \gamma\alpha(M)$ is not sufficient to prove that M is a collecting model. This is because the notion of collecting semantics is too weak to provide a characterization of collecting models in terms of the T operator.

Definition 3.3 *[model completeness]*
A collecting semantics $\langle C', T'\rangle_\alpha^\gamma$ in $\mho_{\langle C,T\rangle}$ is model-complete *if $T\circ\alpha \sqsubseteq_C \alpha\circ T'$.*

It is straightforward to observe that the composition of model complete collecting semantics is model complete. Because $T = \alpha\circ T'\circ\gamma$ and by Galois insertion, it is easy to prove that in any model complete collecting semantics: $T\circ\alpha = \alpha\circ T'$. Model completeness specifies that, from the core semantics viewpoint, T and T' are equivalent. Notice that the condition $T\circ\alpha = \alpha\circ T'$ is stronger than $T = \alpha\circ T'\circ\gamma$ (see [15]), and the independent combination of model complete collecting semantics is not in general model complete, as T may not be additive. In Section 6, we prove that model completeness is preserved in semantics combination when semantics are derived by abstraction of a model complete collecting semantics.

Proposition 3.3
Let P be a logic program and $\langle C, T\rangle_\alpha^\gamma$ be a model complete collecting semantics. Then (1) M is a collecting model for P iff $T(M) \sqsubseteq_C \gamma\alpha(M)$; and (2) if $T(M) \sqsubseteq_C M$ then M is a collecting model for P.

Because α is additive in any Galois insertion, it is easy to prove that the class of collecting models for a program in a collecting semantics $\langle C, T\rangle$ forms a sub-cpo of C. Moreover, given a model complete collecting semantics $\langle C, T\rangle$ and a program P: $fp(T) \subseteq \{M \mid M$ is a collecting model of $P\}$. Thus, the fixpoints of the operator T provide only a partial characterization of the class of collecting models for a program. An interpretation M such that $T(M) \sqsubseteq_C M$ will be called a *reachable collecting model*. We define *full collecting semantics* any collecting semantics $\langle C, T\rangle_\alpha^\gamma$ such that for any $I \in C$: $T(I) \sqsubseteq_C \gamma\circ\alpha(I) \Rightarrow T(I) \sqsubseteq_C I$. Notice that in full collecting semantics S, any collecting model is reachable. Therefore an interpretation M is a S-model iff M is a collecting model.

3.3 Herbrand, Clark, Heyting and *s* collecting semantics

In this section we consider a system of collecting semantics for logic programs based on the minimal Herbrand model semantics as core semantics. This system includes well known semantics for logic programs, and provides a "logical" notion of model for programs at different levels of abstraction. In the following we fix a function *ground* that maps any syntactic object to the set of its ground instances.

Definition 3.4 *[the Herbrand semantics $\mathcal{H}$ [31]]*
Let P be a logic program. The Herbrand semantics $\mathcal{H}$ of P is a pair $\langle\wp(Atom_\emptyset), T_P\rangle$ where for each $I \in \wp(Atom_\emptyset)$: $T_P(I) = \{h \in Atom_\emptyset \mid h \leftarrow \bar{b} \in ground(P),\ \bar{b} \subseteq I\}$.

It is well known that a Herbrand interpretation M is a Herbrand model iff $T_P(M) \subseteq M$ ([2]). Therefore, $\mathcal{H}$-models corresponds precisely to the Herbrand models of the program. In the following,

Heyting
S-sem
Clark (if infinite contexts these are equal)
Herbrand
core semantics.

when not specified otherwise, we will consider $\mathcal{H}$ as the *core semantics* in our examples of systems of collecting semantics.

The *s* semantics introduced in [20] is intended to provide a fully abstract description of computed answer substitutions of logic programs. This semantics has been successfully considered as a base semantics for abstraction, and applied to static program analysis in [6, 10]. The Clark's semantics instead has mostly a theoretical interest, being fully abstract with respect to the "more abstract" notion of atomic consequences of a program ([20])[2]. However, as we will show in Section 6, this simpler semantics may provide a sound basis for static analysis for some non trivial program properties. In the following we fix a function *up* that maps any syntactic object to the set of its instances. Let P be a program. The *s* and Clark's semantics are: $\langle\wp(Atom), T_P^s\rangle$ and $\langle up(\wp(Atom)), T_P^c\rangle$ resp., where:

$$T_P^s(I) = \left\{ h\vartheta \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1,\dots,b_n \in P, \\ \langle b_1',\dots,b_n'\rangle \ll_C I \\ \vartheta = mgu(\langle b_1,\dots,b_n\rangle, \langle b_1',\dots,b_n'\rangle) \end{array} \right\} \quad T_P^c(I) = \left\{ h\vartheta \;\middle|\; \begin{array}{l} C \equiv h \leftarrow b_1,\dots,b_n \in P \\ \vartheta \in Sub,\ b_1\vartheta,\dots,b_n\vartheta \in I \end{array} \right\}$$

$\mathcal{H}$ is an abstract interpretation of the (more concrete) *s* semantics in [20], yet providing the best correct approximation. Observe that $(\wp(Atom)_{\subseteq}, \alpha_g, \wp(Atom_\emptyset)_{\subseteq}, \gamma_g)$ such that: $\alpha_g = \lambda I.ground(I)$ and $\gamma_g = \lambda I.\{A \mid ground(A) \subseteq I\}$, is a Galois insertion.

Theorem 3.4
For each $I \in \wp(Atom_\emptyset)$: $T_P(I) = \alpha_g(T_P^s(\gamma_g(I)))$.

Likewise, Clark's semantics can be proved to be a collecting semantics, which is also an abstract interpretation of the *s* semantics yet providing the best correct approximation of T_P^s. Indeed, *up* is an upper closure operator on $\wp(Atom)$, naturally inducing a Galois insertion. In particular, $\langle up(\wp(Atom)), T_P^c\rangle$ is a collecting semantics and $T_P^c\circ up = up\circ T_P^s$. Clark's semantics is model complete ([21]), and therefore the *s* semantics is also model complete. Moreover, it is easy to see that the *s* and Clark's semantics are not a full collecting semantics (e.g., the least Herbrand model is not reachable).

The Heyting model theoretic semantics for logic programs has been introduced in [27] to provide an intuitionistic (constructive) interpretation for definite clause programs. Because for definite clause programs classical and intuitionistic logic agree, we can easily observe that the constructive approach of Heyting models, if compared with the Herbrand semantics, defines itself a collecting semantics. In the following we will slightly modify the definitions of Heyting semantics in [27]. Let *Tree* denote the domain of labeled trees over *Atom* such that, for any $a \in Atom$: $\langle a, \langle\rangle\rangle$ is a tree, and if $t_1,\dots,t_n$ are trees, then $\langle a, \langle t_1,\dots,t_n\rangle\rangle$ is a tree. For any tree t and substitution ϑ, $t\vartheta$ denotes the tree obtained by applying ϑ to the labels (atoms) in t. $Tree/\sim$ is called the *Heyting base* ([27]). Trees constructively represent proofs for atomic goals. A Heyting model is then a collection of "closed" trees corresponding to proof trees for (any) atomic goal. $M \subseteq Tree$ is a Heyting model ([27]) for a program P iff $\langle a, \langle\rangle\rangle \in M$ for any unit clause $a \in P$, and if $h \leftarrow b_1,\dots,b_n \in P$ and $\langle b_1\vartheta, t_1\rangle,\dots,\langle b_n\vartheta, t_n\rangle \in M$ for some $\vartheta \in Sub$, then $\langle h\vartheta, \langle\langle b_1\vartheta, t_1\rangle,\dots,\langle b_n\vartheta, t_n\rangle\rangle\rangle \in M$. Therefore, the corresponding Heyting semantics is: $\langle up(\wp(Tree)), T_P^H\rangle$ where for any $I \subseteq Tree$:

$$T_P^H(I) = \left\{ t \;\middle|\; \begin{array}{l} h \leftarrow b_1,\dots,b_n \in P,\ \vartheta \in Sub \\ \langle b_1\vartheta, t_1\rangle,\dots,\langle b_n\vartheta, t_n\rangle \in I \\ t = \langle h\vartheta, \langle\langle b_1\vartheta, t_1\rangle,\dots,\langle b_n\vartheta, t_n\rangle\rangle\rangle \end{array} \right\}$$

Proposition 3.5
There exist α and γ such that $\langle up(\wp(Atom)), T_P^c\rangle \overset{\gamma}{\underset{\alpha}{=}} \langle up(\wp(Tree)), T_P^H\rangle$ and $T_P^c\circ\alpha = \alpha\circ T_P^H$.

Thus, $\langle up(\wp(Tree)), T_P^H\rangle$ is a model complete collecting semantics. Notice however that the induced notion of collecting model for the Heyting semantics does not correspond to the notion of Heyting

[2] An application to dataflow analysis of a semantics similar to Clark's semantics is in [28].

model for a program. This is because collecting models in the sense of Definition 3.2 provide a classical view of the constructive (intuitionistic) semantics of Heyting models. This may justify the interest in the Heyting semantics as core semantics instead of Herbrand.

4 Abstract interpretation and abstract model theory

In this section we consider abstract interpretation of collecting semantics. A *property* for a collecting semantics $\langle C, T\rangle$ is any element in $uco(C)$. Let $\rho \in uco(C)$ such that $\rho(C) \cong_\imath A$. In the following we denote $(\alpha_\rho, \gamma_\rho)$ the pair adjoint functions $\alpha_\rho : C \rightarrow A$ and $\gamma_\rho : A \rightarrow C$ associated with the closure ρ, where $\alpha_\rho = \imath \circ \rho$ and $\gamma_\rho = \imath^{-1}$. It is known that $(C, \alpha_\rho, A, \gamma_\rho)$ is a Galois insertion (the Galois insertion induced by the closure ρ [13]).

Definition 4.1
Let $\mathcal{S}$ and $\mathcal{X}$ be collecting semantics such that $\mathcal{S}\underset{\alpha}{\overset{\gamma}{\rightleftharpoons}}\mathcal{X}$. A property ρ on $\mathcal{X}$ is extendible *to $\mathcal{S}$ iff $\alpha\circ\rho\circ\gamma$ is an upper closure operator on $\mathcal{S}$. $\alpha\circ\rho\circ\gamma$ is the* induced *property on $\mathcal{S}$.*

It is immediate to prove that if $\mathcal{S}$ and $\mathcal{X}$ are collecting semantics such that $\mathcal{S}\underset{\alpha}{\overset{\gamma}{\rightleftharpoons}}\mathcal{X}$, and ρ is a property on $\mathcal{X}$ such that $\gamma\circ\alpha \sqsubseteq_C \rho$, then ρ is extendible to $\mathcal{S}$.

Following the standard Cousot & Cousot's approach, for any program property we can define a notion of *abstract interpretation* of a collecting semantics. In what follows, the term "abstract interpretation" is clearly overloaded, corresponding both to the general framework of Cousot & Cousot, and to abstract semantic objects (interpretations). An abstract interpretation for a collecting semantics $\langle C, T\rangle$ with respect to a property ρ (denoted $\langle C, T, \rho, A, T^a\rangle$) is a semantics $\langle A, T^a\rangle$ such that $A \cong \rho(C)$ and $\alpha_\rho\circ T\circ\gamma_\rho \sqsubseteq_A T^a$. For a property ρ on a collecting semantics $\langle C, T\rangle$, the *best correct abstract interpretation* is therefore $\langle A, \alpha_\rho\circ T\circ\gamma_\rho\rangle$ for $A \cong \rho(C)$ ([13]).

Definition 4.2 *[abstract model-theoretic semantics]*
Let P be a logic program and $\langle C, T, \rho, A, T^a\rangle$ be an abstract interpretation. $M^a \in A$ is an abstract model for P iff there exists a collecting model M such that $\alpha_\rho(M) = M^a$ (or equivalently $M \sqsubseteq_C \gamma_\rho(M^a)$).

Abstract models capture the approximation induced by the abstract interpretation. An abstract interpretation is an abstract model if and only if it is the approximation of a collecting model. From the standard properties of Galois insertions it is easy to prove that if $M^a \in A$ and $\gamma_\rho(M^a)$ is a collecting model, then M^a is an abstract model.

Example 1 *The domain Dep (see [4]) was proposed by Marriott and Søndergaard as a domain of abstract substitutions. We lift it to the domain of abstract atoms. The domain is formalized as a Galois insertion denoted $(\wp(Atom), \alpha_p, Dep, \gamma_p)$ on the s semantics, and consists of equivalence classes of propositional formulae indexed on Π, constructed using the connectives $\leftrightarrow$ and $\wedge$, and ordered by implication. We say that a truth assignment ξ satisfies a propositional formula f, written $\xi \models f$, if $\xi(f)$ is a tautology. An object in Dep is a set of pairs $\langle p(\bar{x}), f\rangle$ where $\bar{x}$ are distinct variables and f is a prop-formula on $\bar{x}$, up to variable renaming. A truth assignment $assign_\theta = (\lambda x.var(\theta(x)) = \emptyset)$ is associated with a substitution θ, and $\gamma_p(\langle p(\bar{x}), f\rangle) = \{p(\bar{x})\theta \mid \theta' \leq \theta \Rightarrow assign_{\theta'} \models f\}$. We denote ρ_{Dep} the corresponding property on $\wp(Atom)$. Consider the program P {*`sum(X,0,X)`, `sum(X,s(Y),s(Z))` $\leftarrow$ `sum(X,Y,Z)`*}. Then $\{\langle sum(x, y, z), y \wedge (x \leftrightarrow z)\rangle\}$ is an abstract model for P. It is the abstraction of the corresponding collecting s-model: $\{sum(x, s^n(0), s^n(x)) \mid n \geq 0\}$.*

Define a *model complete abstract interpretation* as the abstract interpretation of any model complete collecting semantics. In the following, pre-fixpoints of the abstract operator T^a are called *reachable abstract models.*

Proposition 4.1
Let P be a logic program and $\langle C, T, \rho, A, T^a\rangle$ be a model complete abstract interpretation. Let $M^a \in A$ be an abstract interpretation. Then (1) if $T^a(M^a) \sqsubseteq_A M^a$ then M^a is an abstract model for the logic program P; and (2) if M^a is a reachable abstract model for P then $\gamma_\rho(M^a)$ is a reachable collecting model for P.

Definition 4.3 *[property completeness]*
Let ρ be a property for $\langle C, T\rangle$. $\langle C, T\rangle$ is ρ-complete iff $\rho\circ T\circ\rho = \rho\circ T$.

Proposition 4.2
Let $\langle C, T, \rho, A, T^a\rangle$ be a model complete abstract interpretation, where $\langle C, T\rangle_\alpha^\gamma$ is ρ-complete and $\rho\circ(\gamma\circ\alpha) = (\gamma\circ\alpha)\circ\rho$. If M^a is an abstract model of P, then $\gamma_\rho(M^a)$ is a collecting model of P.

Under the previous hypothesis: an abstract interpretation is an abstract model iff its concretization is a collecting model. In this case, ρ is a *model deformation* [7], i.e., ρ maps collecting models into collecting models.

4.1 Logic-based abstract compilation

In this section we relate model deformations (defined by abstraction) with program transformation for analysis, such as *abstract compilation.* In abstract compilation, the analysis is obtained by "transforming" the source program P into P' such that, when executed, P' returns precisely the desired dataflow information about P. Abstract compilation is then a *program deformation,* where the semantics of programs is not usually preserved but approximated. The idea is to study model deformations as an indirect way of studying abstract compilations. In particular we define abstract compilation as the class of programs for which the analysis is exact, i.e., where the analysis is *sound* (anything that can happen is predicted) and *minimal* (anything that is predicted can happen).

Definition 4.4 *[abstract compilation]*
Let P be a program and $\mu = \langle C, T, \rho, A, T^a\rangle$ be an abstract interpretation. The corresponding abstract compilation is $\Psi_P^\mu \subseteq$ Program, such that for each $P' \in \Psi_P^\mu$: $lfp(T(P'))|_{pred(P)} = \gamma(lfp(T^a(P)))$.

In the following, a collecting interpretation I is *finitely definite clause axiomatizable* (FDC as in [7]) if there exists $P \in$ *Program* such that $lfp(T(P)) = I$. Abstract compilation is then the collection of programs that provide a FDC axiomatization for $\gamma(lfp(T^a(P)))$.

Definition 4.5 *[FDC deformations [7]]*
Let $\langle C, T\rangle$ be a collecting semantics. $\delta : C \rightarrow C$ is a FDC deformation *iff whenever $I \in C$ is FDC axiomatizable, then $\delta(I)$ is FDC axiomatizable.*

Theorem 4.3
Let $\langle C, T, \rho, A, T^a\rangle$ be an abstract interpretation and P be a program. If $\gamma_\rho\circ T^a\circ\alpha_\rho$ is a FDC deformation, then for every $n \geq 0$, $\gamma_\rho(T^a(P)\uparrow n)$ is FDC axiomatizable.

Clearly, with terminating abstract interpretations μ (as those for program analysis), $\gamma_\rho(lfp(T^a(P)))$ is FDC axiomatizable, and therefore $\Psi_P^\mu \neq \emptyset$.

5 Systematic design of systems of collecting semantics

In this section we introduce a systematic approach to collecting semantics design in logic programming. Up to now, a collecting semantics is any concretization of a core semantics (e.g., the Herbrand semantics). Thus, an arbitrary collecting semantics for a program may be completely unrelated with the "real" program execution. Let $\langle C, T\rangle_\alpha^\gamma$ be a collecting semantics. A *correct collecting semantics* (with respect to $\langle C, T\rangle$) is a collecting semantics $\langle C', T'\rangle$ which is sound with respect to $\langle C, T\rangle$ (i.e., such that $\exists\alpha', \gamma'$: $\langle C', T'\rangle\underset{\alpha'}{\overset{\gamma'}{\rightleftharpoons}}\langle C, T\rangle$). Correct collecting semantics can be defined by abstraction of a *reference collecting semantics* if the property induced by the core semantics on the reference semantics can be extended to its abstractions, as stated in the following

Theorem 5.1
Let $\langle C, T\rangle_\alpha^\gamma$ be a (model complete) collecting semantics in $\mho_{\langle B, T^B\rangle}$ and $\langle C', T'\rangle$ such that $\exists\alpha', \gamma'$: $\langle C', T'\rangle\underset{\alpha'}{\overset{\gamma'}{\rightleftharpoons}}\langle C, T\rangle$. If $\gamma'\circ\alpha' \sqsubseteq_C \gamma\circ\alpha$ then $\langle C', T'\rangle$ is a (model complete) collecting semantics.

$$R1 : \frac{c = h \leftarrow b_1, \ldots, b_n \in P}{h \xrightarrow{c} \langle b_1, \ldots, b_n \rangle \in \mathcal{E}} \qquad R2 : \frac{\begin{array}{c} c = h \leftarrow b_1, \ldots, b_n \in P \\ \langle \langle a_i \xrightarrow{\bar{c}_i} \Lambda \rangle_{i=1}^{k-1}, a_k \xrightarrow{\bar{c}_k} \bar{b} \rangle \ll_c \mathcal{E} \ \ (1 \leq k \leq n) \\ \theta = mgu(\langle b_1, \ldots, b_k \rangle, \langle a_1, \ldots, a_k \rangle) \end{array}}{(h \xrightarrow{c} \langle b_1, \ldots, b_n \rangle \xrightarrow{\bar{c}_1} \ldots \xrightarrow{\bar{c}_k} \bar{b} :: \langle b_{k+1}, \ldots, b_n \rangle)\theta \in \mathcal{E}}$$

Table 1: AND-compositional traces

A natural choice for a reference semantics is the operational description of the computation process, in logic programming: *SLD* resolution. In the following we fix the Prolog left-to-right selection rule. The operational semantics of a logic program P is defined as a labeled transition system defining *SLD* resolution: $SLD = \langle State, \{\xrightarrow{c} \mid c \in P\} \rangle$, where states are pairs of goals and substitutions: $State = Atom^* \times Sub$, and transitions are labeled with program clauses: $\xrightarrow{c} \subseteq State \times State$, such that $\langle a \mid B, \sigma \rangle \xrightarrow{c} \langle body :: B, \vartheta \rangle$ iff $c = h \leftarrow body$ is a renamed apart clause in P, and $\vartheta = \sigma mgu(a\sigma, h)$. The transitive closure of $\longrightarrow$ is denoted $\xrightarrow{\bar{c}}{}^*$ where $\bar{c} \in P^*$. A collecting semantics can be derived from *SLD* by modeling execution traces. We denote $\mathcal{T}(SLD)$ the set of (finite) execution traces of *SLD*, with arbitrary elements π. π_0 denotes the first element of the trace π. $\mathcal{T}(SLD)$ is inductively defined by the rules:

$$\frac{s \in State}{s \in \mathcal{T}(SLD)} \qquad \frac{s \xrightarrow{c} \pi_0 \ \wedge \ \pi \in \mathcal{T}(SLD)}{s \xrightarrow{c} \pi \in \mathcal{T}(SLD)}$$

It is a common practice in logic program semantics to restrict the interest to *AND-compositional execution traces* only (e.g., any of the fixpoint semantics in [31, 20, 5, 22] provide AND-compositional denotations for logic programs). Intuitively a set of traces is AND-compositional if the execution trace of any (possibly non-atomic) goal can be reconstructed by composing traces for atomic goals in the set. The inductive definition of the set $\mathcal{E}$ of AND-compositional execution traces for atomic goals is in Table 1. It is a (positive) inductive definition with universe the set of traces from atomic goals only $\mathcal{T}_a(SLD)$. The first rule specifies an atomic transition from an atomic goal h with clause c. The second rule specifies the AND-compositionality of derivations for a clause c. This is obtained by composing the successful transitions for the first $k-1$ atoms of the body, with the state (goal) produced from a derivation of the k^{th} atom of the body.

Theorem 5.2 *[AND-compositionality]*
Let $G = b_1, \ldots, b_n$ be a goal. $\langle G, \sigma \rangle \xrightarrow{\bar{c}}{}^+ \langle B, \vartheta \rangle \in \mathcal{T}(SLD)$ iff $\exists \langle \langle h_i \xrightarrow{\bar{c}_i} \Lambda \rangle_{i=1}^{k-1}, h_k \xrightarrow{\bar{c}_k} B_k \rangle \ll_G \mathcal{E}$, $1 \leq k \leq n$ such that $\delta = mgu(\langle b_1, \ldots, b_k \rangle \sigma, \langle h_1, \ldots, h_k \rangle)$, $\bar{c} = \bar{c}_1 :: \ldots :: \bar{c}_k$, and $B\vartheta \sim (B_k :: \langle b_{k+1}, \ldots, b_n \rangle)\sigma\delta$ and $G\vartheta \sim G\sigma\delta$.

By Theorem 5.2, the set of SLD-traces $\mathcal{T}(SLD)$ can be characterized in terms of traces from atomic goals only, i.e., traces in $\mathcal{T}_a(SLD)$. The operator φ_P on $\wp(\mathcal{T}_a(SLD))$ induced by the inductive definition of $\mathcal{E}$ is ([1]):

$$\begin{array}{rcl} \varphi_P(X) & = & X \ \cup \ \{h \xrightarrow{c} \bar{B} \mid c = h \leftarrow \bar{B} \in P\} \ \cup \\ & & \left\{ \pi\theta \ \middle| \ \begin{array}{l} c = h \leftarrow b_1, \ldots, b_n \in P \\ \langle \langle a_i \xrightarrow{\sigma_i} \Lambda \rangle_{i=1}^{k-1}, a_k \xrightarrow{\sigma_k} \bar{B} \rangle \ll_c X \ (1 \leq k \leq n) \\ \theta = mgu(\langle b_1, \ldots, b_k \rangle, \langle a_1, \ldots, a_k \rangle) \\ \pi = h \xrightarrow{c} \langle b_1, \ldots, b_n \rangle \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_k} \bar{B} :: \langle b_{k+1}, \ldots, b_n \rangle \end{array} \right\} \end{array}$$

It is easy to prove that $\varphi_P(X)$ is continuous and $\mathcal{E} = \varphi_P \uparrow \omega$, i.e., the inductive definition is well formed. In the following we consider $\langle \wp(\mathcal{T}_a(SLD)), \varphi_P \rangle$ as *reference semantics*.

Proposition 5.3
$\langle \wp(\mathcal{T}_a(SLD)), \varphi_P \rangle$ is a model complete collecting semantics.

$\langle \wp(\mathcal{T}_a(SLD)), \varphi_P \rangle_\alpha^\gamma$ is therefore a collecting semantics where the abstraction function α maps any successful sequence $h \xrightarrow{\sigma}{}^* \Lambda$ into $ground(h)$ while non successful traces are simply ignored.
• The *angelic abstraction* α_a is obtained by approximating finite traces by the pair of their initial and final state (see [17]) enhanced with the sequence of clauses used in the trace. $\alpha_a(X) = \{\langle h, \bar{B}, \sigma \rangle \mid h \xrightarrow{\sigma}{}^* \bar{B} \in X\}$. It is easy to associate with α_a a concretization function γ_a inducing a Galois insertion such that $\gamma_a \circ \alpha_a \subseteq \gamma \circ \alpha$. By Theorem 5.1, the best correct angelic approximation is a model complete collecting semantics. This semantics has been recently used in [5] to model Prolog depth-first search.
• The *sequence abstraction* α_s simply ignores the sequence of clauses used in the trace. It can be composed with α_a to approximate traces with their initial and final state only. As before it induces a model complete collecting semantics for *partial answers*. The semantics for *call patterns* in [23] can be further derived by approximating $\langle h, b \mid \bar{B} \rangle$ with $\langle h, b \rangle$.
• The *success abstraction* α_{ss} approximates any finite successful trace with its initial state, while non successful traces are simply ignored. Notice that when composed with sequence abstraction, it induces a best correct approximation which is equivalent ($\simeq$) to $\langle \wp(Atom), T_P^s \rangle$. Moreover: $\alpha = ground \circ \alpha_{ss}$.
• Finally, the *Heyting abstraction* α_H is defined in terms of a map transforming successful (finite) traces into trees.

6 Combining semantics and properties

In this section we formally relate and combine collecting semantics. In order to specify the basic operators to combine collecting semantics, we require some *completeness* conditions about the involved system of semantics. A system of collecting semantics $\mho_{\mathcal{S}}$ is *(upper) complete* when there exists a (reference) collecting semantics $\mathcal{W} \in \mho_{\mathcal{S}}$ such that for each $\mathcal{Q} \in \mho_{\mathcal{S}}$, $\mathcal{Q}$ is sound with respect to $\mathcal{W}$. Complete systems are actually complete lattices. In the following we denote $\mho_{\mathcal{S}}^{\mathcal{W}}$ the system of *all* collecting semantics which can be derived by abstraction from $\mathcal{W}$ and having $\mathcal{S}$ as core semantics (i.e., generated by $\mathcal{S}$).

Theorem 6.1
Let $\mathcal{W} = \langle C, T \rangle_\alpha^\gamma$ be a collecting semantics in the system of collecting semantics with respect to $\mathcal{S}$. Then $\mho_{\mathcal{S}}^{\mathcal{W}}$ is a complete lattice isomorphic to the sub-lattice of $uco(C)$: $\langle \Gamma, \sqsubseteq_{uco}, \sqcap, \sqcup, \gamma \circ \alpha, \lambda x.x \rangle$ where $\Gamma = \{\rho \in uco(C) \mid \rho \sqsubseteq_{uco} \gamma \circ \alpha\}$.

In particular we can apply the join and meet operators in $uco(C)$ to compose collecting semantics. Let $\mathcal{W} = \langle C, T \rangle$ and $\{\rho_{\mathcal{A}_i}\}_{i \in I}$ be the closure operators associated with the semantics $\{\mathcal{A}_i\}_{i \in I} \in \mho_{\mathcal{S}}^{\mathcal{W}}$. We define: $\oplus_{i \in I} \mathcal{A}_i = \langle \sqcup_{i \in I} \rho_{\mathcal{A}_i}(C), (\sqcup_{i \in I} \rho_{\mathcal{A}_i}) \circ T \rangle$ and $\otimes_{i \in I} \mathcal{A}_i = \langle \sqcap_{i \in I} \rho_{\mathcal{A}_i}(C), (\sqcap_{i \in I} \rho_{\mathcal{A}_i}) \circ T \rangle$. By Theorem 6.1, $\langle \mho_{\mathcal{S}}^{\mathcal{W}}, \rightleftharpoons, \otimes, \oplus, \mathcal{S}, \mathcal{W} \rangle$ is a complete lattice.

Any set of collecting semantics $\mho$ can be extended to a system of collecting semantics by observing that any semantics in $\mho$ is complete with respect to $\oplus\mho$, which is actually the most concrete semantics having this property and which can be derived by abstraction from semantics in $\mho$. Therefore, for any set of semantic definitions $\mho$, then $\mho \cup \{\oplus\mho\}$ is always a system of collecting semantics. Analogously, any system of collecting semantics $\mho_{\mathcal{S}}$ can be extended to a corresponding (upper) complete system by observing that any semantics in $\mho_{\mathcal{S}}$ is correct with respect to $\otimes\mho_{\mathcal{S}}$. In this case $\mho_{\mathcal{S}} \cup \{\otimes\mho\}$ is complete.

Proposition 6.2
Let $\mathcal{W}$ be a possibly non model complete collecting semantics. The family of model complete collecting semantics in $\mho_{\mathcal{S}}^{\mathcal{W}}$ is inf-closed.

Therefore, the reduced cardinal product of model complete semantics is a model complete semantics. In particular, by Theorem 5.1 it is easy to prove the following

Corollary 6.3
If $\mathcal{W}$ is model complete, then any semantics in $\mho_{\mathcal{S}}^{\mathcal{W}}$ is model complete.

computational ordering & [illegible] abstract [illegible] are the same.

Therefore, the join of model complete collecting semantics is model complete provided that the reference semantics is a model complete semantics.

In the following of this section, for simplicity, we consider the system of all collecting semantics generated by Herbrand's semantics and correct with respect to SLD, i.e., $\mho^{SLD}_{\mathcal{H}}$. It is worth noting that all of the following results can be generalized to any complete system of collecting semantics. In particular, note that by Proposition 5.3 and Corollary 6.3, any semantics in $\mho^{SLD}_{\mathcal{H}}$ is model complete.

Example 2 *A new semantics including both the s and the Heyting semantics can be obtained by reduced cardinal product of the s and the Heyting semantics defined in Section 3.3:* $\langle \wp(Tree), T^{Hs}_P\rangle \simeq \langle \wp(Atom), T^s_P\rangle \otimes \langle up(\wp(Tree)), T^H_P\rangle$, *where for each* $I \subseteq Tree$:

$$T^{Hs}_P(I) = \left\{ t \;\middle|\; \begin{array}{l} t = \langle h, \langle\langle h_1, t_1\rangle, \dots, \langle h_n, t_n\rangle\rangle\rangle\theta \\ C = h \leftarrow b_1, \dots, b_n \in P,\ \langle\langle h_1, t_1\rangle, \dots, \langle h_n, t_n\rangle\rangle \ll_C I \\ \theta = mgu(\langle b_1, \dots, b_n\rangle, \langle h_1, \dots, h_n\rangle) \end{array} \right\}$$

By Proposition 6.2, $\langle \wp(Tree), T^{Hs}_P\rangle$ *is model complete. Moreover, we can verify that Clark's semantics is precisely the common semantics between s and Heyting, namely* $\langle up(\wp(Atom)), T^c_P\rangle \simeq \langle \wp(Atom), T^s_P\rangle \oplus \langle up(\wp(Tree)), T^H_P\rangle$.

6.1 The "best" collecting semantics for analysis

In the following, we characterize when a collecting semantics is *too concrete* for a given property, and the *best* collecting semantics for analysis. The following notion of *too concreteness* corresponds precisely to the existence of a collecting semantics which is more abstract but equivalent on the property to model.

Definition 6.1
A collecting semantics $\langle A', T'\rangle$ *is too concrete for a property* ρ *of* $\langle A', T'\rangle$ *iff there exists a collecting semantics* $\langle A, T\rangle$ *such that:* $\langle A, T\rangle \overset{\gamma}{\underset{\alpha}{\rightleftharpoons}} \langle A', T'\rangle$, ρ *is extendible to* $\langle A, T\rangle$, *and* $\rho\circ\gamma\circ T\circ\alpha\circ\rho = \rho\circ T'\circ\rho$.

Example 3 *Most of the (bottom-up) abstract interpretations designed for success pattern approximation are based on the abstraction of the s collecting semantics (e.g., [6, 10]). However, notice that the property Dep is extendible to Clark's semantics and* $\rho_{Dep}\circ T^s_P\circ\rho_{Dep} = \rho_{Dep}\circ T^c_P\circ up\circ\rho_{Dep}$, *i.e., the s collecting semantics is (too) concrete with respect to Dep. The more simple c semantics can be equivalently used as a base semantics for Dep abstraction. This is not true for Sharing [25], because the abstraction up which relates Clark and s semantics, may introduce new sharings which are not produced by the program.*

Intuitively, the *best* collecting semantics is a semantics which is not too abstract to lose useful information but also not too concrete (in view of Definition 6.1). More formally, given a collecting semantics $\langle C, T\rangle$, which is too concrete with respect to a given property ρ, we are interested in systematically derive an abstraction of $\langle C, T\rangle$ which leads to the best collecting semantics for that property.

Theorem 6.4 *[the best collecting semantics]*
Let $\langle C, T\rangle^{\gamma}_{\alpha}$ *be a collecting semantics. Let* ρ *be a property. The best collecting semantics is a semantics* $\langle C^b, T^b\rangle$ *such that* $C^b \cong \rho \sqcap (\gamma\circ\alpha)(C)$ *and* $T^b = \alpha_{\rho\sqcap(\gamma\circ\alpha)}\circ T\circ\gamma_{\rho\sqcap(\gamma\circ\alpha)}$.

When $\mathcal{S}$ is the core semantics, we will sometimes denote the best collecting semantics for a property ρ as $\mathcal{S} \sqcap \rho$. Therefore, given a property ρ, and a system $\mho^{\mathcal{W}}_{\mathcal{S}}$, we can always define an operator $\beta_\rho : \mho_{\mathcal{S}} \rightarrow \mho_{\mathcal{S}}$ which transforms any (possibly too abstract) semantics $\mathcal{X} \in \mho_{\mathcal{S}}$, into the least (most abstract) semantics which is more concrete than $\mathcal{Q}$ and suitable for the analysis, namely $\beta_\rho = \lambda\mathcal{X}.\mathcal{X} \sqcap \rho$. It is immediate to see that β_ρ is a closure operator on the complete lattice $\mho^{\mathcal{W}}_{\mathcal{S}}$, whose fixpoints are exactly the collecting semantics suitable for the analysis of ρ. We call these semantics *concrete semantics for* ρ. Thus, if $\mathcal{Q}$ and $\mathcal{X}$ are concrete semantics for a property ρ, (i.e., $\beta_\rho(\mathcal{Q}) = \mathcal{Q}$ and $\beta_\rho(\mathcal{X}) = \mathcal{X}$) then both $\mathcal{Q} \sqcup \mathcal{X}$ and $\mathcal{Q} \sqcap \mathcal{X}$ are concrete for ρ. By Theorem 5.1, the best collecting semantics construction maintains model completeness and correctness:

Corollary 6.5
Let $\langle C, T\rangle$ *be a collecting semantics and* $\langle C^b, T^b\rangle$ *be the corresponding best collecting semantics for a property, as stated in Theorem 6.4. Then, (1) if* $\langle C, T\rangle$ *is model complete, then* $\langle C^b, T^b\rangle$ *is model complete; and (2) if* $\langle C, T\rangle$ *is correct, then* $\langle C^b, T^b\rangle$ *is correct.*

Example 4 *It is easy to see that Clark's semantics is indeed the best collecting semantics for ground program properties (e.g.,* ρ_{Dep}*), namely it is isomorphic to the reduced cardinal product of Herbrand with Dep. To prove this it is sufficient to observe that, by fixing the s semantics as collecting semantics:* $\rho_{Dep} \sqcap ground = up$. *Note that, in the hypothesis of [3], i.e. assuming that* Σ *contains infinitely many constants, then the Herbrand and Clark's semantics are isomorphic (see [3]), namely:* $\mathcal{H}$ is the best collecting semantics for analysis of groundness! *By Theorem 6.4, the best semantics for Sharing is strictly more abstract than s and strictly more concrete than Clark's semantics. We are currently looking for its "explicit" definition.*

7 Related works

In logic programming, the most related works are [6, 15, 11, 21, 26]. [26] firstly applied a notion of *core semantics* to build collecting semantics. The approach however was neither oriented to a systematic design of semantics nor provided with a model theoretic interpretation for collecting and abstract semantics. While [15] firstly observed that $\mathcal{H}$ is an abstract interpretation of a more concrete *backward semantics*, [6] applied (only) the *s* collecting semantics to program analysis. In [21] $\mathcal{H}$, Clark and *s* semantics are related, providing a model theoretic interpretation for *s* models. We combine and extend those approaches in the first part of the paper, introducing a generic notion of collecting semantics for logic programs. This includes the results in [6, 21] as a special case of some of the results in Sections 3 and 4. Moreover, by using abstract interpretation to relate semantics, we can systematically derive and compare collecting semantics, and constructively define "optimal" collecting semantics for analysis. The approach is general enough to include also different semantics like Heyting semantics, semantics for call patterns *etc.* Independently, [11] also applied abstract interpretation to derive semantics by abstraction from SLD trees, similarly to Section 5. The main difference with our approach is that [11] does not consider a *core semantics* in concrete semantic definitions. The core semantics is here a key notion in order to extend to collecting and abstract semantics many of the (desirable) properties of the standard semantic definition of logic programs, like its simplicity and its model theoretic interpretation. Morover, it is essential to characterize both the *best* collecting semantics for analysis (Theorem 6.4), and the class of *correct* collecting semantics for a program.

Acknowledgments

The stimulating discussions with Patrick Cousot, Maurizio Gabbrielli, Thomas P. Jensen, Giorgio Levi, Francesco Ranzato and Enea Zaffanella are gratefully acknowledged.

References

[1] P. Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

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

[3] K. R. Apt and M. Gabbrielli. Declarative Interpretations Reconsidered. In P. Van Hentenryck, editor, *Proc. ICLP'94*, pages 74–89, 1994.

[4] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: algebraic properties and efficient representation. In B. Le Charlier, editor, *Proc. SAS'94*, LNCS 864, pages 266–280, 1994.

[5] R. Barbuti, M. Codish, R. Giacobazzi, and M. Maher. Oracle Semantics for PROLOG. In H. Kirchner and G. Levi, editors, *Proc. ALP'92*, LNCS 632, pages 100–114, 1992. Extended version to appear in *Information and Computation.*

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

[7] A. Batarekh and V. S. Subrahmanian. Topological model set deformations. *Fundamenta Informaticae*, 12:357–400, 1989.

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

[9] K. L. Clark. Predicate logic as a computational formalism. Technical Report Dept. of Computing, Imperial College, 1979.

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

[11] M. Comini and G. Levi. An Algebraic Theory of Observables. In M. Bruynooghe, editor, *Proc. ILPS'94*, 1994.

[12] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. ACM POPL'77*, pages 238–252, 1977.

[13] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. ACM POPL'79*, pages 269–282, 1979.

[14] P. Cousot and R. Cousot. Constructing hierarchies of semantics by abstract interpretation. Invited Lecture, *Workshop on Static Analysis*, WSA'92 Bordeaux 1992.

[15] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[16] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–549, 1992.

[17] P. Cousot and R. Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In *Proc. ACM POPL'92*, pages 83–94, 1992.

[18] S. K. Debray. Efficient Dataflow Analysis of Logic Programs. *JACM*, 39(4):949–984, 1992.

[19] S. K. Debray. On the Complexity of Dataflow Analysis of Logic Programs. In W. Kuich, editor, *Proc. ICALP'92*, LNCS 623, pages 505–520. 1992.

[20] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *TCS*, 69(3):289–318, 1989.

[21] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[22] M. Gabbrielli, G. Levi, and M. C. Meo. Observational Equivalences for Logic Programs. In K. Apt, editor, *Proc. JICSLP'92*, pages 131–145, 1992.

[23] M. Gabbrielli and M. C. Meo. Fixpoint Semantics for Partial Computed Answer Substitutions and Call Patterns. In H. Kirchner and G. Levi, editors, *Proc. ALP'92*, LNCS 632, pages 84–99, 1992.

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

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

[26] N. D. Jones and H. Søndergaard. A Semantics-based Framework for the Abstract Interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Ltd, 1987.

[27] R. Kemp and G. Ringwood. Reynolds and Heyting Models of Logic Programs. Technical report, Dept. of Computer Science, Queen Mary and Westfield College, 1991.

[28] K. Marriott and H. Søndergaard. Semantics-based Dataflow Analysis of Logic Programs. In G. Ritter, editor, *Information Processing 89*, North Holland, 1989.

[29] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[30] B. Steffen. Optimal data flow analysis via observational equivalence. In *Proc. MFCS'89*, LNCS 379, pages 492–502, 1989.

[31] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *JACM*, 23(4):733–742, 1976.

Contributions to a theory of existential termination for definite logic programs

Giorgio Levi and **Francesca Scozzari**
Dipartimento di Informatica,
Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy
{levi,scozzari}@di.unipi.it

Abstract

We suggest a new formalization of the existential termination problem of logic programs under the PROLOG leftmost selection rule and depth-first computation rule. First of all, we give a characterization of the problem in terms of occurrence sets, by proving that a ⟨*program*, *goal*⟩ existentially terminates if and only if there exists a finite correct occurrence set. Then we show that in order to study existential termination, we do not need to specify the occurrences of the atoms, since existential termination turns out to be decidable, when instances of atoms are used more than once (up to renaming).

We then reduce the verification of existential termination to the search of a suitable semi occurrence set for the pair ⟨*program*, *goal*⟩, by providing an algorithm for proving that the proposed semi occurrence set is a correct occurrence set. Finally we propose a simple method (based on abstract interpretation techniques) for generating such semi occurrence sets.

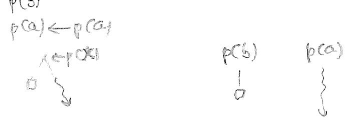
Keywords: existential termination, program analysis.

1 Introduction

The termination of logic programs became, in the last few years, an interesting research topic. The theories and techniques developed for studying the termination in imperative languages and term rewriting systems were used first. New termination techniques specific to logic programs have then been developed. In logic programs, we can make a distinction between universal and existential termination. A program is universally terminating for a given goal if the computation produces all the solutions and then terminates. It is existentially terminating for a goal if it either finitely fails or returns at least one solution. (finite search space.

Much work has been recently devoted to universal termination. For instance [1, 2, 3, 4] suggest new theoretical characterizations. In addition there are many proposals of effective methods, such as those in [12, 14, 15, 16], which infer interargument relations using AND/OR Dataflow Graphs, and those in [6, 7], where the termination is proved for programs enriched with assertions. Up to now, very little attention has been devoted to existential termination. [5] proposes a semantic approach using inductive proof techniques, [11] studies an effective method for function free programs, while [10, 19] just contain the basic definitions. Actually, whoever wants to develop an effective (and general enough) method for proving existential termination, has to deal with the impossibility of making any term abstraction, since existential termination does not enjoy any closure property, neither w.r.t. instantiation nor w.r.t. anti-instantiation. Moreover, the dependence on the computation rule forces us, in most cases, to choose a specific search strategy (usually the depth-first rule of PROLOG), which further complicates the analysis of the control because of the backtracking mechanism.

p(a, ~~x~~). p(~~X, X~~)



The development of existential termination techniques becomes essential in order to generalize the analysis of universal termination to normal programs. In fact, the problem of universal termination for negative goals boils down to the problem of existential termination for positive goals. An extension of the techniques for universal termination for dealing with negation is possible only if we have satisfactory results for existential termination.

In this paper we propose a new theoretical characterization of existential termination with the leftmost selection and depth-first computation rules. We also propose an abstract interpretation technique which can be useful to make our results applicable in practice.

In Section 2 we summarize the basic definitions and propose a technique of program indexing which will be used throughout the paper. In Section 3 we introduce occurrence sets, correct occurrence sets and their main properties, including an algebraic characterization in terms of complete lattices. We then introduce the minimal correct occurrence set which will be used in Section 4 to express our main results, which allow us to get rid of the occurrences in the occurrence sets, still preserving correctness and completeness w.r.t. existential termination, leading to the notion of semi occurrence sets of Section 5. Finally, in Section 6 we discuss an application of abstract interpretation which generates the semi occurrence sets and, in Section 7, show one simple example.

2 Preliminaries

2.1 Definitions

Let us first consider the classical definition of existential termination and some of its basic properties [5, 10]. The definition is based on the operational behavior of the program, using the standard concepts of computation and solution.

Definition 2.1 *A program p* **existentially terminates** *for the goal g if the computation either finitely fails or produces at least one solution.*

in the [illegible] of clauses   in the [illegible] of body

A basic property of existential termination is its dependence on both the selection rule (as in the case of universal termination) and the computation rule. Moreover, it does not enjoy the instantiation closure properties which are typical of universal termination. Namely the existential termination of a goal does not imply the termination of its instances. It is then necessary to study the existential termination in a goal-dependent way without the possibility of generalizing the results to instances or anti-instances [18].

In the discussion that follows, we shall consider logic programs with the PROLOG selection and search rules. We assume the reader to be familiar with the notions of SLD-resolution and SLD-tree. Moreover, we assume the concept of LD-resolution which uses the leftmost selection rule and the depth-first search rule. An LD-computation either terminates when it has found the first solution or diverges in the first infinite branch. An LD-tree is built by collecting from left to right the branches of the SLD-tree up to and including the first non finitely failed branch. Moreover, if the node n has associated the goal $A, \tilde{B}$, we define the selected atom in n as the atom A, according to the leftmost computation rule. For further definitions we refer to [13, 10].

$\langle p, g\rangle$ denotes a pair consisting of a definite program p and a goal g. Let I be a set of atoms. We denote by $up(I)$ the set of all the instances of atoms in I up to renaming. We denote by $\mathbb{N}^\infty$ the set $\mathbb{N} \setminus \{0\} \cup \{\infty\}$ obtained by extending the natural (positive) numbers and its ordering relation in the obvious way. Moreover, all our relations are defined up to renaming. The (atomic) symbol A will denote an atom or its equivalence class modulo renaming.

Existential termination can be expressed in terms of properties of LD-trees, as shown by the following proposition.

Proposition 2.2 *$\langle p, g\rangle$ existentially terminates if and only if the LD-tree of $\langle p, g\rangle$ is finite.*

2.2 Program indexing

A partial computation can be viewed as a sequence of goals $g_0 \to g_1 \to \ldots \to g_n$, where g_{i+1} is obtained from g_i and a (renamed apart) clause $c \equiv H : -A_1, \ldots, A_n$ by a single derivation step. In the approach we propose, it is necessary to identify the clause used in the resolvent in order to know which atoms in the program cause non-termination. In particular, for a goal $g = \leftarrow B_1, \ldots, B_n$, we want to know which clause the selected atom B_1 comes from. To this aim we propose a simple technique of program indexing. Obviously an atom in the head of a clause can never appear in a partial computation. Therefore only atoms in the body and in the goal need to be indexed.

Given a pair $\langle p, g\rangle$, the indexed program and goal is the pair $\langle p', g'\rangle$ obtained from $\langle p, g\rangle$, by indexing all the atoms in the body of any clause in p and in the goal g, so that all the indices are distinct. For the sake of simplicity, we shall use the natural numbers as index base. We associate then to every pair $\langle p, g\rangle$ a set of atoms $Atom_{\langle p,g\rangle}$ containing all the indexed atoms which are in p and in g.

Example 2.3 *Let p be the following program:*

```
q(a).
q(s) :- q(t),p(u).
```

and $g = \leftarrow q(x), p(y)$ be a goal. We define the program p'

```
q(a).
q(s) :- q₁(t),p₂(u).
```

the goal $g' = \leftarrow q_3(x), p_4(y)$ and $Atom_{\langle p,g\rangle} = \{q_1(t), p_2(u), q_3(x), p_4(y)\}$.

In the following, logic programs and goals will be indexed logic programs and indexed goals. The usual operations on atoms are extended in the obvious way: the mgu definition is not affected by the indexing, while the application of a substitution to an atom preserves the index of the atom itself. This extension aims to guarantee that every atom selected in the computation has an index. Therefore we define:

$$mgu(p_n(\tilde{t}), p_m(\tilde{u})) \overset{def}{=} mgu(p(\tilde{t}), p(\tilde{u}))$$

$$(p_n(\tilde{t}))\theta \overset{def}{=} p_n(\tilde{t}\theta)$$

We need to compare different atoms obtained as instances of the same atom in the program. Therefore we introduce a new equivalence relation which partitions the set of atoms on the basis of the associated index.

$$A \sim_{ind} B \overset{def}{\Leftrightarrow} A = p_n(\tilde{t}) \wedge B = p_n(\tilde{u})$$

Let A be an atom which is an instance of an atom in $Atom_{\langle p,g\rangle}$. We denote by $Atom_{\langle p,g\rangle}(A)$ the only element belonging to $Atom_{\langle p,g\rangle} \cap \{ B \mid B \sim_{ind} A \}$.

3 Occurrence sets and correct occurrence sets

The main idea underlying our construction is that one must be able to recognize the termination of a goal, by comparing an occurrence set for that goal with a specific partial computation. In practice, occurrence sets are used to guarantee the effectiveness of the analysis, allowing to compare the already visited atoms on the basis of the associated index. Therefore, our occurrence sets must contain both the visited atoms and their occurrences (i.e. how many times they occur in the computation).

We start by defining the domain of occurrence sets for a pair $\langle p, g\rangle$ and then stating the basic properties of correct occurrence sets. An occurrence set for $\langle p, g\rangle$ is a function which assigns to every atom A in $Atom_{\langle p,g\rangle}$ a set of pairs of the form $\langle A\theta, n\rangle$, where $n \in \mathbb{N}^\infty$. The second argument is called occurrence of the instance and can assume the value ∞.[1]

Definition 3.1 *We define* **occurrence set** *for $\langle p, g\rangle$ a function*

$$I_{\langle p,g\rangle} : Atom_{\langle p,g\rangle} \longrightarrow 2^{up(Atom_{\langle p,g\rangle})\times\mathbb{N}^\infty}$$

such that $\forall A \in Atom_{\langle p,g\rangle}\ \{\, B \mid \langle B, n\rangle \in I_{\langle p,g\rangle}(A)\,\} \subseteq up(A)$.

Therefore, an occurrence set can be thought of as a function whose image is a multiset, and likewise we can define the notion of correct occurrence set. The idea behind the concept of correct occurrence set is to collect a superset of the atoms selected in a (leftmost depth-first) derivation, together with an upper approximation of the occurrences.

Definition 3.2 *Let $I_{\langle p,g\rangle}$ be an occurrence set for $\langle p, g\rangle$. Then $I_{\langle p,g\rangle}$ is* **correct** *for $\langle p, g\rangle$ if and only if:*
$\forall A \in Atom_{\langle p,g\rangle}\ \forall\theta$ if $A\theta$ is selected n times in the LD-tree of $\langle p, g\rangle$, then $\exists m \geq n$ s.t. $\langle A\theta, m\rangle \in I_{\langle p,g\rangle}(A)$.[2]

Note that multiple occurrences of predicates in $Atom_{\langle p,g\rangle}$ are not identified. Every atom in the LD-tree comes from a specific atom in $Atom_{\langle p,g\rangle}$ and we can easily identify that atom thanks to the indexing.

Proposition 3.3 *Every pair $\langle p, g\rangle$ has a correct occurrence set.*

Proof: The occurrence set
$\forall A \in Atom_{\langle p,g\rangle}\ I(A) = up(Atom_{\langle p,g\rangle}) \times \{\infty\}$
is correct for any program and goal. ■

Among the occurrence sets, we are particularly interested in finite and weakly finite occurrence sets and in completions, in order to capture the concept of existential termination.

Informally, an occurrence set is finite if the set of occurrences of atoms in its image is finite. By weakening the definition, we obtain the characterization of weakly finite occurrence set. Namely we simply count the atoms in the image, by ignoring multiple occurrences (or better by assuming all the occurrences to be unary). On the contrary, the completion is obtained by increasing the size of the occurrence set by setting all the occurrences to infinity.

Definition 3.4 *Let $I_{\langle p,g\rangle}$ be an occurrence set. Then it is* **finite** *if the summation*

$$\sum_{A\in Atom_{\langle p,g\rangle}}\ \sum_{\langle B,n\rangle\in I_{\langle p,g\rangle}(A)} n$$

is finite. Likewise, an occurrence set is **weakly finite** *if*

$$\sum_{A\in Atom_{\langle p,g\rangle}}\ \sum_{\langle B,n\rangle\in I_{\langle p,g\rangle}(A)} 1$$

is finite.

[1] We prefer to use the notation $\langle A\theta, n\rangle$ rather than $\langle\theta, n\rangle$ in order to simplify the definition of equivalence up to renaming. Consider the atom $A = p(X, Y)$ and the substitutions $\theta = \{X \leftarrow f(Z)\}$, $\sigma = \{X \leftarrow f(W)\}$ and $\gamma = \{X \leftarrow f(Y)\}$. It turns out that $A\theta$ is equivalent to $A\sigma$ up to renaming but it is not equivalent to $A\gamma$, i.e. the renaming equivalence depends on the atom to which we apply the substitution. Therefore it is an equivalence relation which is parametric w.r.t. atoms.

[2] Since the relations are up to renaming, actually the definition should be: $\forall A \in Atom_{\langle p,g\rangle}\ \forall\theta$ if $A\theta$ appears n times (up to renaming) in the LD-tree of $\langle p, g\rangle$, then $\exists m \geq n$ s.t. $\langle B, m\rangle \in I_{\langle p,g\rangle}(A)$ where $A\theta$ is equivalent up to renaming to B.

Definition 3.5 *Let $I_{\langle p,g\rangle}$ be a weakly finite occurrence set. We call* **completion** *the occurrence set $Comp(I_{\langle p,g\rangle})$ defined as follows.*

$$Comp(I_{\langle p,g\rangle})(A) = \{\langle B, \infty\rangle \mid \langle B, n\rangle \in I_{\langle p,g\rangle}(A)\}\ \forall A \in Atom_{\langle p,g\rangle}.$$

Proposition 3.6 *The completion of a weakly finite occurrence set is weakly finite.*

Our characterization of termination is directly obtained by using the above definitions.

Theorem 3.7 *$\langle p, g\rangle$ existentially terminates if and only if there exists a finite correct occurrence set of $\langle p, g\rangle$.*

3.1 The lattice of occurrence sets

We analyze now the structure of the occurrence set $Int_{\langle p,g\rangle}$ for the pair $\langle p, g\rangle$ in order to be able to introduce the notion of minimal correct occurrence set, which will be used to state our main results. Let us start by defining two new operations, i.e. join and meet on occurrence sets.

Definition 3.8 *Let $I, J \in Int_{\langle p,g\rangle}$ be two occurrence sets. We define a new occurrence set $(I \sqcup J)(A)$ for all $A \in Atom_{\langle p,g\rangle}$ as follows.*

$$\begin{aligned}(I \sqcup J)(A) \overset{def}{=} \{\ & \langle B, n\rangle \mid \\ & 1.\ \langle B, x\rangle \in I \wedge \langle B, y\rangle \in J \wedge n = max\{x, y\}, \\ & 2.\ \langle B, n\rangle \in I \wedge \forall y\ \nexists\langle B, y\rangle \in J, \\ & 3.\ \langle B, n\rangle \in J \wedge \forall y\ \nexists\langle B, y\rangle \in I\,\}.\end{aligned}$$

Analogously we define the meet $(I \sqcap J)(A)$ for all $A \in Atom_{\langle p,g\rangle}$:

$$(I \sqcap J)(A) \overset{def}{=} \{\,\langle B, n\rangle \mid \langle B, x\rangle \in I \wedge \langle B, y\rangle \in J \wedge n = min\{x, y\}\,\}.$$

$Int_{\langle p,g\rangle}$ can be partially ordered by using the join operation. Let $I, J \in Int_{\langle p,g\rangle}$ be two occurrence sets. We define:

$$I \sqsubseteq J \overset{def}{\Leftrightarrow} I \sqcup J = J.$$

We generalize the join and meet operations to sets of occurrence sets in the obvious way. By using the above ordering, we prove that $\langle Int_{\langle p,g\rangle}, \sqsubseteq, \sqcup, \sqcap\rangle$ is a complete lattice. By exploiting the lattice structure we are able to state the first important result.

Theorem 3.9 *The meet of all the correct occurrence sets for the pair $\langle p, g\rangle$ is a correct occurrence set for $\langle p, g\rangle$. Such a occurrence set is called the minimal correct occurrence set and is unique.*

For the sake of simplicity, we denote the minimal correct occurrence set for a pair $\langle p, g\rangle$ by $I^{min}_{\langle p,g\rangle}$.

Theorem 3.10 *An occurrence set $I_{\langle p,g\rangle}$ is correct for $\langle p, g\rangle$ if and only if $I^{min}_{\langle p,g\rangle} \sqsubseteq I_{\langle p,g\rangle}$.*

Lemma 3.11 *The completion of a weakly finite, correct occurrence set is weakly finite and correct.*

Theorem 3.10 gives a correct occurrence set characterization, by using the ordering on the occurrence sets lattice. However the resulting characterization is obviously not effective because of the two following facts:

1. not all the correct occurrence sets have a finite representation;
2. the computation of $I^{min}_{\langle p,g\rangle}$ is not effective.

In order to solve the first problem, we shall consider weakly finite occurrence sets (and therefore weakly finite, correct occurrence sets) only, since they always have an effective representation. The above constraint automatically solves also the second problem, since the condition $I^{min}_{\langle p,g\rangle} \sqsubseteq I_{\langle p,g\rangle}$ turns out to be decidable, as shown in detail in the next Sections.

4 Towards a decision procedure based on weakly finite occurrence sets

Dealing with occurrence sets is definitely very close to the original model (the LD-tree) and results in a very simple theory. Actually, in order to capture existential termination, we can ignore occurrences and use sets of instances only. This intuition leads to a remarkable simplification in the construction of the minimal correct occurrence set, since we do not need to worry about the derivations which use a given instance infinitely many times. The result which allows us to obtain this simplification is the following. Let A be an atom which is selected at least twice in the LD-tree of $\langle p, g\rangle$. By inspecting the (finite) partial computation, included between the first two selections of the atom A, we are able to decide whether A existentially terminates w.r.t. the program p.

Theorem 4.1 *Let $A \in Atom_{\langle p,g\rangle}$ be such that $\exists\langle A\theta, n\rangle \in I^{min}_{\langle p,g\rangle}(A)$ with $n > 1$. Then it is decidable whether $\langle p, A\theta\rangle$ existentially terminates.*

Proof: By making a depth-first traversal of the LD-tree, let n_1 and n_2 be the first and the second node labeled $A\theta, \tilde{B}$ and $A\theta, \tilde{C}$ respectively (such nodes do exist, because $n \geq 2$ by hypothesis). We take under consideration the branch of the LD-tree starting from the root to the node n_2. We have two cases:

1. the node n_1 belongs to such a branch. This means that the goal $A\theta, \tilde{B}$ is rewritten to $A\theta, \tilde{C}$. We have again two cases:

   i the resolution of the atom $A\theta$ terminates with computed answer solution σ and the computation continues with the new goal $\tilde{B}\sigma$ (and $\tilde{B}\sigma$ is rewritten to $A\theta, \tilde{C}$). In this case $A\theta$ successfully existentially terminates.

   ii the goal $A\theta$ is rewritten to the goal $A\theta, \tilde{D}$ and therefore $A\theta$ diverges.

   Note that it is decidable whether the computation associated to $A\theta$ terminates, since we have to analyze a finite number of steps (i.e. the nodes included between n_1 and n_2).

2. the node n_1 does not belong to such a branch. This means that it necessarily belongs to a previously visited and finitely failed branch. Therefore the goal $A\theta$ either finitely fails or successfully terminates.

Since it is decidable whether n_1 belongs to a finite branch, the existential termination of $A\theta$ is decidable too. ■

By using the previous result we can characterize, in effective way, whether the completion of a weakly finite occurrence set represents a correct occurrence set.

Theorem 4.2 *Let $I_{\langle p,g\rangle}$ be a weakly finite occurrence set. Then it is decidable whether $I_C = Comp(I_{\langle p,g\rangle})$ is a correct occurrence set for $\langle p, g\rangle$, i.e. whether $I^{min}_{\langle p,g\rangle} \sqsubseteq Comp(I_{\langle p,g\rangle})$.*

Proof: The proof is given by the following algorithm (where Visit and Term are two sets of atoms).

```
let ⟨$D, \tilde{E}$⟩ be the label of the root of the LD-tree of ⟨$p, g$⟩,
    $f(\langle A, \tilde{C}\rangle, Visit, Term)$ =
    let $B = Atom_{\langle p,g\rangle}(A)$
    in if $\langle A, \infty\rangle \notin I_C(B)$
       then ⟨$I_C$ is not a correct occurrence set,a⟩
       else  if $A \in Visit \setminus Term$ and $A$ does not existentially terminate
             then ⟨$I_C$ is a correct occurrence set,b⟩
```

```
             else if the depth-first traversal of the LD-tree of ⟨$p, g$⟩
                     is terminated
                  then ⟨$I_C$ is a correct occurrence set,c⟩
                  else let ⟨$A', \tilde{C}'$⟩ be the label of the next selected node
                       in $f(\langle A', \tilde{C}'\rangle, Visit \cup \{A\}, Term \cup (\{A\} \cap Visit))$
in $f(\langle D, \tilde{E}\rangle, \emptyset, \emptyset)$.
```

In order to prove the correctness of the algorithm we have to show that:

i. all the choices are decidable.

- $\langle A, \infty\rangle \notin I_C(B)$. Since I_C is weakly finite, $I_C(B)$ is finite and therefore the condition turns out to be decidable.
- A does not existentially terminate. We test this condition only after having checked that $A \in Visit \setminus Term$, i.e. that A has already occurred in the visited subtree. Thus, A occurs at least twice in the minimal correct occurrence set of $\langle p, g\rangle$ and therefore, by theorem 4.1, it is decidable whether A existentially terminates.

ii. the algorithm always terminates. In the algorithm we check whether the selected atom belongs to the set $I_C(B)$. By definition of weakly finite occurrence set, the set $\{B \mid \exists A \in Atom_{\langle p,g\rangle}\langle B, n\rangle \in I_C(A)\}$ is finite. It follows that the algorithm considers finitely many distinct atoms. We have only to prove that no atom is selected infinitely many times. We assume, by contradiction, that there exists a set of atoms which are selected infinitely many times. This implies that at least one of these atoms recursively calls itself and therefore the atom does not existentially terminate. But this is a contradiction, since the algorithm should terminate with label b.

iii. it returns correct answers. It is worth noting that the sets $Visit$ and $Term$ contain, respectively, the already selected atoms and, among those which have been selected at least twice, those which existentially terminate. If the algorithm terminates with label a, I_C cannot be a correct occurrence set, because it does not contain the selected atom. If it terminates with label c, the complete LD-tree has been visited and all the atoms belong to I_C, and therefore we can assert that it is a correct occurrence set. The interesting case is the exit with label b. In this case there exists an atom A which has been visited at least twice and does not existentially terminate. Theorem 4.1 (part 1.ii.) states that A is rewritten to itself, and therefore the computation goes into an infinite loop where the selected atoms of the visited goals are all and only the atoms included between the atom A and its next invocation. Thus I_C is a correct occurrence set since it contains all these atoms with occurrence ∞.

■

Some remarks about the above theorems are necessary. The above results state that, for a weakly finite occurrence set, we can decide whether its completion is correct. Furthermore, we have proved that if a weakly finite occurrence set is correct, then, by theorem 3.10, also the completion is correct, but the vice versa is not valid. This means that, when moving from an occurrence set to its completion, the chances to obtain a correct occurrence set do increase. We can thus get rid of weakly finite occurrence sets and take into consideration their completions only, since we know that we have more chances to find a correct occurrence set in this class of occurrence sets. In practice we can neglect the occurrences and consider weakly finite occurrence sets only, without caring about atoms with multiple occurrences. This is captured by the notion of semi occurrence set.

5 Semi occurrence sets

Definition 5.1 *A* semi occurrence set *for $\langle p, g\rangle$ is a function*

$$I_{\langle p,g\rangle} : Atom_{\langle p,g\rangle} \rightarrow 2^{up(Atom_{\langle p,g\rangle})},$$

$I_{p,g}(A)$ is

with the constraint that $\forall A \in Atom_{\langle p,g\rangle}$ $I_{\langle p,g\rangle}(A) \subseteq up(A)$ and finite.

Note that the set of semi occurrence sets and the set of completions are isomorphic. Therefore we can embed the structure of semi occurrence sets in the lattice of occurrence sets. To this aim we define the extension of a semi occurrence set as the occurrence set obtained by setting all the occurrences to infinity.

Definition 5.2 *Let I be a semi occurrence set for $\langle p, g\rangle$. The occurrence set $Ext(I)$ is defined as:*

$$\forall A \in Atom_{\langle p,g\rangle} \; Ext(I)(A) \stackrel{def}{=} \{\langle B, \infty\rangle \mid B \in I(A)\}.$$

By exploiting the concept of semi occurrence set, theorem 4.2 becomes:

Theorem 5.3 *Let $I_{\langle p,g\rangle}$ be a semi occurrence set. Then it is decidable whether $Ext(I_{\langle p,g\rangle})$ is a correct occurrence set for $\langle p, g\rangle$, i.e. whether $I^{min}_{\langle p,g\rangle} \sqsubseteq Ext(I_{\langle p,g\rangle})$.*

Our task is therefore to provide a semi occurrence set and, by using the above theorems and algorithms, to check whether the correct occurrence set is finite and therefore to infer whether $\langle p, g\rangle$ existentially terminates.

In order to achieve our aim, we only have to check whether the finiteness of weakly finite, correct occurrence set is decidable. The characterization we obtain is still stronger and allows us to decide, for a weakly finite, correct occurrence set, whether the minimal correct occurrence set of $\langle p, g\rangle$ is finite, and therefore if there exists a finite correct occurrence set. The next theorem guarantees the effectiveness of such a check by using the previously specified algorithm.

Theorem 5.4 *Let $I_{\langle p,g\rangle}$ be a semi occurrence set such that $Ext(I_{\langle p,g\rangle})$ is a correct occurrence set for $\langle p, g\rangle$. Then it is decidable whether $I^{min}_{\langle p,g\rangle}$ is finite.*

Proof: The proof is based on the algorithm given in theorem 4.2. We use such an algorithm and the occurrence set $Ext(I_{\langle p,g\rangle})$. Since the occurrence set is correct, it turns out that the algorithm terminates either with label b or c. If the algorithm terminates with label b, we may conclude that $I^{min}_{\langle p,g\rangle}$ can not be finite since it contains at least an atom which does not existentially terminate. If the algorithm terminates with label c, the minimal correct occurrence set is finite since the LD-tree of $\langle p, g\rangle$ is finite. ∎

6 Generation of the semi occurrence set

As a conclusion, we suggest a method to automatically generate the semi occurrence sets. The basic idea is to generate an approximation of the atoms selected in the LD-tree, by a refinement of the depth-k abstraction ([17]) used for program analysis based on abstract interpretation ([8, 9]). Since existential termination does not enjoy any closure property, we are forced to compute an abstract denotation depending on both the program and the goal. To this end, we shall use a (leftmost) top-down construction based on the depth-first search rule. We start defining some useful functions and the abstract domain used by the fixpoint operator.

Let us consider the function $|\ldots| : Term \rightarrow \mathbb{N}$, such that

$$|t| = \begin{cases} 1 & \text{if } t \text{ is a constant or a variable} \\ max\{|t_1|, \ldots, |t_n|\} + 1 & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

and a given positive integer k. The associated equivalence relation is $\sim_k$, such that $t_1 \sim_k t_2$ iff $\alpha_k(t_1) = \alpha_k(t_2)$, where $\alpha_k(t)$ represents the term which can be obtained from the concrete one by substituting with a fresh variable each subterm t' in t, such that $|t| - |t'| = k$.

We can define the abstract universe as the image of α_k, that is $Term^a = \alpha_k(Term) = \{[t]_{\sim_k} : t \in Term\}$ and the abstract base $Atom^a = \{p(t^a_1, \ldots, t^a_n) : p \in \Pi, t^a_i \in Term^a\}$. The abstract semantics is defined as a set of pairs of the form $\langle g, h\rangle$, where g is the current goal and h is the sequence of renamed apart clauses used in a partial (leftmost depth-first) derivation starting from g. We shall refer to h as the history of the (partial) derivation. It follows that the abstract interpretation domain is simply $\mathcal{P}((Atom^a)^* \times Clause^*)$.

The next step concerns the abstract domain ordering. First of all, we define the structure of the program and the ordering on sequences of clauses. A program p is defined as a sequence of clauses $p = c_1 :: \ldots :: c_n$. The ordering on the clauses reflects the position in the program, i.e. $\forall i, j \; c_i < c_j \Leftrightarrow i < j$. We extend the ordering on sequences of clauses. Given a clause $c \in Clause$ and $s, s', s'' \in Clause^*$ we define

$$c :: s < s' \Leftrightarrow \begin{array}{l} 1) \; s' = \lambda \text{ or} \\ 2) \; s' = d :: s'' \text{ and } c < d \\ 3) \; s' = d :: s'' \text{ and } c = d \text{ and } s < s''. \end{array}$$

Note that the empty sequence λ is the greatest element. We order the pairs $\langle g, h\rangle$ according to the ordering on the sequences, i.e. by comparing only the histories of the derivation: $\langle g, h\rangle < \langle g', h'\rangle \Leftrightarrow h < h'$. This allows us to implement exactly the depth-first search strategy.

The abstraction function α_k and the equivalence relation $\sim_k$ are extended on $Atom$ and $Atom^*$ in the obvious way, that is $\alpha_k(p(t_1, \ldots, t_n)) = p(\alpha_k(t_1), \ldots, \alpha_k(t_n))$ and $A_1, \ldots, A_n \sim_k B_1, \ldots, B_n \Leftrightarrow \forall i = 1 \ldots n \; \alpha_k(A_i) \sim_k \alpha_k(B_i)$.

We define the equivalence relation $\sim_k$ on pairs, in such a way that two pairs are in $\sim_k$-relation if and only if the selected atoms in the goals are in the same relation.

$$\langle g, h\rangle \sim_k \langle g', h'\rangle \Leftrightarrow (g = A, \tilde{A} \text{ and } g' = B, \tilde{B} \text{ and } A \sim_k B) \text{ or } g = g' = \lambda.$$

It is straightforward to extend α_k on the abstract domain and to define a corresponding Galois insertion $\langle \alpha_k, \gamma_k\rangle$ such that $\gamma_k(I) = \{A \in Atom : \alpha_k(A) \in I\}$.[3]

The next step concerns the abstract fixpoint operator. First of all, we define a function

$$SLD_{Left} : \mathcal{P}((Atom)^* \times Clause^*) \rightarrow \mathcal{P}((Atom)^* \times Clause^*)$$

which computes all the pairs reachable by one step of SLD-resolution via the leftmost selection rule.

$$SLD_{Left}(I) = \{\langle (\tilde{B}, \tilde{A})\theta; h :: c\rangle \mid \begin{array}{l} \langle A, \tilde{A}; h\rangle \in I \\ c \equiv H : -\tilde{B} \in p \\ \theta = mgu(A, H)\}. \end{array}$$

In order to recognize successful computations, we define the function Id_λ which returns the pairs with empty goal: $Id_\lambda(I) = \{\langle \lambda; h\rangle \mid \langle \lambda; h\rangle \in I\}$.

Finally, the abstract semantics is obtained by computing, at each step, the set $SLD_{Left}(I)$ minus the pairs already computed in the previous steps, (possibly) augmented with a success pair. The least element of this set, when it exists, is exactly the result obtained by one step of SLD-resolution via the leftmost selection and depth-first search rules. The definition of the $T_{p,g}$ become:

[3] Note that we do not need to prove that the abstract semantics is correct w.r.t. a suitable concrete one, just because our semi occurrence sets are not necessarily correct occurrence sets. However, one can easily prove that our abstract semantics is a Galois insertion of the more concrete one, which can be obtained as a limit with $k = \infty$.

$$T_{p,g}(I) = I \cup min((SLD_{Left}(I) \setminus I) \cup Id_\lambda(I))$$

where $min(\emptyset) = \emptyset$[4]. Using a top-down construction, we have to start with the goal g and history λ, just because in the sequence ordering the empty history represents the top element. The least fixpoint is then computed in the following way:

$$T_{p,g} \uparrow_k 0 = \{\langle g, \lambda \rangle\}$$

$$T_{p,g} \uparrow_k (n+1) = \begin{cases} \alpha_k(T_{p,g} \uparrow_k n) & \text{if } T_{p,g}(T_{p,g} \uparrow_k n) \sim_k T_{p,g} \uparrow_k n \\ T_{p,g}(T_{p,g} \uparrow_k n) & \text{otherwise,} \end{cases}$$

where $\alpha_k(I) = \{\langle \alpha_k(g), h \rangle : \langle g, h \rangle \in I\}$ and $I \sim_k J \Leftrightarrow \forall p \in I\ \exists q \in J\ p \sim_k q\ and\ \forall p \in J\ \exists q \in I\ p \sim_k q$.

Note that the $T_{p,g} \uparrow_k$ operator always terminates just because the partition induced by the $\sim_k$-relation on the abstract domain is finite.

Given the above abstract semantics, we extract the semi occurrence set $I_{\langle p,g \rangle}$, by simply collecting all the selected atoms and getting rid of the history of the derivation, i.e.

$$I_{\langle p,g \rangle} = \{A \mid \langle A, \tilde{A}; h \rangle \in T_{p,g} \uparrow_k \omega\}.$$

7 Example

k is the maximum between the deepest term in the program and the deepest term in the input.

Let p be the following program:

```
visit([1|X]).

visit([0|Y]) ← append₁(Y,[0],Z),visit₂(Z).

append([],S,S).

append([T|U],V,[T|W]) ← append₃(U,V,W).
```

The behavior of the program w.r.t. a given goal $visit(t)$ is the following:

1. if $t = [t_1, \ldots, t_n \mid X]$ then $visit(t)$:

   i infinitely fails $\Leftrightarrow \forall 1 \leq i \leq n\ t_i = 0$;

   ii finitely fails $\Leftrightarrow \exists 1 \leq j \leq n\ \forall 1 \leq i < j\ t_i = 0$ and $t_j \neq 0, 1$ and t_j is not a variable;

   ii succeeds $\Leftrightarrow \exists 1 \leq j \leq n\ \forall 1 \leq i < j\ t_i = 0$ and ($t_j = 1$ or t_j is a variable);

2. $visit(t)$ finitely fails otherwise.

We choose $k = min\{i \mid p \sim_k \alpha_k(p)$ and $g \sim_k \alpha_k(g)\}$. The semi occurrence set extracted from $T_{p,g} \uparrow_k \omega$ is powerful enough to allow the algorithm to state the (non-)termination for every goal[5].

[4]Note that $<$ is a total order on $\mathcal{P}((Atom^a)^* \times Clause^*)$ and therefore min is always defined.

[5]It is worth noting that, in some cases, the technique is able to decide about the non-termination of a goal. In fact, when the algorithm terminates with label b, we can infer that the minimal correct occurrence set is infinite and therefore that the goal does not existentially terminate.

8 Conclusions

Our theory for capturing the notion of existential termination applies to positive logic programs, under the leftmost and depth-first rules and, according to the well-known properties of termination, can be applied to study the universal termination of negative goals in normal logic programs.

An interesting extension is related to the possibility of using different selection rules. In fact, the choice of the selection rule affects only the notion of LD-tree, while all our results are still valid. On the contrary, the choice of a different search rule, would affect the validity of theorem 4.1, on which our technique is based. Nevertheless, we believe that our result can be generalized to any search rule.

As a final remark, it is worth noting that our results can be applied in a "reverse" mode. Namely, in order to prove that $\langle p, A \rangle$ existentially terminates, we can look for a goal g such that the minimal correct occurrence set of $\langle p, g \rangle$ contains the atom A with occurrence ≥ 2. In such a way, we could reduce the problem of existential termination of A to the search of a goal with given properties.

References

[1] K. R. Apt and D. Pedreschi. Studies in Pure Prolog: Termination. In J. W. Lloyd, editor, *Computational Logic*, pages 150–176. Springer-Verlag, Berlin, 1990.

[2] K. R. Apt and D. Pedreschi. Proving Termination of General Prolog Programs. In T. Ito and A.R. Meyer, editors, *Proc. of Int. Conf. on Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 265–289. Springer-Verlag, Berlin, 1991.

[3] K.R. Apt and D. Pedreschi. Reasoning about Termination of Prolog Programs. Technical Report TR-14/91, Dip. di Informatica, Univ. di Pisa, 1991.

[4] K.R. Apt and D. Pedreschi. Modular Termination Proofs for Logic and Pure Prolog Programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Clarendon Press - Oxford, 1994.

[5] M. Baudinet. Proving Termination Properties of Prolog Programs: A Semantic Approach. *Journal of Logic Programming*, 14:1–29, 1992.

[6] A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. TAPSOFT'91*, volume 494 of *LNCS*, pages 153–180. Springer-Verlag, Berlin, 1991.

[7] A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and Their Use in Proving Universal Termination of a Logic Programs. *TCS*, 124(2):297–328, 1994.

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

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

[10] D. De Schreye and S. Decorte. Termination of Logic Programs: the Never-Ending Story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[11] N. Francez, O. Grumberg, S. Katz, and A. Pnueli. Proving Termination of Prolog Programs. In R. Parikh, editor, *Logics of Programs*, volume 193 of *LNCS*, pages 89–105. Springer-Verlag, 1985.

[12] G. Gröger and L. Plümer. Handling of Mutual Recursion in Automatic Termination Proofs for Logic Programs. Unpublished draft.

[13] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987.

[14] L. Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, Berlin, 1990.

[15] L. Plümer. Termination Proofs for Logic Programs based on Predicate Inequalities. In D. H. D. Warren and P. Szeredi, editors, *Proc. Seventh ICLP*, pages 634–648. The MIT Press, 1990.

[16] L. Plümer. Automatic Termination Proofs for Prolog Programs Operating on Nonground Terms. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 503–517. The MIT Press, 1991.

[17] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[18] F. Scozzari. Analisi di terminazione mediante interpretazione astratta. Master's thesis, Dipartimento di Informatica, Università di Pisa, 1994. in italian.

[19] T. Vasak and J. Potter. Characterization of Terminating Logic Programs. In *Proc. Third IEEE ILPS*, pages 140–147. IEEE Comp. Soc. Press, 1986.

A case study in logic program verification: the Vanilla metainterpreter

Dino Pedreschi and Salvatore Ruggieri

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail: {pedre,ruggieri}@di.unipi.it

Abstract

We take the formal verification of the Vanilla metainterpreter as an excuse for explaining a proof method for reasoning about logic programs. The choice of a semantics suitable for program verification is discussed. We consider a variant of the *least Herbrand model* semantics which abstracts from ill-typed atoms and the underlying (first order) language, thus enhancing modularity and ease of specification. Then, *proof outlines* and *proof obligations* are introduced in a Hoare's logic style. In the resulting proof theory, triples of the form $\{Pre\}P\{Post\}$ can be derived for a program P, which allow us to establish partial and total correctness. As a consequence of our results, the correctness of Vanilla is directly proved (once again.)

Keywords: Verification, program development, metaprogramming, formal methods.

1 Introduction

Logic programming (and Prolog) is advertised as a *declarative* language, in the sense that *specifications*, when written in an appropriate syntax, can be directly used as *programs*. This ideal situation, however, is often unrealistic in practice, both for the fact that executable specifications may be extremely inefficient, and for problems related to the implementations of logic programming—programs may fail to terminate or may result in errors when executed with particular strategies.

It is therefore needed to assess the correctness of a logic or Prolog programs with respect to its specification, or *intended interpretation*: a task that has been underestimated in the literature until recently. Of course, when verifying a logic program P, it would be helpful to use its declarative semantics, namely its *least Herbrand model* M_P. A natural approach consists of considering M_P as the intended specification — therefore, the verification of a program is viewed as checking that the intended specification of program and its least Herbrand model do coincide.

This approach, however, turns out to be inadequate: strangely enough, the least Herbrand models semantics is not sufficiently abstract. The justification of this statement is based on two considerations.

On the one hand, specifications generally deal with top level procedures only, leaving unspecified the description of auxiliary procedures. This is the case in top-down program development and modular programming.

On the other hand, the absence of types implies that the least Herbrand model is generally dirty with ill-typed atoms. Consider the APPEND program and its specification taken from [SS86]:

append(xs, ys, zs) $\leftarrow$ zs is the concatenation of the lists xs and ys.

```
append( [], Xs, Xs ).
append( [X|Xs], Ys, [X|Zs] ) ←
   append( Xs, Ys, Zs ).
```

The APPEND program is intuitively correct with respect to its specification but (if there are sufficiently many symbols in the language) its *intented interpretation* is not a model of the program.

In fact, in the least Herbrand model ill-typed atoms appear, such as `append([], foo, foo)`. For efficiency reasons run-time type-checks are dropped.

As a consequence, reasoning about the whole least Herbrand model implies having to take into account ill-typed atoms, thus making the specification complex and counter-intuitive. This problem becomes much harder in modular program development, since adding more symbols to the language in the upper modules entails changing the least Herbrand model of lower modules, and hence their correctness properties.

A clear point emerges from the previous discussion: a semantics for verification should take the *intended* or *well-typed* queries into account.

In this paper, we survey an approach to logic program verification which is based on the *intended* or *well-typed* fragment of the least Herbrand models [PR95, Rug94]. To illustrate the proposed proof method and its ability to support modular program development, we tackle the problem of verifying a classic of logic programming, namely the `Vanilla` meta-interpreter.

A word on terminology is in order. Throughout the paper we use the standard notation of logic programming, as in [Llo87, Apt90], when not specified otherwise. For instance, we use queries instead of goals and consider a fixed language L in which programs and queries are written. Ambivalent syntax is allowed, in the sense that function and predicate symbols may overlap [KJ95, Jia94]. By $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ we denote a ground instance of a clause from P, and by $bag(a_1, \ldots, a_n)$ the multiset consisting of elements $a_1, \ldots, a_n$. Given a Herbrand interpretation I and a query Q we write $I \models Q$ if I is a model of Q. In particular, if A is a ground atom then $I \models A$ iff $A \in I$.

Specifications and Semantics

Following a Hoare's logic style of defining partial and total correctness, we stipulate that a *specification* is a pair $Pre, Post$ of (Herbrand) interpretations, i.e., subsets of B_L.

The rationale under this choice is the following. The first interpretation, Pre, specifies the *intended*, or *well-typed* one-atom queries, i.e., those queries for which we designed the program under consideration. The second interpretation, $Post$, specifies some property of successful one-atom queries. In this sense, a specification $Pre, Post$ describes the input-output behavior of a logic program, in a way that closely resembles that in Hoare's logic. According to this choice, the *well-typed* fragment of the least Herbrand model is $M_P \cap Pre$.

We are now ready to define our notions of (weak) partial and (weak) total correctness.

Definition 1.1 Let P be a logic program, and $Pre, Post$ a specification.

(i) P is *partially correct* w.r.t. a specification $(Pre, Post)$ iff $M_P \cap Pre = Post$.

(ii) P is *weak partially correct* w.r.t. a specification $(Pre, Post)$ iff $M_P \cap Pre \subseteq Post$.

(iii) P is *totally correct* w.r.t. a specification $(Pre, Post)$ iff $M_P \cap Pre = Post$ and $Pre \subseteq M_P \cup FF_P$, where FF_P is the finite failure set of P.

(iv) P is *weak totally correct* w.r.t. a specification $(Pre, Post)$ iff $M_P \cap Pre \subseteq Post$ and $Pre \subseteq M_P \cup FF_P$. □

As a consequence of this definition, partial (and total) correctness of a program P w.r.t. a specification $Pre, Post$ entails that $Post$ coincides with the well-typed fragment of M_P. In the next section we introduce a proof theory for partial (and total) correctness which, in particular, will allow us to abstract away from the ill-typed fragment of M_P. Notice that the *weak* version of either notions entails that $Post$ specifies a property of $M_P \cap Pre$. Observe that the condition $Pre \subseteq M_P \cup FF_P$ used to define (weak) total correctness is equivalent to require the *existential termination* of the atoms in the precondition w.r.t. a fair selection rule and a complete search strategy.

As an example, observe that `APPEND` is totally correct w.r.t. $Pre, Post$ where

$$Pre = \{ \texttt{append}(xs,\ ys,\ zs) \mid xs, ys, zs \text{ are lists} \}$$
$$Post = \{ \texttt{append}(xs,\ ys,\ zs) \mid zs \text{ is the concatenation of the lists } xs \text{ and } ys \}.$$

Vanilla

A jewel of Logic Programming is the elegant meta-program which, by means of the ambivalent syntax, specifies the meta-circular interpreter (i.e., the interpreter of LP in LP). This program, referred to as the `Vanilla` metainterpreter, and first introduced in [BK82], is denoted by Van_P when instantiated on an object program P, and consists of the following clauses:

```
prove( true ).
prove( A & B ) ←
    prove( A ),
    prove( B ).
prove( A ) ←
    clause( A , B ),
    prove( B ).
clause( A , B1 & ... & Bn ).          for every A ← B1, ..., Bn ∈ P
```

where `B1 & ... & Bn` is an abbreviation for

- `B1 & ( B2 & ...( Bn-1 & Bn ) ...)` , if $n > 1$,
- `B1` , if $n = 1$
- `true` , if $n = 0$.

For instance, for P = `APPEND` the definition of `clause` is

```
clause( append([], Xs, Xs) , true ).
clause( append([X|Xs], Ys, [X|Zs]) , append(Xs, Ys, Zs) ).
```

Before analyzing the meta-interpreter, we have to agree on a notion of correctness of `Vanilla` w.r.t. the semantics of the object program P. We adopt the following criterion, which states that, for the intended queries, provability at the object level and the meta level coincide.

Definition 1.2 The `Vanilla` instantiated by P is correct w.r.t. $Pre \subseteq B_L$ iff for every $A \in Pre$ we have

$$A \in M_P \quad \text{iff} \quad \texttt{prove}(A) \in M_{Van}$$ □

In the literature, the case $Pre = B_P$ is generally considered [Kal95, BT95]. We require that the symbols `&` and `true` are not predicate symbols of L, i.e., object level predicates. Indeed, without this assumption the `Vanilla` is not correct. Consider, in fact, the program `q(a).` , `p(b).` , `&(p(c),p(c)).` Then `prove(&(p(b),q(a)))` is in M_{Van} whereas the atom `&(p(b),q(a))` is not in M_P. A similar argument applies to the program `true ← true`.

2 Proof Theory

We now introduce a proof method for the various notions of correctness. The aim of this section is to introduce the concept of (Hoare's logic style) triples $\vdash \{Pre\}\ P\ \{Post\}$ (for programs) and $\vdash \{Pre\}\ Q\ \{Post\}$ (for queries), which are the basic tool to prove correctness. As a specification $Pre, Post$ of a program is assigned in terms of sets of ground atoms, we can simply reason about ground instances of program clauses and queries. First, the following notion of a level mapping is needed.

DEFINITION 2.1 A *level mapping* (on L) is a function $|\ |: B_L \longrightarrow N$ of ground atoms to natural numbers. □

DEFINITION 2.2 Consider a program P and a query Q. Given a specification $(Pre, Post)$ we write

- $\vdash_t \{Pre\}\ P\ \{Post\}$ iff there exists a level mapping $|\ |$ such that for every $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$

  1. for $i \in [1,n]$
  $Pre \models A \ \wedge\ Post \models B_1, \ldots, B_{i-1} \Rightarrow$
    (a) $Pre \models B_i$ and
    (b) $|A| > |B_i|$

  2. $Pre \models A \ \wedge\ Post \models B_1, \ldots, B_n \Rightarrow Post \models A$

  We write $\vdash \{Pre\}\ P\ \{Post\}$ when *(1a)* and *(2)* hold. *Pre* is called a precondition and *Post* a postcondition.

- $\vdash \{Pre\}\ Q\ \{Post\}$ iff for every ground instance $A_1, \ldots, A_n$ of it

  3. for $i \in [1,n]$ $\quad Post \models A_1, \ldots, A_{i-1} \Rightarrow Pre \models A_i$

- $\vdash_t \{Pre\}\ Q\ \{Post\}$ iff there exist a level mapping $|\ |$ and $k \in N$ such that for every ground instance $A_1, \ldots, A_n$ of it

  4. for $i \in [1,n]$ $\quad Post \models A_1, \ldots, A_{i-1} \Rightarrow Pre \models A_i \ \wedge\ k > |A_i|$ □

The relation $\vdash$ is devised for proving (weak) partial correctness, whereas $\vdash_t$ allows for proving (weak) total correctness: this role of triples will be clarified later. Some considerations about the above definition are in order.

Proving a triple for a given program or query involves reasoning on their ground instances only. Basically, the definition provides a standard way for *lifting up* the results to non-ground queries. The advantage is that this lifting is made *a posteriori*.

Intuitively speaking, a precondition characterizes the intended (or well-typed) queries. A ground instance of a clause whose head is not in the precondition is superfluous as we never use it in a ground derivation starting with an atom of the precondition. The level mapping plays the role of a termination function. Strictly speaking, the level mapping has to be defined only on the precondition.

NOTE 2.3 Given a program P such that $\vdash \{Pre\}\ P\ \{Post\}$ and $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$, if $A \in Pre$ then it is immediate to prove $\vdash \{Pre\}\ B_1, \ldots, B_n\ \{Post\}$. This points out how triples for programs and queries are related.

Proof Outlines

The proof of a triple $\vdash_t \{Pre\}\ P\ \{Post\}$ can be presented in a suggestive way using *proof outlines*. A proof outline $\mathcal{PO}$ for a clause $A \leftarrow A_1, \ldots, A_n$ and $|\ |$, *Pre*, *Post* is a construct of the form

$$
\begin{array}{llll}
\{g_0\} & & & \\
A_0 & \leftarrow & & \{t_0\} \\
 & \{g_1\} & & \\
 & A_1, & & \{t_1\} \\
 & \{f_1\} & & \\
 & \{g_2\} & & \\
 & \vdots & & \\
 & A_{n-1}, & & \{t_{n-1}\} \\
 & \{f_{n-1}\} & & \\
 & \{g_n\} & & \\
 & A_n. & & \{t_n\} \\
 & \{f_n\} & & \\
\{f_0\} & & &
\end{array}
$$

where t_i and f_i, g_i, for $i \in [0,n]$ are respectively integer expressions and (meta) assertions, such that for every ground instance $\mathcal{PO}'$ the following *proof obligations* are satisfied. Here, we denote by h_i for $i \in [1,n]$ the assertion $g_0 \wedge f_1 \wedge \ldots \wedge f_{i-1}$.

(i) for $i \in [0,n]$: $h'_i \Rightarrow t'_i = |A'_i|$,
(ii) $g'_0 \Leftarrow A'_0 \in Pre$, and for $i \in [1,n]$: $h'_i \wedge g'_i \Rightarrow A'_i \in Pre$,
(iii) for $i \in [1,n]$: $f'_i \Leftarrow A'_i \in Post \wedge h'_i$, and $h'_n \wedge f'_0 \Rightarrow A'_0 \in Post$,
(iv) for $i \in [1,n]$: $g'_0 \wedge f'_1 \wedge \ldots \wedge f'_{i-1} \Rightarrow g'_i \wedge t'_0 > t'_i$.
(v) $g'_0 \wedge f'_1 \wedge \ldots \wedge f'_n \Rightarrow f'_0$,

In this definition, the intuitive meaning of assertions f'_i and g'_i is that, for $i \in [1,n]$, f'_i holds iff $A'_i \in Pre$, and analogously g'_i holds iff $A'_i \in Post$. However, this constraints are weakened as shown above, in order to facilitate the construction of the proof outlines. The assertions h_i are intended to model the assumptions already considered before stage i of the proof, in order to avoid to repeatedly assume them.

By construction, we have that $\vdash_t \{Pre\}\ P\ \{Post\}$ if and only if there exists a level mapping $|\ |$ and a proof outline for each clause of P and $|\ |, Pre, Post$.

Consider, as an example, the following proof outline for the recursive clause of APPEND.

```
{ [X|Xs] and Ys lists }
append( [X|Xs], Ys, [X|Zs] ) ←                 { |Xs|+1 }
      { Xs and Ys lists }
    append( Xs, Ys, Zs ).                      { |Xs| }
      { Zs is the concatenation of Xs and Ys }
{ [X|Zs] is the concatenation of [X|Xs] and Ys }
```

where $|Xs|$ denotes the length of a list Xs.
It is immediate to verify, by a simple, natural and intuitive reasoning, that the proof obligations are satisfied.

The proof outline system become simpler when considering the relation $\vdash$. In the labelled clause no t_i appears and the proof-obligations are (ii-iii-v) and
(iv′) for $i \in [1,n]$: $g'_0 \wedge f'_1 \wedge \ldots \wedge f'_{i-1} \Rightarrow g'_i$.

Also in this case, by construction $\vdash \{Pre\}\ P\ \{Post\}$ holds if and only if there exists a proof outline for each clause of P and $Pre, Post$.

Vanilla

As a first exercise, let us prove that the relation $\vdash$ is closed under `Vanilla`'s instantiation, in the sense that if $\vdash \{Pre\}\ P\ \{Post\}$ holds, then $\vdash \{Pre_{Van}\}\ Van_P\ \{Post_{Van}\}$ holds for certain Pre_{Van}, $Post_{Van}$ defined starting from Pre and $Post$.

Let $B_1, \dots, B_n$ be ground atoms, with $n \geq 0$. We remember that B_1 & ... & B_n is an abbreviation for

- $(B_1$ & $(B_2$ & $\dots(B_{n-1}$ & $B_n)\dots))$, if $n > 1$,
- B_1, if $n = 1$
- and `true` , if $n = 0$

Conversely, by writing $(B_1$ & ... & $B_n)^-$ we denote the query $B_1, \dots, B_n$.
Let us define:

$$\begin{array}{rll} \texttt{prove}(B_1\ \texttt{\&}\ \dots\ \texttt{\&}\ B_n) \in Pre_{Van} & \text{iff} & \vdash \{Pre\}\ B_1, \dots, B_n\ \{Post\} \\ \texttt{clause}(A,\ B) \in Pre_{Van} & \text{iff} & true \end{array}$$

$$\begin{array}{rll} \texttt{prove}(B_1\ \texttt{\&}\ \dots\ \texttt{\&}\ B_n) \in Post_{Van} & \text{iff} & Post \models B_1, \dots, B_n \\ \texttt{clause}(A,\ B_1\ \texttt{\&}\ \dots\ \texttt{\&}\ B_n) \in Post_{Van} & \text{iff} & A \leftarrow B_1, \dots, B_n \in ground_L(P) \end{array}$$

No other atom is in Pre_{Van} or in $Post_{Van}$.
The next proof outlines establish that $\vdash \{Pre_{Van}\}\ Van_P\ \{Post_{Van}\}$.

```
        { $true$ }
(a)     prove(true).
        { $true$ }

        { $\vdash \{Pre\}\ A, B^-\ \{Post\}$ }
(b)     prove(A & B ) ←
                { $Pre \models A$ }
            prove(A),
                { $Post \models A$ }
                { $\vdash \{Pre\}\ B^-\ \{Post\}$ }
            prove(B).
                { $Post \models B^-$ }
        { $Post \models A, B^-$ }

        { $\texttt{prove}(A) \in Pre_{Van}$ }
(c)     prove(A) ←
                { $true$ }
            clause(A, B),
                { $A \leftarrow B^- \in ground_L(P)$ ) }
                { $\vdash \{Pre\}\ B^-\ \{Post\}$ }
            prove(B).
                { $Post \models B^-$ }
        { $Post \models A$ }

        { $true$ }
(d)     clause(A, B).
        { $A \leftarrow B^- \in ground_L(P)$ }
```

Proof outlines *(a,b)* are self-explanatory, by simply observing that for a ground query $B_1, \dots, B_n$:

(i) $\vdash \{Pre\}\ B_1, \dots, B_n\ \{Post\} \quad \Rightarrow \quad Pre \models B_1$
(ii) $\vdash \{Pre\}\ B_1, \dots, B_n\ \{Post\} \wedge Post \models B_1 \quad \Rightarrow \quad \vdash \{Pre\}\ B_2, \dots, B_n\ \{Post\}$

The proof of *(i,ii)* is immediate by the Definition 2.2. Proof outline *(d)* is actually a *schemata* of proof outlines, one for each clause from P, and it is of immediate verification.

Consider now the proof outline *(c)*. First, we note that:

$$\texttt{prove}(A) \in Pre_{Van} \wedge A \leftarrow B^- \in ground_L(P) \quad \Rightarrow \quad Pre \models A \tag{1}$$

In fact, if A is the head of a ground clause from P then its predicate symbol cannot be `true` or `&` by the assumptions about the language L. Then $\texttt{prove}(A) \in Pre_{Van}$ implies $\vdash \{Pre\}\ A\ \{Post\}$, i.e. $A \in Pre$.

Let us prove the proof obligations (iv) case $i = n$ and (v).

In the former case, assume $\texttt{prove}(A) \in Pre_{Van} \wedge A \leftarrow B^- \in ground_L(P)$. By (1), we have $Pre \models A$ and, by Note 2.3, we conclude $\vdash \{Pre\}\ B^-\ \{Post\}$.

In the latter case, assume $\texttt{prove}(A) \in Pre_{Van} \wedge A \leftarrow B^- \in ground_L(P) \wedge Post \models B^-$. By (1) and the last two conjuncts we conclude directly by Definition 2.2 that $Pre \models A$.

3 Weak Partial Correctness

We start by stating a persistency properties for $\vdash$, and by giving a justification of the intuitive notion that the postcondition is a description of the correct instances of queries satisfying the triple $\vdash \{Pre\}\ Q\ \{Post\}$.

LEMMA 3.1 *Let P be a program and Q a query such that $\vdash \{Pre\}\ P\ \{Post\}$ and $\vdash \{Pre\}\ Q\ \{Post\}$. Then*
(i) for every SLD-resolvent Q' of Q and P $\vdash \{Pre\}\ Q'\ \{Post\}$ holds, and
(ii) for every computed (or correct) instance Q' of Q and P $Post \models Q'$ holds. □

As a consequence, we obtain the weak partial correctness Theorem.

THEOREM 3.2 (WEAK PARTIAL CORRECTNESS) *A program P such that $\vdash \{Pre\}\ P\ \{Post\}$ is weak partially correct w.r.t. the specification $(Pre, Post)$.*

Proof. Consider $A \in M_P \cap Pre$. Then $\vdash \{Pre\}\ A\ \{Post\}$ holds, and, by Completeness of SLD-resolution, the query A has an SLD-refutation. So $A \in Post$ by Corollary 3.1 *(ii)*. □

Vanilla

As we proved $\vdash \{Pre_{Van}\}\ Van_P\ \{Post_{Van}\}$ we can use Theorem 3.2 to conclude that Van_P is weak partially correct w.r.t. the specification $(Pre_{Van}, Post_{Van})$ when $\vdash \{Pre\}\ P\ \{Post\}$.

Moreover, considering a (not necessarily ground) query $B_1, \dots, B_n$ such that

$$\vdash \{Pre\}\ B_1, \dots, B_n\ \{Post\}$$

it turns out directly from the definition of Pre_{Van} that

$$\vdash \{Pre_{Van}\}\ \texttt{prove}(B_1\ \texttt{\&}\ \dots\ \texttt{\&}\ B_n)\ \{Post_{Van}\}$$

4 Partial Correctness

The intuition underlying a *Hoare style* proof method based on *Pre/Post*-conditions, become more concrete when dealing with modular proofs. The modularity theorem may be explained by the following inference rule

$$\frac{\vdash \{Pre\}\ P\ \{Post\} \quad \vdash \{Pre\}\ P\ \{Post'\}}{\vdash \{Pre\}\ P\ \{Post \cap Post'\}}.$$

The importance of this fact is twofold.

On the one hand, it has a relevance from an applicative point of view. It allows for splitting a correctness proof into simpler ones.

On the other hand, it allows us to define a notion of *strongest postcondition.*

DEFINITION 4.1 Let P be a program such that $\vdash \{Pre\}\ P\ \{Post\}$. By $sp(P, Pre)$ we denote the intersection of all $Post'$ such that $\vdash \{Pre\}\ P\ \{Post'\}$. □

It is natural to ask ourselves whether the interpretation $M_P \cap Pre$ is a postcondition. Actually, it turns out that it is strongest one.

THEOREM 4.2 *For a program P such that $\vdash \{Pre\}\ P\ \{Post\}$ we have*

$$\vdash \{Pre\}\ P\ \{M_P \cap Pre\}.$$

Hence, $sp(P, Pre) = M_P \cap Pre$. □

As a result we have the Partial Correctness Theorem.

THEOREM 4.3 (PARTIAL CORRECTNESS) *A program P such that $\vdash \{Pre\}\ P\ \{Post\}$ is partially correct w.r.t. the specification $(Pre, sp(P, Pre))$.* □

The problem is now to characterize the strongest postcondition without having to construct the complete minimal model. Proving weak partial correctness is simple, as one have only to show some proof outlines. Next definition introduces a notion that allows us for proving partial correctness.

DEFINITION 4.4 Let P be a program such that $\vdash \{Pre\}\ P\ \{Post\}$. *Post* is a *well-supported interpretation* (w.r.t. P and *Pre*) iff there exists a well-founded poset $(W, <)$ and a function $|\ | : B_L \rightarrow W$ such that for any $A \in Post \cap Pre$ there exists $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ such that $\forall i \in [1, n] : Post \models B_i \wedge |A| > |B_i|$ □

The underlying idea of this definition is to require that any atom in $Post \cap Pre$ has a successful ground derivation. In fact, for each of them there exists a ground finite (as the poset is well-founded) derivation which is successful because the last selected atom unifies with at least one head.

THEOREM 4.5 *Let P be a program such that $\vdash \{Pre\}\ P\ \{Post\}$. Then*

$$Post \cap Pre = sp(P, Pre) \quad \textit{iff} \quad Post \ \textit{ is well-supported (w.r.t. } P \textit{ and } Pre\textit{).}$$

□

Proof Outlines

Definition 4.4 may seem a little complicated. However, it has a straightforward interpretation in terms of proof outlines. Consider the following proof outline $\mathcal{PO}$ for a clause $A \leftarrow A_1, \ldots, A_n$, a function $|\ | : B_L \rightarrow W$ into a well-founded poset $(W, <)$ and *Pre*, *Post*

$$\begin{array}{lllr}
 & \{g\} & & \\
A & & \leftarrow & \{t_0\} \\
 & & A_1, & \{t_1\} \\
 & & \quad \{f_1\} & \\
 & & \vdots & \\
 & & A_{n-1}, & \{t_{n-1}\} \\
 & & \quad \{f_{n-1}\} & \\
 & & A_n. & \{t_n\} \\
 & & \quad \{f_n\} &
\end{array}$$

where t_i and f_i, g, for $i \in [0, n]$ are respectively integer expression and (meta–)assertions, such that for every ground instance $\mathcal{PO}'$ the following *proof obligations* hold:

(i) for $i \in [0, n]$: $g' \Rightarrow t'_i = |A'_i|$,
(ii) for $i \in [1, n]$: $g' \wedge f'_i \Rightarrow A'_i \in Post$,
(iii) for $i \in [1, n]$: $g' \Rightarrow f'_i \wedge t'_0 > t'_i$.

The formula g is used to instantiate the variables of the clause to the end of satisfying the proof obligations.

By construction, *Post* is well supported w.r.t. P and *Pre* if and only if there exist a number of proof outlines for (instances of) clauses from P, and a function $|\ | : B_L \rightarrow W$ such that every atom in $Pre \cap Post$ is an instance of a clause's head and satisfies $\exists \tilde{x}.g$, where $\tilde{x}$ are the local variables of the clause, and g is the assertion of the head of the clause.

Vanilla

In the last section we have proved $\vdash \{Pre_{Van}\}\ Van_P\ \{Post_{Van}\}$ when $\vdash \{Pre\}\ P\ \{Post\}$. Following that reasoning, it would be now interesting to prove that $Post_{Van}$ is a well-supported interpretation (w.r.t. `Vanilla` and Pre_{Van}) when the postcondition of the instantiating program is well-supported (w.r.t. the program and its precondition).

Suppose *Post* is well-supported. Then there exist $(W, <)$ and a function $|\ | : B_L \rightarrow W$ such that for any $A \in Post \cap Pre$ there exists $A \leftarrow B_1, \ldots, B_n \in ground_L(P)$ such that

$$\forall i \in [1, n] : Post \models B_i \wedge |A| > |B_i| \tag{2}$$

We consider the well-founded ordering $(bag(W), \prec)$ over finite multiset of W induced by $(W, <)$, and the level mapping

$$||\texttt{prove(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{)}|| = bag(|B_1|, \ldots, |B_n|)$$

for $\texttt{prove(}B_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ B_n\texttt{)} \in Pre_{Van}, n \geq 1$ and $bag()$ otherwise.

By straightforward arguments, we note that for $\texttt{prove(}A\ \texttt{\&}\ B\texttt{)} \in Pre_{Van}$

$$||\texttt{prove(}A\texttt{)}|| \prec ||\texttt{prove(}A\ \texttt{\&}\ B\texttt{)}|| \wedge ||\texttt{prove(}B\texttt{)}|| \prec ||\texttt{prove(}A\ \texttt{\&}\ B\texttt{)}|| \tag{3}$$

Next proof outlines show that $Post_{Van}$ is a well-supported interpretation.

```
        { true }
(a)     prove(true).                              { bag() }
```

$$\begin{array}{lll}
 & \{\ \texttt{prove}(A\ \&\ B) \in Pre_{Van}\ \wedge\ Post \models A, B^-\ \} & \\
(b) & \texttt{prove(A \& B)}\ \leftarrow & \{\ \|\texttt{prove}(A\ \&\ B))\|\ \} \\
 & \quad \texttt{prove(A),} & \{\ \|\texttt{prove}(A)\|\ \} \\
 & \qquad \{\ Post \models A\ \} & \\
 & \quad \texttt{prove(B).} & \{\ \|\texttt{prove}(B)\|\ \} \\
 & \qquad \{\ Post \models B^-\ \} & \\
 & & \\
 & \{\ A \in Post \cap Pre\ \wedge\ Post \models B^-\ \wedge\ A \leftarrow B^- \in ground_L(P)\ \wedge & \\
 & \quad \wedge\ \|\texttt{prove}(B)\|\ \prec \|\texttt{prove}(A)\|\ \} & \\
(c) & \texttt{prove(A)}\ \leftarrow & \{\ \|\texttt{prove}(A)\|\ \} \\
 & \quad \texttt{clause(A , B),} & \{\ bag()\ \} \\
 & \qquad \{\ A \leftarrow B^- \in ground_L(P)\ \} & \\
 & \quad \texttt{prove(B).} & \{\ \|\texttt{prove}(B)\|\ \} \\
 & \qquad \{\ Post \models B^-\ \} & \\
 & & \\
 & \{\ A \leftarrow B^- \in ground_L(P)\ \} & \\
(d) & \texttt{clause(A, B).} & \{\ bag()\ \}
\end{array}$$

The proof outlines are of immediate verification by using the definition of $Pre_{Van}, Post_{Van}$.

To conclude $Post_{Van}$ is a well-supported interpretation, we have to show that every $A \in Post_{Van} \cap Pre_{Van}$ satisfies some $\exists\tilde{x}.g$, where g is a formula of the head and $\tilde{x}$ are the local variables of the clause.

For the atoms whose predicate symbol is `clause` the conclusion is trivial.

Suppose the hypothesis $\texttt{prove}(B_1\ \&\ \dots\ \&\ B_n) \in Pre_{Van} \cap Post_{Van}$.

If $n \neq 1$ we refer to the proof outline *(a)* or *(b)* to note as the head's formula g is just the hypothesis.

If $n = 1$ then let us show that $\exists\tilde{x}.g$ holds by considering the proof outline *(c)*. We have to prove that for $A \in Post \cap Pre$ there exists B^- such that $Post \models B^- \wedge A \leftarrow B^- \in ground_L(P)$ and $\|\texttt{prove}(B)\| \prec \|\texttt{prove}(A)\|$. Such a B^- surely exists as (2) holds.

5 (Weak) Total Correctness

The relation $\vdash$ is not powerful enough for dealing with (weak) total correctness, as termination is not taken into account. To pursue this end, we will reason about the relation $\vdash_t$. The results of termination are not reported for lack of space. They state a form of *universal termination* w.r.t. *LDfair* resolution. An SLD-derivation is called *LDfair* if it is finite or the leftmost atom is selected infinitely many often. For instance, the Prolog selection rule is *LDfair*. One can prove that when $\vdash_t \{Pre\}\ P\ \{Post\}$ and $\vdash_t \{Pre\}\ Q\ \{Post\}$ then any *LDfair*-tree for $P \cup \{Q\}$ is finite.

As an immediate corollary of this termination property, we have the Total Correctness Theorems.

THEOREM 5.1 ((WEAK) TOTAL CORRECTNESS) *A program P such that $\vdash_t \{Pre\}\ P\ \{Post\}$ is weak totally correct w.r.t. the specification $(Pre, Post)$, and totally correct w.r.t. the specification $(Pre, sp(P, Pre))$.* □

The next step is finding a proof method for total correctness. A straightforward consequence of the result of section 4 allows us to obtain what we desire.

THEOREM 5.2 *Let P be a program such that $\vdash_t \{Pre\}\ P\ \{Post\}$. Then $Post \cap Pre = sp(P, Pre)$ iff $T_P(Post) \supseteq Post \cap Pre$.* □

Proof Outline and Vanilla

The proof outline system for total correctness is obtained starting from that for partial correctness by simply not considering the *termination* constraints, i.e. $t_0, \dots, t_n$.

In the same way as we proceeded for $\vdash$, it involves no real difficulty to prove

$$\vdash_t \{Pre_{Van}\}\ Van_P\ \{Post_{Van}\}$$

for some $|\ |_{Van}$ when $\vdash_t \{Pre\}\ P\ \{Post\}$ holds. Analogously to the case of relation $\vdash$, we note as for a (not necessarily ground) query $B_1, \dots, B_n$

$$\vdash_t \{Pre_{Van}\}\ \texttt{prove}(B_1\ \&\ \dots\ \&\ B_n)\ \{Post_{Van}\}$$

when $\vdash_t \{Pre\}\ B_1, \dots, B_n\ \{Post\}$. Therefore, we can state that `Vanilla` is *closed w.r.t. the proof theory notions* that we have introduced in the paper.

6 Correctness of Vanilla

The underlying idea we will use to prove correctness of `Vanilla` is the following:

(i) Consider a program P such that $\vdash \{Pre\}\ P\ \{Post\}$. By Theorem 4.2 we have

$$\vdash \{Pre\}\ P\ \{sp(P, Pre)\}$$

(ii) By defining $Pre_{Van}, Post_{Van}$ starting from $Pre, sp(P, Pre)$ we have showed in Section 2 that $\vdash \{Pre_{Van}\}\ Van_P\ \{Post_{Van}\}$ holds. Moreover, as $sp(P, Pre)$ is well-supported, also $Post_{Van}$ is. We proved this in section 4.

(iii) Since $Post_{Van}$ is well-supported, by Theorem 4.5 we obtain

$$sp(Van, Pre_{Van}) = Post_{Van} \cap Pre_{Van} = M_{Van} \cap Pre_{Van} \tag{4}$$

(iv) As a result, for every $A \in Pre$ we have

$$\begin{array}{ll}
 & \texttt{prove}(A) \in M_{Van} \\
\Leftrightarrow & \quad \{\ (4) \wedge \texttt{prove}(A) \in Pre_{Van}\ \} \\
 & \texttt{prove}(A) \in Post_{Van} \\
\Leftrightarrow & \quad \{\ \text{Definition of } Post_{Van}\ \} \\
 & A \in sp(P, Pre) \\
\Leftrightarrow & \quad \{\ \text{Theorem 4.2}\ \} \\
 & A \in M_P
\end{array}$$

i.e., the correctness of the `Vanilla` metainterpreter w.r.t. *Pre*.

Moreover, we can prove the slightly stronger fact.

COROLLARY 6.1 *For a program P and a ground query Q such that $\vdash \{Pre\}\ P\ \{Post\}$ and $\vdash \{Pre\}\ Q\ \{Post\}$, we have*

$$P \models Q \quad \textit{iff} \quad Van_P \models \texttt{prove}(Q)$$ □

As for every logic program we have $\vdash \{B_L\}\ P\ \{B_L\}$ the last result implies that for every program P, a ground query Q is a logical consequence of P iff $\texttt{prove}(Q)$ is a logical consequence of `Vanilla` instantiated by P.

We conclude by spending some words about the case in which one desires to distinguish the underlying language of a program from that of `Vanilla`. Let M_P^L be the least Herbrand model of P with L (extending L_P) the considered underlying language. Observing that $M_P^L \cap B_P = M_P^{L_P}$, directly from (iv) we obtain the correctness theorem generally studied in the literature.

THEOREM 6.2 *For a program P and $A \in B_P$ we have*

$$A \in M_P^{L_P} \quad \textit{iff} \quad \texttt{prove}(A) \in M_{Van}^{L} \qquad \square$$

7 Conclusions

As a consequence of our exercise, we learned that the verification of the `Vanilla` meta-interpreter can be carried out in a simple and natural way within the proof theory sketched in this paper. Moreover, several improvements can be achieved with little effort.

For instance, by using a generalization of the technique of *extended level mappings* from [PR94], it is possible to drop the *groundness* requirement in Corollary 6.1. Moreover, by generalizing the results of [AGP94], we can identify a large class of programs for which it is possible to fully reconstruct the operational semantics (i.e., the computed instances of queries) from the well-typed fragment of M_P. Again, this class of programs is closed under the instantiation of `Vanilla`.

References

[AGP94] K.R. Apt, M. Gabbrielli, and D. Pedreschi. A closer look at declarative interpretations. Technical Report CS-R9470, Centre for Mathematics and Computer Science, Amsterdam, 1994.

[Apt90] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 493–574. Elsevier, 1990.

[BK82] K.A. Bowen and R.A. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K.L. Clark and S.A. Tarnlund, editors, *Logic Programming*, pages 153–173. Academic Press, 1982.

[BT95] A. Brogi and F. Turini. Meta-logic for program composition: semantic issues. In K.R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*. The MIT Press, 1995.

[Cla79] K.L. Clark. Predicate logic as a computational formalism. Technical Report DOC 79/59, Imperial College, Dept. of Computing, 1979.

[Jia94] Y. Jiang. Ambivalent logic as the semantic basis of metalogic programming:I. In P. van Henterynck, editor, *Proceedings of ICLP '94*, pages 387–401. The MIT Press, 1994.

[Kal95] M. Kalsbeck. Correctness of the vanilla meta-interpreter and ambivalent syntax. In K.R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 3–26. The MIT Press, 1995.

[KJ95] M. Kalsbeck and Y. Jiang. A vademecum of ambivalent logic. In K.R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 27–56. The MIT Press, 1995.

[Llo87] J.W. Lloyd. *Foundations of logic programming.* Springer-Verlag, Berlin, second edition, 1987.

[PR94] D. Pedreschi and S. Ruggieri. Termination is language-independent. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proceedings of the 1994 Joint Conference, GULP-PRODE'94.* Universidad Politecnica de Valencia, 1994.

[PR95] D. Pedreschi and S. Ruggieri. Verification of prolog programs. Technical Report, 1995.

[Rug94] S. Ruggieri. Metodi formali per lo sviluppo di programmi logici. Master's thesis, Dipartimento di Informatica, Università di Pisa, 1994.

[SS86] E. Shapiro and L. Sterling. *The Art of Prolog.* The MIT press, 1986.