Abstract Interpretation based Verification of Logic Programs

Marco Comini, Roberta Gori, Giorgio Levi, Paolo Volpe

Dipartimento di Informatica
Università di Pisa
Pisa, Italy

Abstract

This paper is an overview of our results on the application of abstract interpretation concepts to various problems related to the verification of logic programs. These include the systematic design of semantics modeling various proof methods and the characterization of assertions as abstract domains.

1 Abstract Interpretation and Verification

Abstract interpretation [14,15] is a general theory for approximating the semantics of discrete dynamic systems, originally developed by Patrick and Radhia Cousot, in the late 70's, as a unifying framework for specifying and validating static program analyses. The *abstract semantics* is an approximation of the concrete one, where exact (concrete) properties are replaced by approximated properties, modeled by an abstract domain. The framework of abstract interpretation can be useful to study hierarchies of semantics and to reconstruct data-flow analysis methods and type systems. It can be used to prove the safety of an analysis algorithm. However, it can also be used to systematically derive "optimal" abstract semantics from the abstract domain. The systematic design aspect can be pushed forward, by using suitable abstract domain design methodologies (e.g. domain refinements) [23,25,27], which allow us to systematically improve the precision of the domain.

Abstract interpretation was originally intended as a method for automatically generating program invariants. Even more recently [16,17,13], it was shown to be very useful to understand, organize and synthesize proof methods for *program verification*. In particular, we are interested in one specific approach to the generation of abstract interpretation-based partial correctness conditions, which is used also in abstract debugging [4,10,6].

The ideas behind this approach are the following:

- The concrete semantics $[\![P]\!]$ of a program P is defined as the least fixpoint of a semantic evaluation function $\mathcal{T}_P$ on the concrete domain $(\mathbb{D}, \leq)$.
- As in standard abstract interpretation based program analysis, the class of properties we want to verify is formalized as an abstract domain $(\mathbb{A}, \sqsubseteq)$, related to $(\mathbb{D}, \leq)$ by the usual Galois connection $\alpha : \mathbb{D} \to \mathbb{A}$ and $\gamma : \mathbb{A} \to \mathbb{D}$ (abstraction and concretization functions). The corresponding *abstract semantic evaluation function* $\mathcal{T}_P^\alpha$ is systematically derived from $\mathcal{T}_P$, α and γ. The resulting abstract semantics is a correct approximation of the concrete semantics by construction and no additional "correctness" theorems need to be proved.
- An element $\mathcal{S}$ of the domain $(\mathbb{A}, \sqsubseteq)$ is simply an abstract specification of the intended concrete semantics.
- The partial correctness of a program P w.r.t. a specification $\mathcal{S}$ can be expressed as $[\![P]\!] \leq \gamma(\mathcal{S})$ or, equivalently, as

$$\alpha([\![P]\!]) \sqsubseteq \mathcal{S}. \tag{1}$$

- Since $[\![P]\!]$ is defined as the least fixpoint of the operator $\mathcal{T}_P$, a sufficient condition for (1) to hold is $\mathcal{T}_P(\gamma(\mathcal{S})) \leq \gamma(\mathcal{S})$ or, equivalently,

$$\mathcal{T}_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}. \tag{2}$$

  In fact $\mathcal{T}_P^\alpha(\mathcal{S}) \sqsubseteq \mathcal{S}$ implies $\mathit{lfp}(\mathcal{T}_P^\alpha) \sqsubseteq \mathcal{S}$ and, since $\alpha(\mathit{lfp}(\mathcal{T}_P)) \sqsubseteq \mathit{lfp}(\mathcal{T}_P^\alpha)$, the condition $\alpha([\![P]\!]) \sqsubseteq \mathcal{S}$ can be derived. Note that (2) means that the specification $\mathcal{S}$ is a *pre-fixpoint* of the abstract semantic evaluation function $\mathcal{T}_P^\alpha$.

Following the above approach, verification techniques inherit the nice features of abstract interpretation. Namely, we can define a verification framework, parametric with respect to the (abstract) property we want to model. Given a specific property, the corresponding verification conditions are systematically derived from the framework and guaranteed to be indeed sufficient partial correctness conditions. As we will discuss in Section 2, the verification method is guaranteed to be complete, if the abstraction is precise (complete according to abstract interpretation theory).

The inductive verification method based on the sufficient condition (2) does not require to compute fixpoints. In order to make it effectively applicable, we need

- a concrete fixpoint (denotational) semantics, which allows us to observe the property we want to verify.
- a finite representation of the intended abstract behavior (specification).

Abstract interpretation theory provides results and techniques which can be used to tackle all the above problems. Namely, it can be used to

- systematically design a semantics, which models suitable observable behaviors *and* which exhibits suitable properties related to precision and compositionality. This problem will be considered in Section 3 for behaviors which are abstractions of successful derivations and in Section 6 for behaviors related to infinite derivations.
- reconstruct the existing notions of partial correctness and related verification methods (for logic programs) simply in terms of different choices of the concrete semantics (see Section 4). Here we can use another important feature of abstract interpretation, namely its ability to compare different semantics by reasoning in terms of abstraction.
- choose the appropriate abstract domain to model the property, so as to allow specifications to be always finite, in order to make effective the verification method. The easy solution is to choose the usual abstract domains developed for static program analysis (types, groundness, etc.). A more interesting solution is to consider assertions in a suitable specification language, as we will do in Section 5.

There exist other approaches to verification of logic programs, using abstract interpretation techniques. For example, [12,31] define a verification method for Prolog, which applies to specifications related to properties such as termination, and size-cardinality relations between inputs and outputs.

2 The issue of completeness

In general, given an inductive proof method, if a program is correct with respect to a specification $\mathcal{S}$, the verification condition might not hold for $\mathcal{S}$. However, if the method is *complete*, then when our program is correct with respect to specification $\mathcal{S}$, there exists a property $\mathcal{R}$, stronger than $\mathcal{S}$, which verifies the verification condition. We have proved [34,41] that, for verification conditions which have the form of condition (2) for a suitable α, the derived method is complete if and only if the abstraction is *precise with respect to* $\mathcal{T}_P$, that is if $\alpha(\mathit{lfp}(\mathcal{T}_P)) = \mathit{lfp}(\mathcal{T}_P^\alpha)$.

In fact, it is known from simple lattice theoretic facts (Park's fixpoint induction) that for a monotonic operator F on a complete lattice

$$\mathit{lfp}(F) \leq \varphi \Longleftrightarrow \exists \psi \leq \varphi : F(\psi) \leq \psi. \tag{3}$$

We can easily derive the following lemma.

Lemma 2.1 *Let (C, A, α, γ) be a Galois connection between the complete lattices C and A. Assume $F : C \to C$ is a monotonic operator on C and $F^\alpha = \alpha \circ F \circ \gamma : A \to A$ satisfies the condition $\alpha \circ F \sqsubseteq F^\alpha \circ \alpha$. Then, for each $\varphi \in A$, $\alpha(\mathit{lfp}(F)) \sqsubseteq \varphi \Longrightarrow \exists \psi \sqsubseteq \varphi : F^\alpha(\psi) \sqsubseteq \psi$ if and only if (C, A, α, γ) is precise with respect to F.*

Then, if the proof method is derived from condition (2) and the abstraction is precise, if a program P is correct with respect to the property $\mathcal{S}$ (that is $\alpha(\llbracket P \rrbracket) \sqsubseteq \mathcal{S}$) then there exists a property $\mathcal{R}$ stronger than $\mathcal{S}$ (that is $\mathcal{R} \sqsubseteq \mathcal{S}$), which verifies the verification condition of the method (that is $\mathcal{T}_P^\alpha(\mathcal{R}) \sqsubseteq \mathcal{R}$). Note that precision of abstract interpretation can be quite difficult to prove. A sufficient condition for precision, generally easier to check, is *full precision*, that is $\alpha \circ \mathcal{T}_P = \mathcal{T}_P^\alpha \circ \alpha$. [26] contains some methods which allow us to systematically enrich a domain of properties so as to obtain an abstraction which is fully precise with respect to a given function. These methods can be viewed as the basis for the systematic development of complete proof methods.

3 Designing semantics by abstract interpretation

The aim of the approach to semantics, known as *s-semantics* [3,24], was the definition of denotations modeling various operational properties (observables) in a compositional way, to be used as the semantic basis of analysis, verification and transformation methods. All the semantics proposed by this approach have been reconstructed (and systematically derived) as instances of a framework based on abstract interpretation [9,8]. The framework is based on a concrete semantics [11], which models *SLD*-trees and is formalized both denotationally and operationally. Let us introduce some notation.

- $\mathcal{B}\llbracket \boldsymbol{G} \; in \; P \rrbracket$ is the operational semantics of goal $\boldsymbol{G}$ in program P (roughly speaking, the set of its *SLD*-derivations).
- $\mathcal{T}_P$ is the (denotational) semantic evaluation function for a program P (set of definite clauses).
- $\llbracket P \rrbracket = \mathit{lfp}(\mathcal{T}_P)$ is the denotational semantics of P.
- $\mathcal{Q}\llbracket \boldsymbol{G} \; in \; P \rrbracket$ is the denotational derivation of the semantics of the goal $\boldsymbol{G}$ from $\llbracket P \rrbracket$.

The main properties of the concrete semantics are the equivalence between the operational semantics and the denotational semantics, i.e. $\forall \boldsymbol{G}, P. \mathcal{B}\llbracket \boldsymbol{G} \, in \, P \rrbracket = \mathcal{Q}\llbracket \boldsymbol{G} \; in \; P \rrbracket$ and the existence of a goal-independent denotation $\llbracket P \rrbracket$, which is *AND*-compositional, i.e. which allows us to precisely derive (by means of the function $\mathcal{Q}$) the behavior of every (conjunctive) goal.

An *observable* is a Galois insertion between the domain of *SLD*-trees and an abstract domain describing the property to be modeled. The abstract denotational semantic functions $\mathcal{T}_P^\alpha$ and $\mathcal{Q}^\alpha$, together with the abstract denotational semantics $\llbracket P \rrbracket^\alpha = \mathit{lfp}(\mathcal{T}_P^\alpha)$, are systematically derived from the concrete ones, by replacing the concrete semantic operators by their optimal abstract versions.

The next step is the definition of a taxonomy of classes of observables. An observable belongs to a class if it satisfies a set of conditions relating the

concrete semantic operators and the Galois insertion. Once we have shown that an observable belongs to a given class, we know how to automatically derive the "best" semantics and which are the properties of such a semantics. The properties we consider include precision, relation between concrete operational semantics and abstract denotational semantics, existence of a goal-independent denotation for a set of definite clauses and compositionality w.r.t. various syntactic operators.

The two classes of observables which are relevant to the present discussion are the following.

Denotational observables, for which we can obtain the optimal abstract semantics in a denotational way, by taking the optimal abstract version $\mathcal{T}_P^\alpha$ of $\mathcal{T}_P$. Denotational observables have a precise abstract denotational semantics ($[\![P]\!]^\alpha = \alpha([\![P]\!])$) and the abstract (goal-independent) denotation is *AND*-compositional. The class includes correct answers substitutions, computed answer substitutions, call patterns and resultants.

Semi-denotational observables, which are intended to model the properties useful for static program analysis, where we give up precision to achieve termination in the construction of the abstract semantics. The semantic construction of semi-denotational observables is the same of denotational observables. We just lose the precision of the abstract denotational semantics, i.e. $\alpha([\![P]\!]) \sqsubseteq [\![P]\!]^\alpha$. Semi-denotational observables have also an abstract operational semantics $\mathcal{B}^\alpha$. The (abstract) semantics for any goal $\boldsymbol{G}$ computed denotationally is as precise as the operational one, i.e. $\forall \boldsymbol{G}, P. \mathcal{B}^\alpha[\![\boldsymbol{G}\ in\ P]\!] = \mathcal{Q}^\alpha[\![\boldsymbol{G}\ in\ P]\!]$. The class includes the domain $depth(k)$, the domain $\mathcal{POS}$ for groundness analysis, proved to be optimal [38], by using the theory of refinement operators, and other optimal domains (such as the type domain in [33], designed by using the same operators).

It is worth stressing the importance of the *AND*-compositionality property, which guarantees that we can be as precise as possible (even in the case of approximate semi-denotational observables), when using denotational definitions. Note also that denotational observables are precise and will therefore lead to complete verification methods.

4 Verification methods

As already mentioned, the sufficient condition (2) (in the case of logic programs) was initially used in *abstract diagnosis* [10,8], a technique which extends declarative debugging [39,21] to a debugging framework, which is parametric w.r.t. the abstraction. Abstract diagnosis considers properties which are abstractions of computed answers. The corresponding specifications can then be viewed as postconditions. A similar approach is taken in [6], where

different approximations (modeled by abstract interpretation) can be used in the semantics and in the specification.

Example 4.1 *Let us consider the following program which is a wrong*[1] *version of an automaton which recognizes the language* $L = \{(ab)^n \mid n \geq 0\} \cup \{(ab)^n a \mid n \geq 0\}$.

```
c1:  accept([a|Xs]) :- acc(Xs).
c2:  acc([a|Xs]) :- accept(Xs).
c3:  accept([]).
```

and the following specification w.r.t. the depth(2) *answer observable (denoted by* τ_2*).*

$$\mathcal{I}^{\tau_2} := \begin{cases} accept(X) \mapsto \{\{X = [\,]\}, \{X = [a]\}, \{X = [a, \hat{V}|\hat{W}]\}\} \\ acc(X) \quad\;\; \mapsto \{\{X = [\,]\}, \{X = [b]\}, \{X = [b, \hat{V}|\hat{W}]\}\}. \end{cases}$$

Intuitively an abstraction over depth(k) *is obtained by cutting terms which have a depth greater than* k*, i.e. by replacing each sub-term rooted at depth* k *with a new variable taken from a set* $\hat{V}$ *(disjoint from* V*, the set of variables of the first order language* $\mathcal{L}$ *considered). A depth*(k) *term represents all the concrete terms obtained by instantiating the variables of* $\hat{V}$ *with terms built over* V*.*

By applying the $\mathcal{T}_P^{\tau_2}$ *operator we find out that the clause* `c2` *is probably wrong (as it is actually the case) and that the element* $acc(x) \mapsto \{x = [\,]\}$ *is uncovered.*

More general specifications (including pre and post conditions) are considered in [34,41], which defines a verification framework, where well known verification methods can be reconstructed, by simply choosing different observables (semantics). It is worth noting that the existing verification methods for logic programs have been defined by using ad-hoc constructions. Their reconstruction in terms of abstract interpretation allows us to compare the different techniques and to show the essential differences.

The approach can be explained in terms of two steps of abstraction, both modeled by observables. The first step is concerned with the derivation of the semantics which models the proof method. The second step performs the abstraction needed to model a specific class of properties (so as to lead to a finite specification). The methods which are considered are

Success-correctness. Here we consider post-conditions only. The right (denotational) observable is *correct answers*. The verification condition, obtained by unfolding (2) with the correct answers $\mathcal{T}_P^{\alpha}$, is essentially the same as those defined by Clark [7] and by Deransart [18]. The method is complete,

[1] The clause `c2` has 'a' instead of 'b' and the clause `accept([]).` is missing.

since the observable is precise.

I/O correctness. Specifications are pairs of pre- and post-conditions. We prove that the post-condition holds whenever the pre-condition is satisfied. The right (denotational) observable is *computed answers* (s-semantics). The method is complete. It boils down to the previous method, whenever the properties, considered in the second abstraction step, are closed under instantiation (as, for example, in [19]).

I/O and call correctness. Specifications are still pairs of pre- and post-conditions. However, we prove also that the pre-conditions are satisfied by all the procedure calls. The right (denotational) observable is *call patterns*. The method is complete. The verification condition, obtained by unfolding (2) with the call patterns $\mathcal{T}_P^\alpha$, is a slight generalization of the one defined by the Drabent-Maluszynski method [20]. If we consider, in the second abstraction step, properties closed under instantiation, we reconstruct the Bossi-Cocco condition [2], and, by further abstractions (modes, types, etc.), the hierarchy of verification conditions in [1].

As already mentioned, the second abstraction step is concerned with the choice of an abstract domain to model the property. Here we can make available to program verification all the abstract domains designed for the static analysis of properties such as modes, types, groundness dependencies, etc. The reasoning on the domain of properties is performed by efficient abstract computation steps and the sufficient condition can simply be proved by using the operations on the abstract cpo. As is the case for static analysis, in general we loose the precision (and therefore the completeness of the verification method). However we succeed in getting finite specifications. Note that, if we model the property by semi-denotational observables, even if the method is not complete, the abstract $\mathcal{T}_P^\alpha$, used in condition (2), introduces the same amount of approximation of the "best" goal-dependent abstract operational semantics.

Another possibility is to specify properties as assertions in a suitable specification language. This issue will be considered in the next section.

Example 4.2 *Let us consider the append program*

```
c1:  append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
c2:  append([], Xs, Xs).
```

and the corresponding intended specification w.r.t. the depth(2) *answer observable*

$$\begin{aligned}\mathcal{I}^{T2}_{Post} := append(X,Y,Z) \mapsto \{&\{X=[\,],Y=[\,],Z=[\,]\},\\ &\{X=[R],Y=[\,],Z=[R]\},\\ &\{X=[R,\hat{t}|\hat{u}],Y=[\,],Z=[R,\hat{v}|\hat{w}]\},\end{aligned}$$

$$
\begin{aligned}
&\{X = [\,], Y = [S], Z = [S]\},\\
&\{X = [R], Y = [S], Z = [R, \hat{v}|\hat{w}]\},\\
&\{X = [R, \hat{t}|\hat{u}], Y = [S], Z = [R, \hat{v}|\hat{w}]\},\\
&\{X = [\,], Y = [S, \hat{t}|\hat{u}], Z = [S, \hat{v}|\hat{w}]\},\\
&\{X = [R], Y = [S, \hat{t}|\hat{u}], Z = [R, \hat{v}|\hat{w}]\},\\
&\{X = [R, \hat{t}|\hat{u}], Y = [S, \hat{p}|\hat{q}], Z = [R, \hat{v}|\hat{w}]\}\big\}
\end{aligned}
$$

which is simply the depth(2) *cut of the infinite set* $\big\{\{X = [X_1, \ldots, X_n], Y = [Y_1, \ldots, Y_m], Z = [X_1, \ldots, X_n, Y_1, \ldots, Y_m] \mid n, m > 0\big\}$.

In presence of post-conditions only, the clause `c2` *wrong, since the variable* `X` *can unify with any term. This does not hold in presence of a pre-condition stating that the first and second argument of* `append` *should be lists. Thus with such a specification the I/O correctness method is able to verify the correctness of this program.*

5 Assertions and specification languages

In program verification, a specification is usually given by means of assertions (i.e. formulas in a suitable formal specification language), while in our case a specification is (extensionally defined as) the intended abstract semantics. On the other hand, assertions clearly define an abstract domain (as shown by the Cousot's in the early papers on abstract interpretation).

We will consider the case of success-correctness only, where the concrete domain consists of sets of atoms. Similar constructions can be given for the other notions of correctness.

Let us consider a first order language $\mathcal{L} = \langle \Sigma, \Pi, V \rangle$. Let $\mathcal{F}$ be a set of formulas (*assertions*) of $\mathcal{L}$, expressing properties of predicate arguments. We assume the signature of $\mathcal{L}$ to include functions, constants and variables of the programs we want to verify.

We need to define what it means for an atom $p(t)$ *to satisfy a property* Φ of $\mathcal{F}$. This can be done either semantically or syntactically (in terms of derivability from a theory). We just show the semantic version. We choose a term-interpretation $\mathcal{I} = \langle \mathit{Terms}(\Sigma, V), \Sigma_{\mathcal{I}}, \Pi_{\mathcal{I}} \rangle$ (the set of non-ground terms viewed as an $\mathcal{L}$ structure). An atom $p(\mathcal{T}1, \ldots, \mathcal{T}n)$ *satisfies* the formula $\Phi\,[x_1, \ldots, x_n]$ (of $\mathcal{F}$), if and only if for each σ,

$$\mathcal{I} \models_{\sigma[x_1,\ldots,x_n \backslash \mathcal{T}1,\ldots,\mathcal{T}n]} \Phi\,[x_1, \ldots, x_n]\,.$$

Given the set of assertions $\{\Theta_p\}_{p \in \Pi}$, which associates an assertion Θ_p to each predicate p, a program P is *success-correct* with respect to $\{\Theta_p\}_{p \in \Pi}$, if and only if $\forall p(t) \in \mathit{Atoms},\ p(t) \overset{\theta}{\rightsquigarrow} \Box$ implies that $p(t)\theta$ satisfies Θ_p.

A natural pre-order is induced on $\mathcal{F}$ by implication under the interpretation $\mathcal{I}$, i.e.

$$\Theta \preceq_{\mathcal{I}} \Phi \text{ iff } \mathcal{I} \models \Theta \Rightarrow \Phi.$$

By abuse of notation, we still denote by $\preceq$ the induced partial order. Let us now define the domain

$$\mathbb{A}_{\mathcal{I}} = (\Pi \rightarrow \mathcal{F}/_{\equiv_{\mathcal{I}}}, \leq),$$

whose elements can be represented as sets $\{\Theta_p\}_{p\in\Pi}$, where each Θ_p is a formula of $\mathcal{F}$ with free variables corresponding to arguments of p. The order is given by the pointwise extension of the order between formulas of $\mathcal{F}$.

Consider now the following function from $\mathbb{A}_{\mathcal{I}}$ to $\mathbb{C}$:

$$\gamma_{\mathcal{I}}(\Theta) = \lambda p(\mathbf{x}).\{p(\mathbf{t}) \in \mathit{Atoms} \mid p(\mathbf{t}) \textit{ satisfies } \Theta_p\}.$$

If $\mathcal{F}$ is a complete lattice, closed under conjunction, the function $\gamma_{\mathcal{I}}$ is meet-additive. Then, by standard abstract interpretation results, it induces a Galois connection between $\mathbb{C}$ and $\mathbb{A}$.

Hence we can define the best abstraction of $\mathcal{T}_P^{\mathbb{C}}$ on $\mathbb{A}_{\mathcal{I}}$.

$$\mathcal{T}_P^{\mathcal{I}}(\Theta) = \lambda p(\mathbf{x}). \bigvee_{p(\mathbf{t})\leftarrow p_1(\mathbf{t}_1),\ldots,p_n(\mathbf{t}_n)\in P} \bigwedge\{\Phi \mid \mathcal{I} \models \bigwedge_{i=1}^{n} \Theta_{p_i}[\mathbf{x}_i\backslash\mathbf{t}_i] \Rightarrow \Phi[\mathbf{x}\backslash\mathbf{t}]\}$$

It can be shown that P is *success-correct* with respect to $\{\Theta_p\}_{p\in\Pi}$, if and only if $\alpha_{\mathcal{I}}(\mathit{lfp}(\mathcal{T}_P^{\mathbb{C}})) \leq \Theta$, where $\alpha_{\mathcal{I}}$ is the adjoint function of $\gamma_{\mathcal{I}}$. A sufficient condition for success correctness can then be obtained from the condition $\mathcal{T}_P^{\mathcal{I}}(\Theta) \leq \Theta$.

Theorem 5.1 *Let P be a logic program and $\{\Phi_p\}_{p\in\Pi}$ be assertions of $\mathcal{F}$. A sufficient condition for P to be success-correct with respect to $\{\Phi_p\}_{p\in\Pi}$ is that for each clause $p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n)$ it is true that*

$$\mathcal{I} \models \bigwedge_{i=1}^{n} \Phi_{p_i}[\mathbf{x}_i\backslash\mathbf{t}_i] \Rightarrow \Phi_p[\mathbf{x}\backslash\mathbf{t}] \tag{4}$$

This result is essentially the verification method proposed by Deransart [18].

If the relation $\models$ is decidable, we have an effective test to check the conditions. As an example, we can take the language of properties by Marchiori [35,36], which allows us to express groundness, freeness and sharing of terms. [35] contains also a decidable axiomatization for a fragment of the language. A decidable extension of the same language, including polymorphic types, is described in [40]. Another example of a decidable specification language, oriented to the verification of functional programs (in a simple first order language) can be found in [32].

Example 5.2 *Let us consider the following* naive sort*, where we assume the procedure leq(X, Y), which is successful if X and Y are numbers and $X \leq Y$, and the procedure, perm(Xs, Ys), which returns in Ys a permutation of the list in Xs.*

```
c1:  sort(Xs,Ys) :- perm(Xs,Ys), ord(Ys).
c2:  ord([]).
c3:  ord([X,Y|Zs]) :- leq(X, Y), ord([Y|Zs]).
```

The programs is correct w.r.t. the following specification.

$$\mathcal{I}^{\alpha} := \begin{cases} sort(X,Y) & \mapsto glist(X) \rightarrow glist(Y) \\ perm(X,Y) & \mapsto glist(X) \rightarrow glist(Y) \\ ord(X) & \mapsto glist(X) \rightarrow \mathbf{true} \\ leq(X,Y) & \mapsto ground(X) \wedge ground(Y) \rightarrow \mathbf{true}, \end{cases}$$

where the predicates ground(X) and glist(X) specify ground terms and lists of ground terms, respectively.

Let us consider now a small change in the program, obtained by inverting the order of the predicates in the body of the clause `c3`*, obtaining*

```
c3':  sort(Xs,Ys) :- ord(Ys), perm(Xs,Ys).
```

In this case we have that the predicate ord may be called with a non ground argument, even if the predicate sort is called correctly w.r.t. its pre-condition. This possibly wrong situation is detected by observing that the verification condition glist$(Xs) \Rightarrow$ glist(Ys) associated to the first clause is false.

Concerning the completeness of the axiomatic method, the same result of extensional properties holds, namely the method is complete if the abstraction is precise. In the case of assertions, this means that the strongest set of assertions $\{\varphi_p\}_{p \in \Pi}$, for which P is success-correct, verifies condition 4. Obviously this strongly depends on the choice of the language and of the set of properties $\mathcal{F}$.

We have shown that assertions can indeed be handled as abstract domains. The corresponding verification methods are practical only if we can effectively derive, for a given assertion language, the abstraction function $\alpha_{\mathcal{I}}$, which is needed to design the optimal semantic operators. This would also lead to a notion of abstract execution on the domain of assertions. Other open interesting issues are the problem of defining more expressive (decidable) specification languages, the problem of comparing (from the viewpoint of precision) standard "static analysis" abstract domains to assertions, and the problem of applying refinement operators to domains defined by assertions.

6 Finite failure and infinite derivations

All the observables which can be defined in the semantic framework of Section 3 are abstractions of the set of successful derivations. Hence we cannot handle properties such as finite failure. Finite failure was shown to have some of the properties which are relevant to verification, e.g. *AND*-compositionality [29]. However, a fixpoint semantics correctly modeling finite failure in an *AND*-compositional way did not exist. Our approach is to derive such a semantics, by starting from a concrete traces semantics [28], which extends with infinite computations the traces semantics in [11], and by defining an abstract domain D^{ff}, chosen so as to model finite failure and to make the abstract operator $\mathcal{T}_P^{ff}$ precise. The corresponding abstract fixpoint semantics, $lfp(\mathcal{T}_P^{ff})$, is the Non-Ground Finite Failure set. It correctly models finite failure and is *AND*-compositional.

Once we have a fixpoint semantics modeling finite failure, we can use the standard condition (2) as a sufficient condition for the correctness w.r.t. finite failure.

In [30], stronger verification conditions are generated, by using Ferrand's approach [22], based on two specifications. Ferrand uses the standard ground immediate consequences operator $\mathcal{T}_P$, while the specifications are $\mathcal{S}$, intended $lfp(\mathcal{T}_P)$, and $\mathcal{S}'$, intended $gfp(\mathcal{T}_P)$. The standard sufficient condition for partial correctness ($\mathcal{T}_P(\mathcal{S}) \subseteq \mathcal{S}$) allows us to reason about the ground success set., while the new sufficient condition ($\mathcal{S}' \subseteq \mathcal{T}_P(\mathcal{S}')$) is somewhat related to sufficiency or missing answers (using the declarative debugging terminology).

We adopt Ferrand's approach by replacing $gfp(\mathcal{T}_P)$ by $\mathcal{T}_P \downarrow \omega$. $\mathcal{T}_P^{ff}$ is in fact not co-continuous (as is the case for Ferrand's $\mathcal{T}_P$). In the case of finite failure, the complement of $\mathcal{T}_P^{ff} \downarrow \omega$ has a very interesting characterization as the set of (possibly non-ground) atoms which do not have a successful derivation (called *unsolvable* in [5]) We can then provide a specification $\mathcal{S}'$ of the complement of the set of atoms which are intended to succeed and derive another meaningful sufficient condition $\mathcal{S}' \subseteq \mathcal{T}_P^{ff}(\mathcal{S}')$, which guarantees that the actual set of successful atoms is included in the intended one.

The above conditions are not effective because $\mathcal{T}_P^{ff}$ is not finitary and because both $\mathcal{S}$ and $\mathcal{S}'$ are infinite sets. However, the analysis and verification of properties of finite failure, can be based on effective approximations of the operator. Using two semantics and two specifications will allow us to use two different (related) abstractions, an upward approximation (of the least fixpoint semantics) and a downward approximation (of $\mathcal{T}_P^{ff} \downarrow \omega$), both related to the $depth(k)$ abstraction. The two abstractions give finite approximations of the Non-Ground Finite Failure set and of the complement of the success set respectively (see [30] for the technical construction of the two abstractions). Note that the idea of considering upward and downward approximations for

verification and debugging has also recently been proposed in [6].

As already mentioned, the upward abstraction α^{up} is applied to the least fixpoint of $\mathcal{T}_P^{ff}$, while the downward abstraction α^{dw} is applied to $\mathcal{T}_P^{ff} \downarrow \omega$. The two corresponding specifications are

- $\mathcal{S}_{\alpha^{up}}$ is the α^{up} abstraction of the intended Non-Ground Finite Failure set.
- $\mathcal{S}'_{\alpha^{dw}}$ is the α^{dw} abstraction of the intended set of atoms which either finitely fail or (universally) do not terminate. Alternatively, $\mathcal{S}'_{\alpha^{dw}}$ can be viewed as the complement of the set of atoms (of depth $\leq k$) which have a successful derivation.

A program P is correct w.r.t. the finitely failed atoms not deeper than k if

$\boldsymbol{c_1}$ $\alpha^{up}(lfp(\mathcal{T}_P^{ff})) \subseteq \mathcal{S}_{\alpha^{up}}$,

$\boldsymbol{c_2}$ $\mathcal{S}'_{\alpha^{dw}} \subseteq \alpha^{dw}(\mathcal{T}_P^{ff} \downarrow \omega))$.

If the above conditions are satisfied, the program is correct w.r.t. finitely failed atoms not deeper than k *and* the set of $depth(k)$ successful atoms is correct w.r.t. the complement of $\mathcal{S}'_{\alpha^{dw}}$.

The following theorem gives us sufficient effectively computable conditions for $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$ to hold.

Theorem 6.1 *Let P be a program. If the following conditions are satisfied*

$\boldsymbol{sc_1}$ ${\mathcal{T}_P^{ff}}^{up}(\mathcal{S}_{\alpha^{up}}) \subseteq \mathcal{S}_{\alpha^{up}}$,

$\boldsymbol{sc_2}$ $\mathcal{S}'_{\alpha^{dw}} \subseteq {\mathcal{T}_P}^{ff^{dw}}(\mathcal{S}'_{\alpha^{dw}})$,

then P is correct w.r.t. finitely failed atoms not deeper than k and the w.r.t.the set of successful atoms not deeper than k.

One may wonder whether there exist other abstract domains which can be used to derive meaningful sufficient conditions for effective verification of finite failure. One idea which we are currently pursuing is to use the abstract domain of assertions as discussed in Section 5. Another problem, on which we are currently working and which can be tackled starting from the "infinite derivations" traces concrete semantics, is the problem of reasoning about termination.

Example 6.2 *Let us consider the following program meant to append lists of integers where in the clause* `c4` *instead of* `nat(s(X))` *we have* `nat(s(0))`.

```
c1:  append([X|Xs], Ys, [X|Zs]) :- nat(X), append(Xs, Ys, Zs).
c2:  append([], Xs, Xs).
c3:  nat(0).
c4:  nat(s(0)) :- nat(X).
```

Intuitively this error increments the set of atoms which have a finite failure in the program w.r.t. the set of atom which should have a finite failure in the intended program.

If we consider as specification the upward approximation $S_{\alpha^{up}}$, on the depth(k) domain, of the set of atoms which should have a finite failure, we find that $\mathcal{T}_{P_1}^{ff^{up}}(S_{\alpha^{up}}) \subseteq S_{\alpha^{up}}$. Hence (by Theorem 6.1) the program is not correct w.r.t. the intended depth(k) finite failure set.

Note that for the correct intended program P conditions $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$ are verified. Thus we can conclude that P is correct w.r.t. finitely failed atoms not deeper than k and also w.r.t. the successful atoms not deeper than k.

7 Future developments

One of the most interesting challenges for researchers in the fields of programming languages semantics and formal methods is the verification of code coming from untrusted sources. One promising solution to the above problem is the approach known as *Proof Carrying Code (PCC)* [37]. According to the PCC approach, mobile code is supplied with a formal proof that it satisfies a specification defined by the host system, which will then simply check the proof, to ensure that the code complies with its policy. One step forward is the idea of extending compilers with proof-generation capabilities (*certifying compiler*). In most current (experimental) implementations of the PCC idea, properties expressed in specifications are essentially type properties and certification boils down to type inference and verification.

We believe that abstract interpretation based verification techniques might be very useful in the PCC approach. On one side, abstract interpretation makes it possible to reconstruct in a unifying framework different type systems, proof methods, and dataflow analysis techniques. On the other side, it makes available several systematic design techniques (systematic design of the abstract semantics, systematic design of domains), which might improve the reliability and flexibility of verifiers. Last but not least, abstract interpretation based verification methods are scalable, since they are intrinsically modular (since they are based on a compositional semantics).

There exist several semantics-based techniques which can be useful also in the design of certifying compilers. One such a technique is the combination of abstract interpretation and partial evaluation. One experiment we are currently performing (for an eager higher order functional language) might be relevant. We have a standard interpreter (formally derived from a denotational definition) and an abstract interpreter which computes on a suitable domain of properties (a standard analysis abstract domain in the current experiment, a domain of assertions in the future). We combine (by a sort of reduced product operation) the two interpreters into a single interpreter. The specialization (by partial evaluation) of the combined interpreter w.r.t. a program P returns the compiled code for P, together with the code which computes the abstract property. One can build on this idea to define a tool, which generates (to-

gether with the compiled code) a proof that the inferred property satisfies the specification.

Of course, mobile code is not currently written in declarative languages. Our future plans include an attempt to apply our results to programming languages which are closer to those which are used to program mobile code in the current practice.

References

[1] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[2] A. Bossi and N. Cocco. Verifying correctness of logic programs. In J. Diaz and F. Orejas, editors, *Proceedings of TAPSOFT'89*, pages 96–110, 1989.

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19–20:149–197, 1994.

[4] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation '93*, pages 46–55, 1993.

[5] M. Bruynooghe, H. Vandecasteele, D.A. de Waal, and M. Denecker. Detecting unsolvable queries for definite logic programs. In Palamidessi, Glaser, and Meinke, editors, *Proc. 10th Int'l Conf. and Symp. Principle of Declarative Programming*, Lecture Notes in Computer Science, pages 118–133. Springer-Verlag, 1998.

[6] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In M. Kamkar, editor, *Proceedings of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169. Linköping University, 1997.

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

[8] M. Comini. *An Abstract Interpretation Framework for Semantics and Diagnosis of Logic Programs.* PhD thesis, Dipartimento di Informatica, Universitá di Pisa, 1998.

[9] M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Information and Computation*, 1999. To appear.

[10] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

[11] M. Comini and M. C. Meo. Compositionality properties of *SLD*-derivations. *Theoretical Computer Science*, 211(1-2):275–309, 1999.

[12] A. Cortesi, B. Le Charlier, and S. Rossi. Specification-based automatic verification of prolog programs. In *Proceedings of LOPSTR '96*, volume 1207 of *Lecture Notes in Computer Science*, pages 38–57. Springer-Verlag, 1996.

[13] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. In S. Brookes and M. Mislove, editors, *Proceedings of the 13th International Symposium on Mathematical Foundations of Programming Semantics MFPS'97*, volume 6 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam, 1997. URL:http://www.elsevier.nl/locate/entcs/volume6.html.

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

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

[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 *Proceedings of Nineteenth Annual ACM Symp. on Principles of Programming Languages*, pages 83–94. ACM Press, 1992.

[18] P. Deransart. Proof methods of declarative properties of definite programs. *Theoretical Computer Science*, 118(2):99–166, 1993.

[19] W. Drabent. It is Declarative. In *Workshop on Verification, Model Checking and Abstract Interpretation, ILPS'97*, 1997.

[20] W. Drabent and J. Maluszynski. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

[21] G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4:177–198, 1987.

[22] G. Ferrand. The notions of symptom and error in declarative diagnosis of logic programs. In P. A. Fritzson, editor, *Automated and Algorithmic Debugging, Proceedings of AADEBUG '93*, volume 749 of *Lecture Notes in Computer Science*, pages 40–57. Springer-Verlag, 1993.

[23] G. Filè, R. Giacobazzi, and F. Ranzato. A unifying view on abstract domain design. *ACM Computing Surveys*, 28(2):333–336, 1996.

[24] M. Gabbrielli, G. Levi, and M. C. Meo. Observable Behaviors and Equivalences of Logic Programs. *Information and Computation*, 122(1):1–29, 1995.

[25] R. Giacobazzi and F. Ranzato. Functional dependencies and Moore-set completions of abstract interpretations and semantics. In J. Lloyd, editor, *Proceedings of the 1995 Interantional Symposium on Logic Programming (ILPS'95)*, pages 321–335. The MIT Press, 1995.

[26] R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: A domain perspective. In M. Johnson, editor, *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997.

[27] R. Giacobazzi and F. Scozzari. Intuitionistic implication in abstract interpretation. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of Ninth International Symposium on Programming Languages, Implementations, Logics and Programs PLILP'97*, volume 1292 of *Lecture Notes in Computer Science*, pages 175–189. Springer-Verlag, 1997.

[28] R. Gori. A fixpoint semantics for reasoning about finite failure. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *LNAI*, pages 238–257, Tbilisi, Republic of Georgia, 1999. Springer.

[29] R. Gori and G. Levi. Finite failure is and-compositional. *Journal of Logic and Computation*, 7(6):753–776, 1997.

[30] R. Gori and G. Levi. On the verification of finite failure. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, LNCS, pages 311–327, Paris, France, 1999. Springer.

[31] B. Le Charlier, C. Leclère, S. Rossi, and A. Cortesi. Automatic verification of behavioral properties of prolog programs. In *Proceedings of ASIAN '97*, volume 1345 of *Lecture Notes in Computer Science*, pages 225–237. Springer-Verlag, 1997.

[32] D. Le Métayer. Proving properties of programs defined over recursive data structures. In *Proceedings ACM PEPM '95*, pages 88–99, 1995.

[33] G. Levi and F. Spoto. An Experiment in Domain Refinement: Type Domains and Type Representations for Logic Programs. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Priciples of Declarative Programming. 10th International Symposium, PLILP'98*, volume 1490 of *Lecture Notes in Computer Science*, pages 152–169, Pisa, Italy, September 1998. Springer-Verlag.

[34] G. Levi and P. Volpe. Derivation of proof methods by abstract interpretation. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Priciples of Declarative*

Programming. 10th International Symposium, PLILP'98, volume 1490 of *Lecture Notes in Computer Science*, pages 102–117, Pisa, Italy, September 1998. Springer-Verlag.

[35] E. Marchiori. A Logic for Variable Aliasing in Logic Programs. In G. Levi and M. Rodriguez-Artalejo, editors, *Proceedings of the 4th International Conference on Algebraic and Logic Programming (ALP'94)*, volume 850 of *Lecture Notes in Computer Science*, pages 287–304. Springer Verlag, 1994.

[36] E. Marchiori. Design of Abstract Domains using First-order Logic. In M. Hanus and M. Rodriguez-Artalejo, editors, *Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96)*, volume 1139 of *Lecture Notes in Computer Science*, pages 209–223. Springer Verlag, 1996.

[37] G. Necula. Proof-carrying code. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, 1997.

[38] F. Scozzari. Logical optimality of groundness analysis. In P. Van Hentenryck, editor, *Proceedings of International Static Analysis Symposium, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1997.

[39] E. Y. Shapiro. Algorithmic program debugging. In *Proceedings of Ninth Annual ACM Symp. on Principles of Programming Languages*, pages 412–531. ACM Press, 1982.

[40] P. Volpe. A first-order language for expressing aliasing and type properties of logic programs. In G. Levi, editor, *Static Analysis, Proceedings of the 5th International Symposium, SAS'98*, volume 1503 of *Lecture Notes in Computer Science*, pages 184–199. Springer-Verlag, 1998.

[41] P. Volpe. *Derivation of proof methods for logic programs by abstract interpretation*. PhD thesis, Dipartimento di Matematica, Università di Napoli Federico II, 1999.