A First-Order Language for Expressing Aliasing and Type Properties of Logic Programs

Paolo Volpe

Abstract

In this paper we study a first-order language that allows to express and prove properties regarding the sharing of variables between non-ground terms and their types. The class of true formulas is proved to be decidable through a procedure of elimination of quantifiers and the language, with its proof procedure, is shown to have interesting applications in validation and debugging of logic programs. An interesting parallel is pointed out between the language of aliasing properties and the first order theories of Boolean algebras.

Keywords: Verification of logic programs, languages of specification, first-order logic.

1 Introduction

In the methods proposed in [12][1][11][15], an assertional language is required that allows to express the properties of programs one is interested in. Some verification conditions are provided that imply the partial correctness of the programs with respect to various aspects of the computations. For example the method proposed in [11] and [5] allows to prove properties of the correct answers of the programs, while in [15] a method is provided to prove properties of the computed answers. The methods proposed in [12] and [1] allow to prove, in addition, that predicates verify given specifications at call time.

In this paper we study a language that allows to express an interesting class of properties of non-ground terms, that is the data on which logic program operate. The language is sufficiently expressive to capture sharing, freeness and types of non-ground terms. Two or more terms are said to *share*, when they have at least one variable in common, while a term is *free* when it is a simple variable. For *types*, we refer to term properties like being a list, a tree, a list of ground terms, etc.

Fragments of this language have already been studied in [16] and [2] and shown to be decidable, but in this paper we show that the full first-order theory is decidable. This allows to use its full expressive power in existing proof methods and algorithmically decide whether the verification conditions are true or not. Indeed for many methods the verification conditions are expressed by formulas of the language. If the verification condition is true, we show the partial correctness of logic programs with respect to a property belonging to the class of aliasing properties and type assertions. If the verification condition is false (and we stress that this can be checked in finite time), obviously this does not

P. Volpe is with the Dipartimento di Informatica, Università di Pisa. Corso Italia 40, 56125 Pisa, Italy. e-mail: volpep@di.unipi.it. tel: +39-50-887248. fax: +39-50-887226.

mean that the program has necessarily an error. Anyway a "warning" can be raisen up, signalling a possible wrong situation. The proof procedure shown in the paper can be easily enriched so as to provide a counterexample in this case. This allows the user to have more information about the warning and to decide whether to change the program (the counterexample is actually a wrong computation of the program) or to refine the specification (the verification condition is false because the specification is too "loose" and impossible computations are considered).
The proof of decidability is based on the method of elimination of quantifiers and points out an interesting set of formulas, which can be viewed as expressing constraints on the cardinality of the sets of variables that can occur in terms. Our proof is based on the parallel between the satisfiability of formulas of our language and the satisfiability of such cardinality constraints, which can be proven decidable as a consequence of the decidability of the theory of Boolean algebras. We think that such class of constraints, which has a quite simple representation and operations of composition and cylindrification, can be of interest in program analysis. For example, well known abstract domains such as $\mathcal{POS}$ [8] and *Sharing*[13] can be naturally viewed as subdomain of the class of cardinality constraints (cfr. also with [17]), with their composition and the cylindrification operator obtained as instances of the general ones.

2 Regular term grammars

To specify families of types we will consider regular term grammars. There is, in literature, a large amount of papers on regular types. They have proved them to be a good trade-off between expressibility and decidability. In fact they are strictly more expressive than regular languages, but strictly contained in context-free languages (which have an undecidable subset relation). Our main references are the papers of Dart and Zobel [9, 10] and Boye and Maluszynski [3, 2].

A *regular term grammar* is a tuple $G = (\Sigma, \mathcal{V}, \mathbb{T}, \mathbb{R})$, where Σ is a set of function symbols, $\mathcal{V}$ is an infinite denumerable set of variables, $\mathbb{T}$ is a finite set of *type symbols*, including *var* and *any*, and $\mathbb{R}$ is a finite set of rules $l \to r$ where

- $l \in (\mathbb{T}/\{var, any\})$
- $r \in Terms(\Sigma, \mathbb{T})$

For every $T \in \mathbb{T}/\{var, any\}$, we define $Def_G(T)$ (also denoted by $Def_{\mathbb{R}}(T)$) as the set $\{r \mid T \to r \in \mathbb{R}\}$. $Def_G(var)$ is defined as the set of variable $\mathcal{V}$, while $Def_G(any)$ as the set $Terms(\Sigma, \mathcal{V})$. We use the notation $T_1 \to_G T_2$ if T_2 is obtained from T_1 by replacing a symbol $T \in \mathbb{T}$ by a term in $Def_G(T)$. Let $\twoheadrightarrow_G$ be the transitive and riflessive closure of $\to_G$. Given the type symbol $T \in \mathbb{T}$, we define the set of terms $[T]_G$, the *type* T, as the set $\{s \in Terms(\Sigma, \mathcal{V}) \mid T \twoheadrightarrow_G s\}$. Notice that $[var]_G = \mathcal{V}$ and $[any]_G = Terms(\Sigma, \mathcal{V})$. We assume function symbols in Σ to contain at least a constant and a function of arity 2. We will often omit the subscript when the grammar is clear from the context. Regular term grammars enjoy several remarkable properties. In fact the emptiness and the subset relation of regular types is decidable and for each symbol type T, the set $[T]_G$ is decidable [9]. Moreover, given a grammar G and two type symbols T and S, there is an algorithm which extends G into G', with a new symbol $S \cap T$ and new rules in such a way that $[S \cap T]_{G'} = [S]_G \cap [T]_G$.

In this paper we will be mainly concerned with *closed discriminative* regular grammars in *normal form.*

Definition 1
A regular term grammar $G = (\Sigma, \mathcal{V}, \mathbb{T}, \mathbb{R})$ is in *normal form* if each rule have the form $T \to var$ or

$$T \to f(T_1, \ldots, T_n)$$

with $f^{(n)} \in \Sigma$, $T \in \mathbb{T}/\{var, any\}$and $T_1, \ldots, T_n \in \mathbb{T}$. ∎

It can be easily shown that each type can be defined by a grammar in normal form.

Definition 2
A regular term grammar G is *discriminative* if it is in normal form and, for each type symbol T, the top functors in $Def_G(T)$ are pairwise distinct. It is *closed* if it is in normal form and, for each type symbol T, the symbol *var* does not occur in any element of $Def_G(T)$. ∎

Notice that most of the types used in logic programming allow closed and discriminative term grammars. It can be easily shown that regular types defined by a closed grammar are indeed closed under substitution [2]. The types that can be defined by a discriminative and closed grammar will be referred to as *simple types.*

Given a discriminative and closed regular term grammar G and two type symbols T and S, we want to extend it to G' with a new symbol T/S and new rules in such a way that $[T/S]_{G'} = [T]_G / [S]_G$, if it is not the case that $[T]_G \subseteq [S]_G$. We provide an algorithm which computes the difference T/S, assuming that type S is simple. In general the grammar G' need not to be nor closed nor discriminative in general. For example, the difference type $[any] / [inst]$ is equal to the set of variables $\mathcal{V}$, which can not be defined by any closed grammar.

Difference Algorithm

INPUT. Two type symbols T and S and the set $\mathbb{R}$ of rules defining T and S, with S simple.
OUTPUT. A pair $(T/S, \mathbb{S})$, where T/S is defined by the rules in $\mathbb{S}$, if $\mathbb{S} \neq \emptyset$; otherwise the difference is empty ($[T] \subseteq [S]$).
METHOD. The algorithm is defined by the following recursive function. A set I of difference symbols T/S is used to ensure termination.

$$\begin{array}{lcl} \mathit{difference}(T, S, \mathbb{R}) & = & \mathit{difference}(T, S, \mathbb{R}, \emptyset) \\ \mathit{difference}(T, S, \mathbb{R}, I) & = & \end{array}$$

- If $T \subseteq S$ then return $(T/S, \emptyset)$;
- If the symbol T/S is in I then retun $(T/S, \mathbb{R})$;
- Otherwise, let $Def_{\mathbb{R}}(T) = \{r_1, \ldots, r_k\}$. For each $i \in \{1, \ldots, k\}$, let H_i be defined as follows:
  - if $r_i = var$ or ($r_i = f_i(T_1, \ldots, T_{n_i})$ and the functor f_i does not occur in $Def_{\mathbb{R}}(S)$) then let $H_i = \{T/S \to r_i\}$;
  - If $r_i = f_i(T_1, \ldots, T_{n_i})$ and $f_i(S_1, \ldots, S_{n_i}) \in Def_{\mathbb{R}}(S)$ then let $(T_j/S_j, \mathbb{S}_j) = \mathit{difference}(T_j, S_j, \mathbb{R}, I \cup \{T/S\})$, for each $j = \{1, \ldots, n_i\}$, and let $H_i = \bigcup_{\mathbb{S}_j \neq \emptyset} \{T/S \to f_i(T_1, \ldots, T_j/S_j, \ldots, T_{n_i})\} \cup \mathbb{S}_j$

Return $(T/S, \mathbb{R} \cup \bigcup_{i=1}^{k} H_i)$

Lemma 2.1 *Let T and S be two type terms defined by the rules of $\mathbb{R}$, S simple. Then difference$(T, S, \mathbb{R})$ terminates and returns a pair $(T/S, \mathbb{S})$ such that $[T/S]_{\mathbb{S}} = [T]_{\mathbb{R}} \,/\, [S]_{\mathbb{R}}$.*

To carry on the elimination of quantifiers in the next section, we need to know the cardinalities of the sets of variables which may occur in a term of a given type. The *var-cardinality* of T, written $|T|$, is defined as the set $\{|Vars(t)| \mid t \in [T]_G\}$. In other terms $k \in |T|$ if and only if there exists $t \in [T]$ such that $|Vars(t)| = k$. We can prove in a straightforward way the following lemma.

Lemma 2.2 *Given a simple type T, then $|T|$ is equal to $\{0\}$ or to ω.*

For difference types $(T_1 \cap \cdots \cap T_n)/S_1/\cdots/S_k$, with $T_1, \ldots, T_n$ and $S_1, \ldots, S_k$ simple types, we can show the following theorem.

Theorem 2.3 *The var-cardinality of the type $(T_1 \cap \cdots \cap T_n)/S_1/\cdots/S_k$, with $T_1, \ldots, T_n$ and $S_1, \ldots, S_k$ simple types, is equal to $S \cup [k, \omega]$, where S is a finite set of natural number and $[k, \omega]$, with $k = 1, \ldots, \omega$ is the set of natural numbers greater or equal to k .*

The proof provides an effective procedure to compute the var-cardinality of a difference type.

3 A language of properties

We introduce a language that allows to express properties of terms used in static analysis and verification of logic programs: these include groundness, freeness, sharing, type assertions. The language is parametric with respect to a family of types defined through a regular term grammar. It is an extension of the language proposed by Marchiori in [16].

We assume a regular term grammar $G = (\Sigma, \mathcal{V}, \mathbb{T}, \mathbb{R})$, discriminative and closed, describing the family of types we are interested in. As before, the set of function symbols Σ is assumed to contain at least a constant and a function of arity 2. The first-order language $\mathcal{L}_G = \langle \Sigma, \Pi, \mathcal{V} \rangle$ will have the set of predicate symbols Π consisting of the predicates $var^{(1)}$, $share^{(n)}$, for each natural n, and a unary predicate $p_T^{(1)}$ for each symbol type $T \in \mathbb{T}/\{any\}$.
Like in [16], we give the semantics of formulas $\mathcal{L}$ by considering the non-ground Herbrand interpretation $\mathcal{H} = \langle Terms(\Sigma, \mathcal{V}), \Lambda, \Gamma \rangle$. Given a state σ, the relation $\models_\sigma$ is defined on atoms as follows.

- $\mathcal{H} \models_\sigma var(t)$ iff $\sigma(t) \in \mathcal{V}$;
- $\mathcal{H} \models_\sigma share(t_1, \ldots, t_n)$ iff $\bigcap_{i=1}^{n} Vars(\sigma(t_i)) \neq \emptyset$;
- $\mathcal{H} \models_\sigma p_T(t)$ iff $\sigma(t) \in [T]_G$, with $T \in \mathbb{T}/\{any\}$.

The semantics of the other formulas of $\mathcal{L}$ can be derived as usual. We will often write $\models \varphi$ (resp. $\models_\sigma \varphi$) for $\mathcal{H} \models \varphi$ (resp. $\mathcal{H} \models_\sigma \varphi$). Formulas expressible in $\mathcal{L}_G$ are, for example, $\forall V\ var(V) \Rightarrow \neg share(V, X)$ which asserts the groundness of X; or $list(X) \wedge \exists V\ var(V) \wedge share(V, X) \wedge (\forall W\ var(W) \wedge share(W, X) \Rightarrow share(V, W))$ which says that X is a list in which exactly one variable occurs.

An important class of formulas of $\mathcal{L}_G$, which are often considered in analysis and verification, is the class of *monotone formulas*, that is the formulas φ such that $\models_\sigma \varphi$ and $\sigma \leq \sigma'$ implies $\models_{\sigma'} \varphi$. For example, $ground(X)$ and $\neg var(X)$ are monotone properties, while $var(X)$ and $share(X,Y)$ are not. Since the grammar G is closed each atom $p_{T_1}(X)$ is monotone. An interesting subclass of monotone properties are the dependences like $\forall V\ var(V) \wedge share(V,Y) \Rightarrow share(V,X)$, which could be read informally as saying that if X is instantiated to a ground value then also Y is.
Notice that properties expressible in the language $\mathcal{L}_G$ are invariant with respect to the name of variables. That is, for each φ, formula of $\mathcal{L}$, if $\models_\sigma \varphi$ and σ' is a variant of σ then $\models_{\sigma'} \varphi$.

4 A proof procedure for $\mathcal{L}$

We are interested in characterizing the set of formulas $Th_{\mathcal{L}}(\mathcal{H})$, that is the formulas of the language $\mathcal{L}$ which are true in the interpretation $\mathcal{H}$.
In [16], it is proposed a proof procedure to decide the validity of formulas $\exists(\varphi_1 \wedge \cdots \wedge \varphi_n)$ where each atom φ_i is an atom $var(t)$, $ground(t)$, $share(t_1, \ldots, t_n)$, or its negation. It is also known that the implication between regular types, that is formulas like $\forall(p_{T_1}(t_1) \wedge \cdots \wedge p_{T_n}(t_n) \Rightarrow p_T(t))$ are decidable [2]). It is not clear whether considering the full first order theory, such as in $\mathcal{L}$, the language is still decidable. In this section we will show that this is indeed the case and it is not such a trivial extension of those previous results.

To show that $Th_{\mathcal{L}}(\mathcal{H})$ is recursive we will use the method of *elimination of quantifiers* [4, 14]. We single out a set Ω, the *elimination set*, of fomulas of $\mathcal{L}$ and show that each formula of $\mathcal{L}$ is equivalent in $\mathcal{H}$ to a boolean combination of formulas of Ω. Once proved the decidability of formulas in Ω, we end up with a complete decision procedure for formulas of $\mathcal{L}$.
We will use the abbreviation $\exists_{\geq k} var(V)\ \phi$ to say that there exist at least k distinct variables which verify formula ϕ. To carry on the elimination of quantifiers we will need a particular class of formulas, which we call cardinality constraints.

Definition 3
Let $\mathcal{B}$ be the class of *boolean terms* on $\mathcal{V}$, that is the terms built from the signature $(\{\cap^{(2)}, \cup^{(2)}, \neg^{(1)}, 0^{(0)}, 1^{(0)}\}, \mathcal{V})$. Fixed a natural k and a boolean term t, a *simple cardinality* constraint $\alpha_k(t)$ is defined as the formula $\exists_{\geq k} var(V)\ \Psi_t(V)$, where $\Psi_t(V)$ is defined inductively on the sintax of t.

- $\Psi_0(V) = false$ and $\Psi_1(V) = true$;
- $\Psi_X(V) = share(V,X)$, with $X \in \mathcal{V}$;
- $\Psi_{t_1 \cap t_2}(V) = \Psi_{t_1}(V) \wedge \Psi_{t_2}(V)$;
- $\Psi_{t_1 \cup t_2}(V) = \Psi_{t_1}(V) \vee \Psi_{t_2}(V)$;
- $\Psi_{\neg t}(V) = \neg \Psi_t(V)$.

∎

A simple cardinality constraint $\alpha_k(t)$ asserts the membership of at least k elements to the combination of variables in t, seen as subsets of $\mathcal{V}$. Often the term $\neg t$ will be written as $\overline{t}$. We will use the formula $\alpha_{=k}(t)$, that is t contains exactly k elements, as an abbreviation for the formula $\alpha_k(t) \wedge \neg\alpha_{k+1}(t)$. Conjunctions of cardinality constraints, are also called *cardinality constraints*.

Example 4.1 Consider the simple cardinality constraint $\alpha_2(X \cap Y \cap \overline{Z})$, that is, the formula $\exists_{\geq 2} var(V)\ share(V,X) \wedge share(V,Y) \wedge \neg share(V,Z)$. This formula is true in $\mathcal{H}$ under the state σ, if there exist at least two variables sharing with $\sigma(X)$ and $\sigma(Y)$ and not with $\sigma(Z)$, that is if the cardinality of $Vars(\sigma(X)) \cap Vars(\sigma(Y)) \cap \overline{Vars(\sigma(Z))}$, is at least equal to 2. ∎

The following lemma allows us to work with simple cardinality constraints just as if they were assertions on set of variables.

Lemma 4.1 *Let $\sigma : \mathcal{V} \to Terms(\Sigma, \mathcal{V})$. Let t^*_σ be obtained by t by replacing each occurence of variable X by $Vars(\sigma(X))$, for each $X \in \mathcal{V}$. Then $\models_\sigma \alpha_k(t)$ if and only if the cardinality of t^*_σ is at least equal to k.*

Let the elimination set Ω be composed by the atomic formulas $var(X), p_{T_1}(X), \ldots, p_{T_n}(X)$, where $X \in \mathcal{V}$, and by the set of simple cardinality costraints $\{\alpha_k(t) \mid k \geq 1, t \text{ is a boolean term}\}$. The idea is to exploit the striking similarity of the simple cardinality constraints in our language with formulas of the first-order theory of the powerset of $\mathcal{V}$ seen as a Boolean algebra. For such a theory the decidability has been shown by Skolem in 1919 just through an argument based on elimination of quantifier (see [14] for a slightly more general account). The main idea is to reduce satisfiability of a formula in $\mathcal{L}$ to satisfiability of conjunctions of simple cardinality constraints (also called *cardinality constraints*). For them the elimination of quantifiers can be carried on for cardinality constraints in a long but straightforward way.

Theorem 4.2 *Let $\psi(X)$ be a cardinality constraint. Then $\exists X\ \psi(X)$ is equivalent to a disjunction of cardinality constraints.*

Let us see how we can eliminate formulas different from cardinality constraints under existential quantifiers. A formula is *flat* if it does not contain any functor. Notice that each formula $\alpha_k(t)$ is flat. Indeed we can consider only flat formulas since

- $var(f(X)) \Leftrightarrow false$ for each functor $f \in \Sigma$;
- $share(t_1, \ldots, f(s_1, \ldots, s_k), \ldots, t_n) \Leftrightarrow \bigvee_{i=1}^{k} share(t_1, \ldots, s_i, \ldots, t_n)$;
- $p_T(f(s_1, \ldots, s_n)) \Leftrightarrow p_{T_1}(s_1) \wedge \cdots \wedge p_{T_n}(s_n)$ for each $f(T_1, \ldots, T_k) \in Def_G(T)$ (remember that G is discriminative).
- $p_T(f(s_1, \ldots, s_n)) \Leftrightarrow false$ if $f(T_1, \ldots, T_k) \notin Def_G(T)$ for any $T_1, \ldots, T_k$.

Type formulas, that is, conjunction of atomic flat formulas $p_T(X)$, possibly negated, can be eliminated under existential quantifier and replaced by cardinality constraints.

Lemma 4.3 *Let $\psi(X)$ be a cardinality constraint and $\phi(X) = p_{T_1}(X) \wedge \cdots \wedge p_{T_n}(X) \wedge \neg p_{S_1}(X) \wedge \cdots \wedge \neg p_{S_k}(X)$ a type formula. Let $S \cup [k, \omega]$ be the var-cardinality of $(T_1 \cap \cdots \cap T_n)/S_1/\cdots/S_k$. Then the following are valid equivalences in $\mathcal{H}$.*

- $(\exists X\ \phi(X) \wedge \psi(X)) \Leftrightarrow$ *false if* S *is empty and* $k = \omega$*;*
- $(\exists X\ \phi(X) \wedge \psi(X)) \Leftrightarrow (\bigvee_{h \in S} \exists X\ \alpha_{=h}(X) \wedge \psi(X)) \vee (\exists X\ \alpha_k(X) \wedge \psi(X))$.

This can also be done for $var(X)$ and for $\neg var(X)$.

Lemma 4.4 *Let* $\psi(X)$ *be a cardinality constraint. Then* $(\exists X\ \neg var(X) \wedge \psi(X)) \Leftrightarrow (\exists X\ \psi(X))$ *and* $(\exists X\ var(X) \wedge \psi(X)) \Leftrightarrow (\exists X\ \alpha_{=1}(X) \wedge \psi(X))$.

There follows then the following theorem, which is at the base of the procedure of elimination of quantifiers.

Theorem 4.5 *For each formula* $\Psi(X)$*, conjunction of formulas of* Ω*, possibly negated, there exists* Φ*, a boolean composition of formulas of* Ω*, such that* $\models (\exists X\ \Psi(X)) \Leftrightarrow \Phi$.

At this point it is easy to show that for each first-order formula of $\mathcal{L}$ it can be computed a Boolean composition of formulas of Ω equivalent to it on $\mathcal{H}$.

Theorem 4.6 *Every formula* Ψ *of* $\mathcal{L}$ *is equivalent in* $\mathcal{H}$ *to a Boolean composition of formulas of* Ω.

Finally the following theorem gives a constructive proof of decidability of formulas of Ω.

Theorem 4.7 *There exists an algorithm which decides the satisfiability of formulas* $\bigwedge \varphi_i$*, where each* φ_i *is a formula of* Ω*, possibly negated.*

Now for Ψ a general combination of formulas of Ω, we have that $\forall\Psi \Leftrightarrow \forall \bigwedge \bigvee \varphi_i \Leftrightarrow \neg\exists \bigvee \bigwedge \neg\varphi_i \Leftrightarrow \neg \bigvee \exists(\bigwedge \neg\varphi_i)$, with each φ_i a formula of Ω, possibly negated. Then $\forall\Psi$ is true if and only if none of the conjunction $\bigwedge \neg\varphi_i$ is satisfiable. Notice that if this is not the case then a conjunction $\bigwedge \neg\varphi_i$ is satisfiable and , a counterexample can be provided.

5 Applications

$\mathcal{L}$ can be employed as an assertional language to be used in inductive proof methods. Indeed the verification conditions of many such methods can be entirely expressed by a formula of $\mathcal{L}$ and effectively be decided.

Assumed $\langle \Sigma, \Lambda, \mathcal{V} \rangle$ as a signature for logic programs, with Λ the set of predicate symbols, an assertion Θ of $\mathcal{L}$ is a *specification* for predicate $p^{(n)} \in \Lambda$, if $Vars(\Theta) \subseteq \{X_1, \ldots, X_n\}$. Informally variable X_i refers to the i-th argument of p. An atom $p(t_1, \ldots, t_n)$ *satisfies the assertion* Θ, written $p(t_1, \ldots, t_n) \models \Theta$, iff $\mathcal{H} \models_{\sigma[X_1,\ldots,X_n \backslash t_1,\ldots,t_n]} \Theta$. The notation $\Theta\,[\mathbf{X}\backslash\mathbf{t}]$ denotes the formula Θ in which the variables $(X_1, \ldots, X_n)$ are contemporary substituted by terms $(t_1, \ldots, t_n)$.

Let us consider first the correct answers of a program and associate a specification Θ_p to each predicate $p \in \Lambda$. Program P in *success-correct* with respect to $\{\Theta_p\}_{p \in Pred}$ iff $\forall p(t) \in Atoms\, p(t) \overset{\theta}{\rightsquigarrow} \Box$ implies $p(t)\theta \models \Theta_p$. A sufficient condition for correctness can be stated as follows. A program P in success-correct with respect to $\{\Theta_p\}_{p \in Pred}$ if for each clause $p(\mathbf{t}) \leftarrow p_1(\mathbf{t_1}), \ldots, p_n(\mathbf{t}_n)$ it is true that

$$\mathcal{H} \models \bigwedge_{i=1}^{n} \Theta_{p_i}\,[\mathbf{X}_i \backslash \mathbf{t}_i] \Rightarrow \Theta_p\,[\mathbf{X}\backslash\mathbf{t}]\,. \tag{1}$$

The method indeed has been proposed in [5] and [11]. In our case, condition (1) can be decided using the procedure of Section (4). If the formula (1) is proved to be true for each clause, then the program is partial correct. Obviously if the formula is false this does not imply that the clause is necessarily wrong. Anyway it could be considered a warning that something wrong can happen. It is possible to provide counterexamples in such cases. The user then would have more information to decide whether the warning can give raise to a real error or simply the specification is too loose and behaviours are considered that can never occur in practice.

If we are interested to check the Input/Output behaviour of logic programs, we can associate to each predicate $p \in \Lambda$ a property $pre_p \rightarrowtriangle post_p$, where pre_p and $post_p$ are specifications for p. Program P is *I/O-correct* with respect the properties $\{pre_p \rightarrowtriangle post_p\}_{p \in Pred}$ if

$$p(t) \models pre_p \text{ and } p(t) \overset{\theta}{\rightsquigarrow} \Box \text{ implies } p(t)\theta \models post_p.$$

If each formula pre_p is monotone, a sufficient condition for P to be I/O-correct with respect the properties $\{pre_p \rightarrowtriangle post_p\}_{p \in Pred}$ is that

$$\mathcal{H} \models (\bigwedge_{i=1}^{n} (pre_{p_i}[\mathbf{X}_i \backslash \mathbf{t}_i] \Rightarrow post_{p_i}[\mathbf{X}_i \backslash \mathbf{t}_i]) \wedge pre_p[\mathbf{X} \backslash \mathbf{t}]) \Rightarrow post_p[\mathbf{X} \backslash \mathbf{t}].$$

Finally, if we want to check the call correctness of predicates we can consider methods like those proposed in [12][1]. To each predicate $p \in \Lambda$, is associated a property $pre_p \rightarrowtriangle post_p$, with pre_p and $post_p$ specifications for p. In this case, anyway, the pre-condition is used also as a specification for the argument of a predicate at call-time. In fact a program P is *call-correct* with respect the properties $\{pre_p \rightarrowtriangle post_p\}_{p \in Pred}$ if

$$p(t) \models pre_p \text{ and } p(t) \overset{\theta}{\rightsquigarrow} \Box \text{ implies } p(t)\theta \models post_p$$

and

$$p(t) \models pre_p \text{ and } p(t) \overset{*}{\rightarrow} \langle q(s), \mathbf{G} \rangle \text{ implies } q(s) \models pre_q.$$

We are assuming a leftmost selection rule for *SLD*-derivations. In [1] it has been shown that, in the case pre_p and $post_p$ are monotone for each p, then a sufficient condition for P to be correct with respect to $\{pre_p \rightarrowtriangle post_p\}_{p \in Pred}$ is that for each clause $p(\mathbf{t}) \leftarrow p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n)$, it is true that for each $1 \leq k \leq n+1$

$$\mathcal{H} \models (pre_p[\mathbf{X} \backslash \mathbf{t}] \wedge \bigwedge_{i=1}^{k-1} post_{p_i}[\mathbf{X}_i \backslash \mathbf{t}_i] \Rightarrow pre_{p_k}[\mathbf{X}_k \backslash \mathbf{t}_k])$$

where $pre_{p_{n+1}}[\mathbf{X}_{n+1} \backslash \mathbf{t}_{n+1}] \equiv post_p[\mathbf{X} \backslash \mathbf{t}]$.
A warning in this case can be raised because there may be a computation that calls a predicate with arguments which violate the specification. Again if a counterexample is provided the user may decide whether the specification is too loose or an actual error has been discovered.

Notice that previous methods are restricted to monotone properties. The reason for not considering all the expressible properties is that in those cases the verification condititions become much more complex and the mgu's have to be considered explicitly (see [15]). Indeed, the verification conditions are no more expressible as formulas of $\mathcal{L}$. Anyway the class of properties that can be mechanically checked is still quite large, including type assertions, groundness, dependencies.

6 Conclusions

In this paper we have studied a language that allows to express and decide aliasing properties such as the sharing or freeness and is enriched with type assertions. Using formulas of $\mathcal{L}$, we may prove many properties of programs in inductive proof methods. Morover since the logic is decidable, the corresponding verification conditions can be mechanically checked and a warning can be raised if a condition is false.
The set of true formulas is proved to be decidable through a procedure of elimination of quantifiers. This points out an interesting class of formulas, which express cardinality constraints on the set of variables that can occur in a term. This can give an interesting insight on domains used for the analysis of logic program. In fact domains such as $\mathcal{POS}$ [8] and *Sharing* [13], used for aliasing analysis, can be seen as fragments of the domain of formulas of cardinality constraints.

We are currently investigating two possibilities for augmenting the expressive power of the logic $\mathcal{L}$, obviously while retaining the decidability.
The first consists in adding a modal operator $\Box$ defined as follows

$$\models_\sigma \Box\varphi \quad \text{iff for each } \sigma' \geq \sigma \models_{\sigma'} \varphi.$$

Such modality would allow, first of all, to characterize monotone properties of language $\mathcal{L}$, which would correspond to the formulas Ψ such that $\Psi \Leftrightarrow \Box\Psi$. Morover we could express dependences between properties, like $\Box(list(X) \Rightarrow list(Y))$, whose informal meaning is that every state that instantiate X to a *list* will also bind Y to a *list*.
Another extension is given by Hoare-like triples $\{\Phi\}[\![t, s]\!]\{\Psi\}$, already considered in [6] and [7], where a formal calculus has been provided for a language of assertions slightly different from $\mathcal{L}$. Their meaning is: if Φ is true under the state σ and $\theta = mgu(\sigma(t), \sigma(s))$, then Ψ is true under the state $\sigma \circ \theta$. These formulas would allow to express the verification of general proof methods, like those in [12] and in [15], for the whole class of formulas of $\mathcal{L}$. At moment, it is still not known if, given a decidable logic such as $\mathcal{L}$, it is possible to decide the validity of such triples.

References

[1] A. Bossi and N. Cocco. Verifying Correctness of Logic Programs. In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89*, pages 96–110, 1989.

[2] J. Boye. *Directional Types in Logic Programming.* PhD thesis, University of Linköping, Computer Science Department, 1997.

[3] J. Boye and J. Maluszynski. Directional Types and the Annotation Method. *Journal of Logic Programming*, 33(3):179–220, 1997.

[4] C. C. Chang and H. J. Kreisler. *Model Theory.* Elsevier Science Publ., 1990. Third edition.

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

[6] L. Colussi and E. Marchiori. Proving Correctness of Logic Programs Using Axiomatic Semantics. In *Proc. of the Eight International Conference on Logic Programming*, pages 629–644. The MIT Press, Cambridge, Mass., 1991.

[7] L. Colussi and E. Marchiori. Unification as Predicate Transformer. In *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 67–85. The MIT Press, Cambridge, Mass., 1992.

[8] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.

[9] P. Dart and J. Zobel. Efficient run-time type checking of typed logic program. *Journal of Logic Programming*, 14(1-2):31–70, 1992.

[10] P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in logic programming*, pages 157–187. The MIT Press, Cambridge, Mass., 1992.

[11] P. Deransart. Proof Methods of Declarative Properties of Definite Programs. *Theoretical Computer Science*, 118(2):99–166, 1993.

[12] W. Drabent and J. Maluszynski. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

[13] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming'89*, pages 154–165. The MIT Press, Cambridge, Mass., 1989.

[14] G. Kreisel and J. L. Krivine. *Elements of Mathematical Logic (Model Theory)*. North-Holland, Amsterdam, 1967.

[15] G. Levi and P. Volpe. Derivation of Proof Methods by Abstract Interpretation. (Submitted). Available at http://www.di.unipi.it/∼volpep/papers.html, 1998.

[16] 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)*, number 850 in LNCS, pages 287–304. Springer Verlag, 1994.

[17] 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)*, number 1139 in LNCS, pages 209–223. Springer Verlag, 1996.