A Logic for Variable Aliasing in Logic Programs

E. Marchiori

Computer Science/Department of Software Technology

CS-R9446 1994

A Logic for Variable Aliasing in Logic Programs

Elena Marchiori
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
`e-mail:elena@cwi.nl`

Abstract

This paper introduces a logic for a class of properties - in particular variable aliasing - used in static analysis of logic programs. The logic is shown to be sound, complete and decidable. Moreover, it is illustrated how this logic can be applied to automatize some parts of the reasoning when proving the partial correctness of a logic program.

1991 Mathematics Subject Classification: 68Q40, 68Q60, 68T15.
1991 CR Categories: F.3.1, F.4.1, I.2.3.
Keywords and Phrases: Properties of substitutions, static analysis of logic programs.

Notes: This research was partly supported by Esprit BRA 6810 (Compulog 2). This paper will appear (in a sligthly different version) in: Proceedings of the Fourth International Conference on Algebraic and Logic Programming.

1 Introduction

A number of properties of substitutions have been identified as crucial when analyzing the run-time behaviour of logic programs. They involve groundness and aliasing: for a substitution α, a variable x is said to be ground if $x\alpha$ does not contain variables; x and y are said to share, or to be aliasing if $x\alpha$ and $y\alpha$ have at least a variable in common. These properties are relevant in static analysis of logic programs. For instance, detection of groundness of certain variables of the program at run-time allows to improve efficiency, by using matching instead of unification. Also, if the arguments of two atoms at run-time do not share any variable, then they may be executed in parallel.

Various assertional methods to prove the correctness and termination of a logic program incorporate these properties in the assertion language ([DM88], [CM91]; see [AM94] for an overview and comparison of various assertional methods). These properties play an even more fundamental role in abstract interpretation of logic programs, where they are used to compute approximations of the set of all possible substitutions which can occur at each step of the execution of the program. The abstract interpretation approach, developed in [CC77] for data-flow analysis of imperative programs, has been successfully applied to logic programs (see [AH87] for a brief introduction to the major stages in the development of the field; see [CC92] for a survey on its applications to logic programs). Since both the problems of groundness and of sharing among program variables at run-time is undecidable, it remains a hard problem to find an abstract interpretation framework for the study of aliasing that is efficient and that provides an accurate analysis.

We introduce a logic where the relation symbols *var*, *ground* and *share* are used to express the basic properties we intend to study and the logical operators $\wedge$ and $\neg$ are used to express composite properties. Then the semantics of the resulting assertions consists of a set of substitutions, where

$\wedge$ and $\neg$ are interpreted as set-theoretic intersection and complementation; the atoms $var(t)$ and $ground(t)$ are interpreted as the set of substitutions which map the term t to a variable and a ground term, respectively; finally the semantics of $share(t_1, \ldots, t_n)$ is the set of substitutions which map the terms $t_1, \ldots, t_n$ to terms sharing some variable. A system of inference rules (used as rewrite rules) is introduced which allows the definition of a terminating procedure which decides truth (hence satisfiability) of assertions in the logic. As an example, we illustrate how this procedure can be applied to mechanize some parts of the reasoning when proving the partial correctness of a logic program.

In [CM92] unification in logic programming is characterized by means of a predicate transformer, where also the assertions of our logic are considered. Moreover, a number of rules occurring in the present paper (viz. the singleton rules of Table 1) are there implicitly used to simplify the form of an assertion. However, the problem of finding a complete axiomatization of these properties is not investigated.

A formalization of groundness by means of a propositional logic has been given in [MS89]. The propositional logic is used as an abstract domain, to analyze variable groundness in logic programs. That logic has further been studied in [CFW91]. However, to the best of our knowledge our contribution is the first rigorous study of those properties of substitutions expressed by *groundness*, *var* and *aliasing* together with their relationship.

2 A Logic for Properties of Substitutions

Syntax

We shall consider terms containing variables. Formally, consider a countable set Var of *variables*. Let Fun be a set of *functors with rank*, containing a set $Const$ of *constants* consisting of the functors with rank zero. The class $Term$ of *terms* is the smallest set T containing $Const \cup Var$ and with the property that if $t_1, \ldots, t_n$ are in T and $f \in Fun$ has rank n then $f(t_1, \ldots, t_n)$ is in T. Then a substitution σ is a map from Var to $Term$ such that its domain $dom(\sigma) = \{x \in Var \mid x\sigma \neq x\}$ is finite. The definition of substitution is extended in the standard way to terms in $Term$, where for a substitution σ and a term t the term $t\sigma$ is obtained by simultaneously replacing every variable x of t by the term $x\sigma$. Moreover for a set S of terms and for a substitution σ we denote by $S\sigma$ the set $\{t\sigma \mid t \in S\}$. The set of substitutions is denoted by $Subst$.

For a syntactic expression o, $Var(o)$ denotes the set of variables occurring in o. Variables are denoted by v, x, y, z. Functors are indicated by f, g and constants by a, b, c. Terms are denoted by the letters r, s, t. The capital letter S is used to denote a finite set of terms, while $|S|$ indicates the cardinality of S.

Properties are expressed by means of formulas called assertions.

Definition 2.1 (Assertions) The set $\mathcal{A}$ of *assertions* is the smallest set A of formulas containing the atoms $var(t)$, $ground(t)$ for all terms t in $Term$, and $share(S)$ for all sets S of terms in $Term$, and with the property that if ϕ is in A then $\neg\phi$ is in A, and if ϕ and ψ are in A then $\phi \wedge \psi$ is in A.

The notation $\phi \vee \psi$ is used as a shorthand for $\neg(\neg\phi \wedge \neg\psi)$. Atoms and their negation form the class of *literals*, where a literal is denoted by L.

Semantics

An assertion ϕ is interpreted as a set $[\![\phi]\!]$ of substitutions. Logical connectives are interpreted set-theoretically in such a way that set intersection and union correspond to $\wedge$ and $\vee$, respectively, while complementation (w.r.t. *Subst*) corresponds to $\neg$. Atoms are interpreted as follows: $var(t)$ is the set of substitutions which map t to a variable, $ground(t)$ is the set of substitutions which map t to a term containing no variables, and $share(\{t_1, \ldots, t_n\})$ is the set of substitutions which map $t_1, \ldots, t_n$ to terms containing at least one common variable.

Definition 2.2 (Semantics)

$[\![var(t)]\!] = \{\sigma \in Subst \mid t\sigma \in Var\}$;

$[\![ground(t)]\!] = \{\sigma \in Subst \mid Var(t\sigma) = \emptyset\}$;

$[\![share(\{s_1, \ldots, s_n\})]\!] = \{\sigma \in Subst \mid \bigcap_{i=1}^{n} Var(s_i\sigma) \neq \emptyset\}$;

$[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$;

$[\![\neg\phi]\!] = Subst - [\![\phi]\!]$.

□

If $[\![\phi]\!] = Subst$ then ϕ is said to be true; if there exists σ s.t. $\sigma \in [\![\phi]\!]$ then ϕ is said to be satisfiable. Two assertions ϕ and ψ are said to be equivalent if $[\![\phi]\!] = [\![\psi]\!]$. Notice that $share(\{t\})$ is equivalent to $\neg ground(t)$. Therefore we will assume in the following that only atoms of the form $share(S)$, with $|S| \geq 2$ occur in an assertion. Moreover it is convenient to introduce the propositional constants *true* and *false* where $[\![true]\!] = Subst$ and $[\![false]\!] = \emptyset$.

Assertions satisfy the classical replacement theorem.

Theorem 2.3 *Let ψ be a sub-assertion of an assertion ϕ. Suppose that ψ is equivalent to ψ'. Let ϕ' be the assertion obtained replacing zero or more occurrences of ψ in ϕ by the assertion ψ'. Then ϕ is equivalent to ϕ'.*

Proof. Easy, by induction on the number of connectives occurring in ϕ. □

3 Axiomatization

In this section, a system of axioms and inference rules is introduced, where all the rules are of a particular simple form $\frac{\phi}{\psi}$, where ϕ and ψ are assertions in $\mathcal{A}$. The meaning of a rule is that ϕ and ψ are equivalent. Equivalence is required because rules will be used as rewrite rules: ψ will be replaced by ϕ. We shall apply then rules also to formulas that occur as subformulas of a larger formula. This will still preserve equivalence because of Theorem 2.3. For instance, the application of the rule $\frac{\phi}{\psi}$ to the formula $\psi \vee \neg\phi$ produces the formula $\phi \vee \neg\phi$.

The system is used to define, in the following section, a terminating procedure which reduces an assertion ϕ to *true* if and only if ϕ is true.
The following collection of *general rules* will be used to simplify the form of assertions.

G1 $true$ **G2** $\neg\, false$ **G3** $\phi \vee \neg\phi$ **G4** $\phi \vee true$

G5 $\dfrac{\phi \vee \psi}{\psi \vee \phi}$ **G6** $\dfrac{\phi}{\phi \vee false}$ **G7** $\dfrac{\phi}{\phi \wedge true}$ **G8** $\dfrac{\phi}{\phi \vee \phi}$

We consider two other collections of rules, given in Tables 1 and 2: the *singleton rules* which describe the semantics of an atom by investigating the structure of its arguments and the *combination rules* which describe the semantics of disjunctions of literals.

Notice that, in the singleton rules, k is greater or equal than 0. Moreover if $k = 0$ then $\bigvee_{i\in[1,k]} \phi_i$ and $\bigwedge_{i\in[1,k]} \phi_i$ should be read as *false* and *true*, respectively. Moreover, in the combination rules S, S_1 and S_2 denote sets of variables.

S1 $\dfrac{false}{var(f(s_1,\ldots,s_k))}$

S2 $\dfrac{\bigwedge_{i\in[1,k]} ground(s_i)}{ground(f(s_1,\ldots,s_k))}$

S3 $\dfrac{\bigvee_{i\in[1,k]} share(S\cup\{s_i\})}{share(S\cup\{f(s_1,\ldots,s_k)\})}$

Table 1: Singleton Rules

C1 $\neg ground(x) \vee \neg var(x)$

C2 $\dfrac{\neg var(x)}{ground(x) \vee \neg var(x)}$

C3 $\dfrac{\neg ground(x)}{\neg ground(x) \vee var(x)}$

C4 $\dfrac{\neg ground(x)}{\neg ground(x) \vee share(S\cup\{x\})}$

C5 $\neg ground(x) \vee \neg share(S\cup\{x\})$

C6 $\dfrac{\neg share(S\cup\{x\})}{ground(x) \vee \neg share(S\cup\{x\})}$

C7 $\dfrac{\neg var(x) \vee \neg share(S_1\cup S_2\cup\{x\})}{\neg var(x) \vee \neg share(S_1\cup\{x\}) \vee \neg share(S_2\cup\{x\})}$

C8 $share(S_1) \vee \neg share(S_1\cup S_2)$

Table 2: Combination Rules

Theorem 3.1 *General rules, singleton rules and combination rules are equivalences.*

Proof. For the general rules the result follows direct from Definition 2.2. For a rule $\frac{\phi}{\psi}$ we have to show that a substitution is in $[\![\phi]\!]$ if and only if it is in $[\![\psi]\!]$; for an axiom ϕ we have to show that every substitution is in $[\![\phi]\!]$. Let α be an arbitrary substitution. Notice that

$$Var(f(s_1,\ldots,s_k)\alpha) = \bigcup_{i=1}^{k} Var(s_i\alpha). \tag{1}$$

S1: $f(s_1,\ldots,s_k)\alpha$ is not in Var.

S2: From (1) it follows that $Var(f(s_1,\ldots,s_k)\alpha) = \emptyset$ if and only if $Var(s_i\alpha) = \emptyset$ for $i \in [1,k]$.

S3: From (1) it follows that $\bigcap_{s\in S} Var(s\alpha) \cap Var(f(s_1,\ldots,s_k)\alpha) \neq \emptyset$ if and only if $\bigcap_{s\in S} Var(s\alpha) \cap Var(s_i) \neq \emptyset$ for some $i \in [1,k]$.

C2: $\alpha \in [\![ground(x)]\!]$ implies $Var(x\alpha) = \emptyset$ which implies $\alpha \in [\![\neg var(x)]\!]$.

C6: $\alpha \in [\![ground(x)]\!]$ implies $Var(x\alpha) = \emptyset$ which implies $\alpha \in [\![\neg share(S \cup \{x\})]\!]$.

C7: If $x\alpha \notin Var$ then the result follows immediate; if $x\alpha \in Var$ then $Var(x\alpha) \cap \bigcap_{y\in S_1} Var(y\alpha) \cap \bigcap_{z\in S_2} Var(z\alpha) = \emptyset$ if and only if $x\alpha \notin \bigcap_{y\in S_1} Var(y\alpha)$ or $x\alpha \notin \bigcap_{z\in S_2} Var(z\alpha)$.

C8: If $\bigcap_{y\in S_1} Var(y\alpha) \neq \emptyset$ then $\alpha \in [\![share(S_1)]\!]$; if $\bigcap_{y\in S_1} Var(y\alpha) = \emptyset$ then $\bigcap_{y\in S_1\cup S_2} Var(y\alpha) = \emptyset$ which implies $\alpha \in [\![\neg share(S_1 \cup S_2)]\!]$.

Moreover it is easy to check that rules **C1** and **C3** can be derived from rule **C2** by straightforward set operations. Analogously, rules **C4** and **C5** can be derived from rule **C6**. These rules are useful in the following section.

□

4 Soundness, Completeness and Decidability of the Logic

The system of rules introduced in the previous section allows to define a terminating procedure which applied to an assertion ϕ yields *true* if and only if ϕ is true. For technical reasons, it is convenient to have only one axiom, namely (G1): thus every other axiom ϕ is translated into the rule $\frac{true}{\phi}$. First, the singleton rules are used to reduce ϕ to a form called flat form; next the conjunctive normal form $\phi_1 \wedge \ldots \wedge \phi_n$ is computed; finally every conjunct ϕ_i is reduced to a normal form by means of the combination rules and the general rules and the outcome *true* is given if and only if the resulting conjuncts are equal to *true*.

4.1 Flat Form and Normal Form

Definition 4.1 (Flat Form) An assertion is in *flat form* if it does not contain any functors.

For example the assertion $share(\{f(x),y\}) \wedge var(x)$ is not in flat form (because the term $f(x)$ contains a functor) while the assertion $\neg var(x) \vee (ground(x) \wedge share(\{y,z\}))$ is in flat form.
The (proof of the) following lemma provides an algorithm to transform an assertion in flat form.

The following function *size* is used to prove that the algorithm terminates: *size* maps a term s to the natural number n, and is defined as follows:

$$size(s) = \begin{cases} 1 & \text{if } s \in Var \\ 1 + \sum_{i=1}^{n} size(s_i) & \text{if } s = f(s_1,\ldots,s_n),\ n \geq 0, \end{cases}$$

where $\sum_{i=1}^{0} size(s_i)$ is assumed to be equal to 0.

Lemma 4.2 *ϕ is equivalent to an assertion in flat form.*

Proof. The flat form of ϕ is obtained by applying repeatedly the singleton rules to every atom occurring in ϕ. The process terminates because the quantity

$$m(\phi) = \begin{cases} 0 & \text{if } \phi \in \{\mathit{false}, \mathit{true}\} \\ \sum_{s \in S} \mathit{size}(s) & \text{otherwise,} \end{cases}$$

where S is the union of the arguments of the literals which occur in ϕ (thus counting multiple occurrences of terms only once; here an argument which is a term, say t, is identified with the singleton set $\{t\}$) decreases when a rule is applied to ϕ. It follows from Theorem 3.1 and Theorem 2.3 that the resulting assertion is equivalent to ϕ. □

Notice that from the proof of the previous lemma it follows that the flat form of an assertion computed using the singleton rules is unique modulo the order in which the literals occur in the assertion.

We introduce now the class of assertions in *normal form.*

Definition 4.3 (Normal Form) An assertion ϕ is in *normal form* if ϕ is in flat form and $\phi = \bigvee_{i=1}^{n} L_i$, $n \geq 1$ such that either ϕ is a propositional constant or ϕ does not contain any propositional constant, $L_i \neq L_j$ for $i \neq j$ and the following conditions hold:

(a) if $L_i = \neg \mathit{ground}(x)$ for some $i \in [1, n]$ then $x \notin \mathit{Var}(L_j)$ for every $j \neq i$;

(b) if $L_i = \mathit{ground}(x)$ for some $i \in [1, n]$ then every other literal containing x is either equal to $\mathit{var}(x)$ or it is of the form $\mathit{share}(S \cup \{x\})$;

(c) if $L_i = \neg \mathit{var}(x)$ for some $i \in [1, n]$ then every other literal containing x is of the form $*\mathit{share}(S \cup \{x\})$ and at most one of them is of the form $\neg \mathit{share}(S \cup \{x\})$ ($*$ denotes $\neg$ or a blank);

(d) if $L_i = \mathit{share}(S)$ for some $i \in [1, n]$ then for every other literal of the form $\neg\ \mathit{share}(S')$ we have that $S \not\subseteq S'$.

For example the assertion $\neg \mathit{ground}(x) \vee \mathit{var}(x)$ is not in normal form (because condition (a) of the definition is not satisfied), the assertion $\mathit{share}(\{x, y\}) \vee \neg \mathit{share}(\{x, y, z\})$ is not in normal form (because condition (d) of the definition is not satisfied) while the assertion $\neg \mathit{ground}(x) \vee \mathit{ground}(y) \vee \mathit{var}(y) \vee \mathit{share}(\{y, z\})$ is in normal form.

The (proof of the) following lemma provides an algorithm to transform into normal form any assertion in flat form consisting of a disjunction of literals.

Lemma 4.4 *Let $\phi = \bigvee_{i \in [1,n]} L_i$. Suppose that ϕ is in flat form. Then ϕ is equivalent to an assertion in normal form.*

Proof. The normal form of ϕ is obtained as follows. For every variable x contained in ϕ the disjunction of literals of ϕ containing x is considered and the combination rules are applied, using the general rules when applicable and using rule ($G5$) only a finite number of times. Notice that all the rules preserve the flat form. The result will be either a propositional constant, by application of rules (G2), (G3), (G4), (G5), (G6), (C1), (C5) and (C8); otherwise the result will not contain any propositional constant, by application of rules (G5) and (G6): moreover it will satisfy (a) by application of rules (C1), (C3), (C4), (C5) and (G3), (G5) and (G8); it will satisfy (b) by application of rules (C2), (C6) and (G3), (G5) and (G8); it will satisfy (c) by application of rules (C1), (C2), (C7) and (G3),(G5) and (G8); finally it will satisfy (d) by application of the rules (G5) and (C8).

The process terminates because by assumption rule (G5) is applied only finitely many times, and the application of every other rule decreases the number of connectives of the assertion. Finally, Theorem 3.1 and Theorem 2.3 imply that the resulting assertion is equivalent to ϕ. $\square$

Notice that from the proof of the previous lemma it follows that the normal form of an assertion consisting of a disjunction of literals, computed using the general rules and the combination rules, is unique modulo the order in which the literals occur in the assertion.

The following example illustrates the application of the axiomatization.

Example 4.5 Consider the assertion ϕ:

$$\mathit{var}(f(w)) \vee \mathit{ground}(x) \vee \neg \mathit{share}(\{x,y\}) \vee \neg \mathit{share}(\{y,z\}) \vee \neg \mathit{share}(\{z,w\}) \vee \mathit{share}(\{x,g(a,y),z\}).$$

1. Application of rule (S1) to $\mathit{var}(f(w))$ yields
   $\mathit{false} \vee \mathit{ground}(x) \vee \neg \mathit{share}(\{x,y\}) \vee \neg \mathit{share}(\{y,z\}) \vee \neg \mathit{share}(\{z,w\}) \vee \mathit{share}(\{x,g(a,y),z\})$;
2. application of rule (S3) to $\mathit{share}(\{x,g(a,y),z\})$ yields
   $\mathit{false} \vee \mathit{ground}(x) \vee \neg \mathit{share}(\{x,y\}) \vee \neg \mathit{share}(\{y,z\}) \vee \neg \mathit{share}(\{z,w\}) \vee \mathit{share}(\{x,a,z\}) \vee \mathit{share}(\{x,y,z\})$;
3. application of rule (S3) to $\mathit{share}(\{x,a,z\})$ yields
   $\mathit{false} \vee \mathit{ground}(x) \vee \neg \mathit{share}(\{x,y\}) \vee \neg \mathit{share}(\{y,z\}) \vee \neg \mathit{share}(\{z,w\}) \vee \mathit{false} \vee \mathit{share}(\{x,y,z\})$,
   which is in flat form.
4. Application of rule (G5) yields
   $\mathit{ground}(x) \vee \neg \mathit{share}(\{x,y\}) \vee \neg \mathit{share}(\{y,z\}) \vee \neg \mathit{share}(\{z,w\}) \vee \mathit{false} \vee \mathit{share}(\{x,y,z\}) \vee \mathit{false}$;
5. application of rule (G6) yields
   $\mathit{ground}(x) \vee \neg \mathit{share}(\{x,y\}) \vee \neg \mathit{share}(\{y,z\}) \vee \neg \mathit{share}(\{z,w\}) \vee \mathit{false} \vee \mathit{share}(\{x,y,z\})$;
6. application of rule (G5) yields
   $\mathit{share}(\{x,y,z\}) \vee \mathit{ground}(x) \vee \neg \mathit{share}(\{x,y\}) \vee \neg \mathit{share}(\{y,z\}) \vee \neg \mathit{share}(\{z,w\}) \vee \mathit{false}$;
7. application of rule (G6) yields
   $\mathit{share}(\{x,y,z\}) \vee \mathit{ground}(x) \vee \neg \mathit{share}(\{x,y\}) \vee \neg \mathit{share}(\{y,z\}) \vee \neg \mathit{share}(\{z,w\})$;
8. application of rule (C6) to $\mathit{ground}(x) \vee \neg \mathit{share}(\{x,y\})$ yields

   $\mathit{share}(\{x,y,z\}) \vee \neg \mathit{share}(\{x,y\}) \vee \neg \mathit{share}(\{y,z\}) \vee \neg \mathit{share}(\{z,w\})$,

   which is in normal form.

4.2 Decidability Procedure

The previous results are used to define the following decidability procedure.

Definition 4.6 (Truth Procedure) The *truth procedure* TP reduces an assertion ϕ as follows. First the flat form ϕ_1 of ϕ is computed by means of Lemma 4.2. Next ϕ_1 is transformed (using standard methods) into a conjunctive normal form $\phi_2 = \psi_1 \wedge \ldots \wedge \psi_n$, where every ψ_i is a disjunction of literals. Finally every ψ_i is reduced to normal form by means of Lemma 4.4 and rule (G7) is applied to the resulting conjunction as many times as possible. □

Thus ϕ is reduced by TP to a conjunction of assertions in normal form. We prove now that TP is correct and terminating. Let $TP(\phi)$ denote the outcome of TP applied to ϕ.

Theorem 4.7 *TP is a terminating procedure and $TP(\phi)$ is equal to true if and only if ϕ is equivalent to true.*

To prove the above statement it is necessary to assume that Fun contains a functor of rank 0 (i.e., a constant) and one of rank 2. If it is not the case, then we add such functors to the language. Moreover some preliminary results are necessary. First, an algorithm called $Prod$ is defined: given as input an assertion ψ in normal form which is neither equal to *true* nor to *false*, $Prod$ produces a substitution σ such that $\sigma \notin [\![\psi]\!]$. This σ is computed in a number of steps. After each step, the intermediate result (still called σ) is applied to the resulting formula, called $A(\psi)$. Thus, two variables are used: a variable σ which contains the part of the substitution actually computed and a variable $A(\psi)$, which contains the assertion obtained from ψ applying σ. Moreover in the algorithm we need to know which of the variables of $A(\psi)$ stem from the application of the computed σ. For instance, suppose $\psi = share(\{x, y\})$ and $\sigma = \{x/f(z)\}$: then $A(\psi) = share(\{f(z), y\})$ and z is a variable which stems from the application of σ. Then to recognize these variables we assume that they are chosen from the set $IVar = Var \setminus Var(\phi)$. Variables of $IVar$ are denoted by capital letters $U, V, \ldots$. In the remainder of this section, the variables of $IVar$ occurring in a syntactic object o are called *image variables*, denoted by $Ivar(o)$, while the other variables occurring in o are called simply *variables*, denoted by $Var(o)$. Finally some other variables are used in the algorithm: for every literal L in ψ of the form $\neg share(S)$ for some S, a variable im_L is introduced which either is equal to a image variable or is undefined. The role of these variables will be explained afterwards. Initially im_L is undefined, and once im_L is set to a particular image variable, it will never change. For a image variable U the notation $U = im_L$ means that im_L is defined and that U is equal to (the value of) im_L.

The algorithm $Prod$ is now defined as follows. Let g be a functor of rank 2 and let a be a constant. Let $g_1(t)$ denote the term $g(t, t)$ and for $n \geq 2$ let $g_n(t_1, \ldots, t_n)$ denote the term $g(t_1, g(t_2, \ldots, g(t_{n-1}, t_n) \ldots))$.

Initially $A(\psi)$ is set to ψ and σ is set to ϵ, the empty substitution. The algorithm consists of the following sequence of three steps.

1 For every variable x occurring in ψ, perform the following sequence of actions:

- **1.1** If the antecedent of (a) holds then set σ to $\sigma \cup \{x/a\}$;
- **1.2** If the antecedent of (b) holds then set σ to $\sigma \cup \{x/g_1(U)\}$, where U is a fresh image variable (i.e. an image variable not yet used);
- **1.3** If the antecedent of (c) holds then set σ to $\sigma \cup \{x/U\}$, where:
  - **1.3.1** if a literal L of the form $\neg share(S \cup \{x\})$ occurs in ψ then either $U = im_L$ or, if im_L is undefined, U is chosen to be a fresh image variable and im_L is set to U;

1.3.2 otherwise (i.e., if no literal of the form $\neg share(S \cup \{x\})$ occurs in ψ) U is a fresh image variable;

1.4 set $A(\psi)$ to $A(\psi)\sigma$.

2 For every variable x occurring in $A(\psi)$, perform the following sequence of actions:

2.1 If $L_1\sigma, \ldots, L_m\sigma$ are all the disjuncts of $A(\psi)$ of the form $\neg share(S \cup \{x\})$, with $m \geq 1$, then set σ to $\sigma \cup \{x/g_m(V_1, \ldots, V_m)\}$, where $V_1, \ldots, V_m$ are distinct image variables such that: either $V_i = im_{L_i}$ or, if im_{L_i} is undefined, V_i is chosen to be a fresh image variable and im_{L_i} is set to V_i.

2.2 Set $A(\psi)$ to $A(\psi)\sigma$.

3 For every variable x occurring in $A(\psi)$ set σ to $\sigma \cup \{x/a\}$. Set $A(\psi)$ to $A(\psi)\sigma$.

□

Some explanation of the steps of the algorithm is needed: as already said, the aim of *Prod*, when applied to an assertion ψ in normal form which is not a propositional constant, is to produce a substitution σ which is not in the semantics $[\![\psi]\!]$ of ψ. Such substitution is built incrementally, by binding each variable of ψ to a suitable term. The first three subcases of step 1 are mutually exclusive, and correspond to the first three cases in the definition of normal form. Thus after step 1 is executed, literals of the form $\neg ground(x)\sigma$, $ground(x)\sigma$, and $\neg var(x)\sigma$ become false. Moreover the variables which are not yet bound by σ occur either in literals of the form $\neg share(S)$, or of the form $share(S)$ or of the form $var(x)$. Step 2 of *Prod* takes care of all the literals of the form $\neg share(S)$: the variables of S are mapped by the substitution to terms having exactly one image variable in common. Finally step 3 of *Prod* takes care of all the literals of the form $var(x)\sigma$ or $share(S)\sigma$ which contain some variable.

To avoid that in step 2 the variables of some literal of the form $share(S')$ become bound to terms having some common image variable, it is sufficient (as will be proven in Lemma 4.12) that the image variables which are shared by the terms of distinct literals of the form $\neg share(S)$, be distinct. This is obtained by means of the variables im_L, which fix once for all the image variable which will be shared eventually by all the terms of L.

We illustrate now the application of *Prod* with an example.

Example 4.8 Let ψ be the formula obtained in Example 4.5:

$$share(\{x, y, z\}) \vee \neg share(\{x, y\}) \vee \neg share(\{y, z\}) \vee \neg share(\{z, w\}).$$

Since ψ is in normal form, we can apply *Prod*. Let L_1 denote $\neg share(\{x, y\})$, let L_2 denote $\neg share(\{y, z\})$ and let L_3 denote $\neg share(\{z, w\})$. The values of the variables of *Prod* corresponding to one possible execution are given below, where only the initial and the final value of $A(\psi)$ are shown:

1. Initialization:

   $A(\psi) = \psi$, $\sigma = \epsilon$, im_{L_i} undefined for $i \in [1, 3]$;

2. Step 2, suppose *Prod* has chosen the variable y:

   $\sigma = \{y/g(V_1, V_2)\}$, $im_{L_1} = V_1$, $im_{L_2} = V_2$, im_{L_3} undefined;

3. Step 2, suppose *Prod* has chosen the variable x:

   $\sigma = \{y/g(V_1, V_2), x/g(V_1, V_1)\}$, $im_{L_1} = V_1$, $im_{L_2} = V_2$, im_{L_3} undefined;

4. Step 2, suppose *Prod* has chosen the variable z:
   $\sigma = \{y/g(V_1, V_2), x/g(V_1, V_1), z/g(V_2, V_3)\}$,
   $im_{L_1} = V_1$, $im_{L_2} = V_2$, $im_{L_3} = V_3$;
5. Step 2, suppose *Prod* has chosen the variable w:
   $\sigma = \{y/g(V_1, V_2), x/g(V_1, V_1), z/g(V_2, V_3), w/g(V_3, V_3)\}$,
   $im_{L_1} = V_1$, $im_{L_2} = V_2$, $im_{L_3} = V_3$;
6. stop (all the variables of ψ have been considered):
   $A(\psi) = \neg share(\{g(V_1, V_1), g(V_1, V_2)\}) \vee \neg share(\{g(V_1, V_2), g(V_2, V_3)\}) \vee$
   $share(\{g(V_1, V_1), g(V_1, V_2), g(V_2, V_3)\}) \vee \neg share(\{g(V_2, V_3), g(V_3, V_3)\})$.

Notice that *Prod* terminates because the number of variables occurring in a formula ψ is finite. Moreover σ is well-defined because the first three cases of step 1 are mutually exclusive and variables of type im_L are distinct, as Lemma 4.9 will show. To show that *Prod* is correct (i.e., that if *Prod* is applied to ψ then the produced substitution σ is not in $[\![\psi]\!]$), we need some preliminary results. The following lemma states a crucial property of the variables of type im_L.

Lemma 4.9 *Let im_L and $im_{L'}$ be two distinct variables of Prod. If im_L and $im_{L'}$ are defined then they are equal to two distinct image variables.*

Proof. Notice that im_L is initially undefined and it becomes defined only when it is bound by *Prod* to a fresh image variable. □

In the following lemma a property is proven to be invariant under the execution of *Prod*. Notice that σ is considered as a variable of the algorithm and that at every step of the algorithm, $A(\psi)$ is equal to $\psi\sigma$, for a suitable *value* of σ. Therefore in the following a literal of $A(\psi)$ is sometimes denoted by $L\sigma$, where L is the corresponding literal of ψ and σ is the actual value of the computed substitution.

Lemma 4.10 *If $x \in dom(\sigma)$ and x occurs in m disjuncts of ψ of the form $\neg share(S)$, for some $m \geq 1$, then*

$$x\sigma = \begin{cases} im_{L_1} & \textit{if } m = 1 \textit{ and the antecedent of (c) holds,} \\ g_m(im_{L_1}, \ldots, im_{L_m}) & \textit{if } m \geq 1 \textit{ and the antecedent of (c) does not hold,} \end{cases}$$

where L_1, ..., L_m are all the disjuncts of ψ of the form $\neg share(S)$ such that $x \in S$.

Proof. Initially *Prod* satisfies trivially the property because $\sigma = \epsilon$. Step 1 preserves the property: for every variable x considered in that step, if the first or second subcase was applied then x does not occur in disjuncts of the form $\neg share(S)$; if the third subcase was applied then if im_L was undefined then x is bound to one fresh image variable and im_L is set to that image variable; otherwise (i.e., im_L defined) x is bound to im_L. Step 2 preserves the property because, for every variable x considered in that step, x is bound to a term t such that: if $m \geq 1$ then t is $g_m(V_1, \ldots, V_m)$, where for $i \in [1, m]$ if im_{L_i} was defined then V_i is equal to im_{L_i}, otherwise V_i is a fresh image variable and im_{L_i} is set to V_i. Finally step 3 preserves the property because the variables considered do not occur in disjuncts of the form $\neg share(S)$. □

Lemma 4.11 *If $S \subseteq dom(\sigma)$ is such that*

1. $S \not\subseteq S'$, for every disjunct of ψ of the form $\neg share(S')$;

2. for every x in S there exists a disjunct of ψ of the form $\neg share(S')$ such that $x \in S'$.

Then $\bigcap_{x\in S} Ivar(x\sigma) = \emptyset$.

Proof. From the hypothesis it follows that S contains at least two elements, i.e., $S = \{x_1, \ldots, x_n\}$, $n \geq 2$. Then by Lemma 4.10 we have that for $i \in [1, n]$

$$x_i\sigma = \begin{cases} im_{L_1^{x_i}} & \text{if } m_i = 1 \text{ and the antecedent of (c) holds,} \\ g_{m_i}(im_{L_1^{x_i}}, \ldots, im_{L_{m_i}^{x_i}}) & \text{if } m_i \geq 1 \text{ and the antecedent of (c) does not hold,} \end{cases}$$

where $L_1^{x_i}$, ..., $L_{m_i}^{x_i}$ are all the disjuncts of ψ of the form $\neg share(S')$ such that $x_i \in S'$. By 2 we have that $m_i \geq 1$ for $i \in [1, n]$. Suppose by absurd that $\bigcap_{x\in S} Ivar(x\sigma)$ is not empty. Then there exist $j_1, \ldots, j_n$ such that for $i \in [1, n]$: $1 \leq j_i \leq m_i$ and $im_{L_{j_1}^{x_1}} = im_{L_{j_2}^{x_2}} = \ldots = im_{L_{j_n}^{x_n}}$. Then by Lemma 4.9 it follows that $L_{j_1}, L_{j_2}, \ldots, L_{j_n}$ are all the same literal, say L and $x_1, \ldots, x_n$ are all contained in L. This contradicts 1. □

Lemma 4.12 *Let $share(S)$ be a disjunct of ψ. Suppose that $Var(A(\psi)) = \emptyset$. Then $\bigcap_{x\in S} Ivar(x\sigma) = \emptyset$.*

Proof. From $Var(A(\psi)) = \emptyset$ it follows that $S \subseteq dom(\sigma)$. If for some $x \in S$, $x\sigma$ is obtained from step 1.2 or from step 3 of *Prod* then it is a term containing only one fresh variable or it is a constant. Then the result follows immediate (recall that $|S| \geq 2$, by assumption). Otherwise every x in S occurs in a disjunct of ψ of the form $\neg share(S')$. Moreover since ψ is in normal form then $S \not\subseteq S'$ for every disjunct of ψ of the form $\neg share(S')$. Then 1 and 2 of Lemma 4.11 are satisfied. Thus $\bigcap_{x\in S} Ivar(x\sigma) = \emptyset$. □

Lemma 4.13 *If L, with relation symbol var or ground, is a disjunct of $A(\psi)$ such that $Var(L) = \emptyset$ then L is equivalent to false.*

Proof. Initially $A(\psi)$ satisfies the property because ψ is in flat form, hence the argument of an unary atom is a variable. The application of step 1 transforms all literals of the form $\neg ground(x)$ (first subcase) or $ground(x)$ (second subcase) or $\neg var(x)$ (third subcase) into an assertion equivalent to *false*. Finally step 2 and step 3 transform all atoms of the form $var(x)$ into an assertion equivalent to *false*. □

Theorem 4.14 *Let ψ be an assertion in normal form. Suppose that ψ is not a propositional constant. Then the algorithm Prod applied to ψ produces a substitution σ which does not belong to $[\![\psi]\!]$.*

Proof. *Prod* terminates when all the variables of ψ have been considered, hence $Var(A(\psi))$ becomes empty. Then the result follows by Lemma 4.10, Lemma 4.12 and Lemma 4.13. □

Proof of Theorem 4.7

By Lemma 4.2, Lemma 4.4 and the fact that (G7) can be applied only a finite number of times, it follows that TP terminates. Suppose that $TP(\phi) = true$. Then ϕ true follows from Lemma 4.2, Lemma 4.4 and Theorem 3.1.

We prove the converse by contraposition. Suppose that $TP(\phi)$ is not equal to *true*. Then $TP(\phi)$ is a conjunction of assertions in normal form, none of them equal to *true*, since rule (G7) has been applied. If one conjunct of $TP(\phi)$ is equal to the propositional constant *false* then ϕ is equivalent to *false*. Otherwise consider a conjunct ψ of $TP(\phi)$. Let σ be the substitution produced by applying the algorithm *Prod* to ψ. Then by Theorem 4.14 it follows that σ does not belong to $[\![\psi]\!]$. Hence ϕ is not true. □

5 Application

We illustrate how the truth procedure TP can be applied to mechanize some parts of the reasoning when proving the partial correctness of a logic program. Partial correctness will here be described in terms of properties of substitutions that are the intermediate results of the computations of a logic program, starting with a certain class of goals, by associating an assertion to each program point before or after an atom in the body of a clause. The class of goals considered is described by means of a goal and an assertion, called precondition, which specifies the possible values of the variables of the goal. Then every clause $h \leftarrow b_1 \ldots b_n$ of the program is annotated with assertions $h \leftarrow I_0\ b_1\ I_1 \ldots b_n\ I_n$, one assertion for every program point. An assertion associated with a program point is said to be a global invariant for the class of goals considered, if it holds every time a computation (of a goal of the considered class) reaches the correspondent program point. If the I_i's are shown to be global invariants for the class of goals considered, then the annotated program is said to be *partially correct* (with respect to the class of goals considered and with respect to these assertions). For instance, consider the following (fragment of the) annotated Prolog program `contained`:

$c1$: `contained(empty,y)` $\leftarrow I_0^{c1}$.
$c2$: `contained(node(`$\mathtt{x}_l$`,x,`$\mathtt{x}_r$`),y)` $\leftarrow$
  I_0^{c2} `member(x,y)` I_1^{c2} `contained(`$\mathtt{x}_l$`,y)` I_2^{c2} `contained(`$\mathtt{x}_r$`,y)` I_3^{c2}.

This program defines the binary relation *contained*, such that `contained(t,l)` holds if t is a binary tree whose nodes are contained in the list l. The program is used in [JL89] to illustrate the relevance of having information about aliasing of program variables at compile time. In particular, it is argued that the recursive calls in $c2$ may be executed in parallel if every time one of them is called, y is ground and x_l and x_r do not share. As an example, we show that `contained` satisfies this condition when the following class of goals is considered: $g = \leftarrow$ `contained(x,y)` with precondition $I_0^g = var(x) \wedge ground(y)$. In this example, the program computes all the trees whose nodes are contained in the list described by the ground term y. To this end, we prove that `contained` is partially correct with respect to this class of goals and with respect to the following assertions associated with the corresponding program points.

$I_0^{c1} = true$,
$I_0^{c2} = var(x_l, x_r) \wedge \neg share(\{x_l, x_r\}) \wedge ground(y)$,
$I_1^{c2} = I_0^{c2}$,
$I_2^{c2} = var(x_r) \wedge \neg share(\{x_l, x_r\}) \wedge ground(y)$,
$I_3^{c2} = \neg share(\{x_l, x_r\}) \wedge ground(y)$,

where, for a relation symbol p which is equal to *ground* or *var*, $p(x_1, \ldots, x_n)$ is used as shorthand for $p(x_1) \wedge \ldots \wedge p(x_n)$.

To prove the partial correctness of `contained`, we apply an inductive method informally illustrated as follows: let a be either (the atom of) g or an atom of the body of some clause of `contained`. Let I_1 and I_2 be the two assertions associated with the program points before and after a, respectively (in case a is the atom of g, assume that $I_1^g = true$ is the assertion associated with the point after g). Let I_1^t denote an assertion obtained from I_1 as follows: for all the variables $x_1, \ldots, x_k$ which could share with some variable occurring in a, replace $x_1, \ldots, x_k$ with the fresh variables $z_1, \ldots, z_k$, and set the sequence $(x_1, \ldots, x_k)$ to be equal to a suitable instance of $(z_1, \ldots, z_k)$. Consider a variant $ci' : h' \leftarrow I_0^{ci'}\, b_1\, I_1^{ci'} \ldots b_n\, I_n^{ci'}$ of a (annotated) clause ci of the program, $i \in [1, 2]$, such that ci' has no variables in common with $I_1^t\, a\, I_2$.

1. For an arbitrary substitution α in the semantics of I_1 consider the following conditions: a) $ci'\alpha$ is a variant of ci' having no variable in common with $(I_1\, a\, I_2)\alpha$; b) $a\alpha$ and $h'\alpha$ are

unifiable. If a) and b) are satisfied then show that $\alpha\beta$ is in the semantics of $I_0^{ci'}$, where β is a fixed most general unifier of $a\alpha$ and h'.

2. For an arbitrary substitution δ in the semantics of the rightmost assertion $I_n^{ci'}$ of ci', consider the following conditions: a) δ is in the semantics of I_1^t; b) for every variable x occurring in I_1^t but not in $\{x_1, \dots, x_k\}$, $x\delta$ and $ci'\delta$ have no variables in common; c) $h'\delta$ and $a\delta$ are equal. If a), b) and c) are satisfied then show that δ is in the semantics of I_2.

Step 1 corresponds to showing that when an atom calls a clause then the leftmost assertion of the clause is satisfied. Step 2 corresponds to showing that when the execution of a clause is terminated, then the assertion after the atom that has called the clause is satisfied. The variables $z_1, \dots, z_k$ of I_1^t represent the values of $x_1, \dots, x_k$ before ci' is called. The call of ci' can affect the values of $x_1, \dots, x_k$, which become instances of $z_1, \dots, z_k$. Notice that this is the only information about $x_1, \dots, x_k$ given by I_1^t. Moreover, I_1^t together with condition b) of step 2 are used to retrieve information about those variables occurring in I_1 which do not share with any variable occurring in a. Finally, the equality in condition c) of step 2 is used to retrieve information about the variables occurring in a. Notice that the Prolog selection rule, which selects atoms in the body of a clause from left to right, is assumed.

To describe step 1 syntactically, i.e., without referring to substitutions and most general unifiers, one can view the unification of a and h' as a function $sp_{a,h'}$ which maps a set of substitutions (the α's) into a set of substitutions (the γ's obtained by composing α with β). This has been done in [CM92], where a set of substitutions is expressed by means of an assertion and the unification of two atoms is described by means of a predicate transformer.

To describe step 2 syntactically, we define I_1^t as follows:

$$I_1^t \overset{\text{def}}{=} inst((x_1, \dots, x_k), (z_1, \dots, z_k), (y_1, \dots, y_j)) \wedge {I_1}^{x_1, \dots, x_k}_{z_1, \dots, z_k},$$

where $(x_1, \dots, x_k)$ denotes the sequence of elements of the set $Var(I_1 a I_2) \setminus Y$, with

$$Y = \{y \mid I_1 \Rightarrow \neg share(y, x), \text{ for all } x \text{ occurring in } a\},$$

$(z_1, \dots, z_k)$ is a variant of $(x_1, \dots, x_k)$ consisting of fresh variables, and $\{y_1, \dots, y_j\}$ is equal to $\{z \mid z = z_i \text{ for some } i \in [1, k] \text{ s.t. } x_i \in Var(a)\}$. Moreover, $\phi^{x_1, \dots, x_k}_{z_1, \dots, z_k}$ denotes the assertion obtained from ϕ by replacing every occurrence of x_i with z_i, for $i \in [1, k]$.
The semantics of the new assertions $r = s$ and $inst(r, s, t)$ is defined as follows:

$[\![r = s]\!] = \{\alpha \mid r\alpha = s\alpha\}$,

$[\![inst(r, s, t)]\!] = \{\alpha \mid r\alpha = s\alpha\beta \text{ for some } \beta \text{ s.t. } dom(\beta) \subseteq Var(t\alpha)\}$.

Using the function $sp_{a,h'}$ and the above definition of I_1^t, one can formalize steps 1 and 2 by means of the following implications, which are based on the assertional method of Colussi and Marchiori [CM91] (see also [AM94]).

$sp_{a,h'}(I_1 \wedge var(ci') \wedge \neg share(ci', I_1 a I_2 \cup ci')) \Rightarrow I_0^{ci'}$; **CALL**

$(I_n^{ci'} \wedge I_1^t \wedge \neg share(Y \cup \{z_1, \dots, z_k\}, ci') \wedge a = h') \Rightarrow I_2$, **EXIT**

where Y, $z_1, \dots, z_k$ and I_1^t are defined as above.
The assertion $var(ci') \wedge \neg share(ci', I_1 a I_2 \cup ci')$ used in $CALL$ expresses the fact that when ci' is called, it is renamed apart. Notice that we have used here $share(o_1, o_2)$ as shorthand for

$$\bigvee_{x \in Var(o_1), y \in (Var(o_2) \setminus \{x\})} share(\{x, y\}),$$

for some syntactic objects o_1, o_2. Moreover, the notation $var(o_1)$ is used as shorthand for $\bigwedge_{x \in Var(o_1)} var(x)$.

So the proof that `contained` is partially correct reduces to the verification of a number of implications. The truth procedure TP can be used to mechanize some of these tests. For instance, consider I_1^{c2} `contained(x`$_l$`,y)` I_2^{c2} and the variant $c2'$ of $c2$ obtained replacing x with x', for every variable x occurring in the atoms or in the assertions of $c2$. The following two implications are obtained:

(a) $x_l = node(x'_l, x', x'_r) \wedge y = y' \wedge var(x', x'_l, x'_r, x_r) \wedge ground(y) \wedge$
$\neg share(\{x'_l, x'_r\}) \wedge \neg share(x_r, \{x'_l, x'_r, x'\}) \;\Rightarrow$
$var(x'_l, x'_r) \wedge \neg share(\{x'_l, x'_r\}) \wedge ground(y');$

(b) $\neg share(\{x'_r, x'_l\}) \wedge ground(y') \wedge inst(x_l, z, z) \wedge var(z, x_r) \wedge \neg share(\{z, x_r\}) \wedge$
$ground(y) \wedge \neg share(\{x_r, y, z\}, \{x'_l, x'_r, x', y'\}) \wedge$
$x_l = node(x'_l, x', x'_r) \wedge y = y' \;\Rightarrow$
$var(x_r) \wedge \neg share(\{x_l, x_r\}) \wedge ground(y).$

These implications contain the relation symbols $=$ and $inst$ which are not in the assertion language $\mathcal{A}$ of our logic (see Definition 2.1). Then (a) and (b) can be transformed in assertions of $\mathcal{A}$ as follows:

(i) replace every assertion of the form $inst(r, s, t)$ by the following conjunction:

$$(ground(s) \Rightarrow ground(r)) \wedge (var(r) \Rightarrow var(s));$$

(ii) replace every equality $s = t$ by the following conjunction:

$$(ground(s) \Leftrightarrow ground(t)) \wedge (var(s) \Leftrightarrow var(t)) \wedge (\neg ground(s) \Rightarrow share(s, t)).$$

Notice that the transformations (i) and (ii) are sound, in the sense that the information about groundness and sharing given by the transformed assertion holds also for the original one. To show this formally, let $\mathcal{A}'$ be the smallest set A of formulas containing $\mathcal{A}$, containing the atoms $r = s$ and $inst(r, s, t)$ for all terms r, s, t in $Term$, and with the property that if ϕ and ψ are in A then both $\psi \wedge \phi$ and $\psi \vee \phi$ are in A. Then the following result holds.

Lemma 5.1 *Let ϕ be an assertion in $\mathcal{A}'$. Let $ap(\phi)$ be the assertion of $\mathcal{A}$ obtained applying the transformations specified by (i) and (ii). For every assertion ψ of $\mathcal{A}$ if $ap(\phi) \Rightarrow \psi$ is true in $\mathcal{A}$ then $\phi \Rightarrow \psi$ is true in $\mathcal{A}'$.*

Proof. $\phi \Rightarrow ap(\phi)$ is true in $\mathcal{A}'$. □

Now, apply the transformation to the assertions (a) and (b) and apply the truth procedure TP to the resulting assertions (after having eliminated all the "$\Rightarrow$" symbols using the equivalence $\phi \Rightarrow \psi = \neg\phi \vee \psi$). The outcome is *true*, as expected. Then from Lemma 5.1, implications (a) and (b) are true.

6 Conclusion

In this paper a logic has been introduced, which allows to model some relevant properties used in static analysis of logic programs, namely *var*, *ground* and *share*. Soundness, completeness and decidability of this logic have been proven. It has been illustrated how the truth procedure TP introduced to prove the decidability of the logic can be applied to mechanize some parts of the reasoning when proving the partial correctness of a logic program.

Another possible area of application of the results of this paper we intend to investigate is abstract interpretation. Our logic could be used as abstract domain in an abstract interpretation framework for the study of aliasing in logic programs. This framework could be defined as follows: the logic is used as abstract domain and the axiomatization of the unification as predicate transformer *sp*, given in [CM92], is used to model unification. Since the assertion obtained by applying *sp* is not in general in the assertion language of the logic, one would have to provide a suitable approximation of the result. Alternatively, the logic can be used as abstract domain to approximate a suitable semantics for logic programs, as the one given in [CMM94]: since this semantics is based on a predicate transformer, an abstract interpretation framework can be defined, based on the theory given in [CC79]. We have the impression that the two approaches sketched above would provide information about aliasing and groundness with a high degree of accuracy; however they would be rather expensive, thus penalizing the efficiency of the resulting analysis.

Acknowledgments I would like to thank Jan Rutten and one anonymous referee of an earlier draft: with their helpful suggestions and detailed comments, both the content and the exposition of this paper have improved. Thanks also to Livio Colussi for useful discussions and to the referees of this conference. Krzysztof Apt suggested, already some years ago, the topic of this paper.

References

[AH87] S. Abramsky and C. Hankin. An Introduction to Abstract Interpretation. *In Abstract Interpretation of declarative languages*, pp. 9–31, eds. S. Abramsky and C. Hankin, Ellis Horwood, 1987.

[AM94] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from Modes through Types to Assertions. *Formal Aspects of Computing*, 1994. In print.

[CM91] L. Colussi and E. Marchiori. Proving Correctness of Logic Programs Using Axiomatic Semantics. *Proceedings of the Eight International Conference on Logic Programming*, pp. 629–644, 1991.

[CM92] L. Colussi and E. Marchiori. Unification as Predicate Transformer. Preliminary version in *Proceedings JICSLP' 92*, 67–85, 1992. Revised version submitted.

[CMM94] L. Colussi, E. Marchiori and M. Marchiori. Combining Logic and Control to Characterize Global Invariants for Prolog Programs. *CWI Report*, The Netherlands, 1994.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation : a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pp. 238–251, 1977.

[CC79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pp. 269–282, 1979.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Report LIX/RR/92/08*, 1992. To appear in the special issue on Abstract Interpretation of the Journal of Logic Programming.

[CFW91] A. Cortesi, G. Filé and W. Winsborough. *Prop* Revisited: Propositional Formula as Abstract Domain. *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pp. 322–327, 1991.

[DM88] W. Drabent and J. Małuszyǹski. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science 59:1*, pp. 133–155, 1988.

[JL89] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. *Proceedings of the North American Conference on Logic Programming*, pp. 155–165, 1989.

[MS89] K. Marriott and H. Søndergaard. Notes for a Tutorial on Abstract Interpretation of Logic Programs. *North American Conference on Logic Programming*, 1989.