On Quantified Negative Queries

Alessandra Di Pierro
Dipartimento di Matematica e Informatica,
Università di Udine,
Via delle Scienze 206, 33100 Udine, Italy
dipierro@dimi.uniud.it

Włodzimierz Drabent
IPIPAN, Polish Academy of Sciences,
Ordona 21, Pl – 01-237 Warszawa, Poland
wdr@ipipan.waw.pl

Abstract

Given a logic program P and a goal G, we introduce a notion which states when an SLD-tree for $P \cup \{G\}$ *instantiates* a set of variables V *with respect to* another one, W. We call this notion *weak instantiation*, as it is a generalization of the *instantiation* property introduced in [3]. A negation rule based on instantiation, the so-called Negation As Instantiation rule (NAI), allows for inferring existentially closed negative queries, that is formulas of the form $\exists\neg Q$, from logic programs. We show that, by using the new notion, we can infer a larger class of negative queries, namely the class of the queries of the form $\forall_W \exists_V \neg Q$ and of the form $\forall_W \exists_V \forall_U \neg Q$, where U is the set of the remaining variables of Q.

1 Introduction

In order to infer negative literals from a logic program, Clark introduced the Negation As Failure (NAF) rule and defined its declarative semantics in terms of the completion, Comp(P), of a program P ([2]). However, this rule allows us to infer only a small part of the negative information which could be drawn from a program, namely the universally closed negative literals (more generally, universally closed negated queries). NAF works as a test: For a program P and a query Q it is able to check whether $\text{Comp}(P) \models \neg Q$ (or, equivalently, $\text{Comp}(P) \models \forall\neg Q$).

One generalization of NAF is constructive negation: Methods of finding instances $Q\theta$ of a given query Q such that $\neg Q\theta$ is a logical consequence of Comp(P).

Recently another generalization of negation as failure was proposed. It is Negation As Instantiation (NAI) rule [3], which allows us to derive existentially closed negative literals (or negated queries). The semantical justification of this inference rule is

still the completion of the program, but over an extended language L^*, containing infinitely many constant symbols, denoted by $\mathrm{Comp}_{L^*}(P)$. As shown in [3], if, for a program P and a query Q, there exists an SLD-tree that instantiates variables $\overline{x}$, then $\mathrm{Comp}_{L^*}(P) \models \exists\neg Q$ and, moreover, $\mathrm{Comp}_{L^*}(P) \models \exists\overline{x}\neg Q$.[1] Actually, as shown in Section 3.1, the existence of such a tree implies that $\mathrm{Comp}_{L^*}(P) \models \exists\overline{x}\forall\overline{y}\neg Q$ (where $\overline{y} = var(Q)\setminus\overline{x}$). Moreover, we will show that the reverse (completeness) holds too, in the case of definite programs and queries. However, if we are interested in deriving formulae of the form $\forall\exists\overline{x}\neg Q$, NAI is bound to be incomplete, even for positive programs and queries: There are programs and goals for which $\mathrm{Comp}_{L^*}(P) \models \exists\overline{x}\neg Q$, but there does not exist a tree for Q that instantiates $\overline{x}$.

In this paper we generalize the notion of instantiation introduced in [3]. With the new notion of weak instantiation we are able to derive conclusions of the form $\mathrm{Comp}_L(P) \models \exists\overline{x}\neg Q$ also in cases when $\mathrm{Comp}_L(P) \not\models \exists\overline{x}\forall\overline{y}\neg Q$. Thus, we define a negation rule based on the new notion of weak instantiation and prove both its soundness and completeness with respect to $\mathrm{Comp}_L(P)$. For the semantics to be correct, a different extension L of the language of the program is needed.

As a first step, in this paper we consider positive programs and queries only.

2 Preliminaries

We refer to [1, 9] for the basics of logic programming, including the negation as failure rule and the Clark completion semantics. For the purposes of this paper, a program P is a definite Horn clause program and a query Q is a conjunction of atoms.

A language L consists of the (well-formed) logical formulas built out of three disjoint sets of symbols: a set of *function symbols*, a set of *predicate symbols* and a set *Var* of *variable symbols*. Each function and predicate symbol is associated with a number representing its arity. Function symbols whose arity is 0 are also called *constant symbols*. We denote by *Term* the set of terms $t, u \ldots$ built of *Var* and the set of function symbols.

A substitution is a mapping $\theta : Var \to Term$ such that the set $Dom(\theta) = \{x \mid \theta(x) \neq x\}$ (*domain* of θ) is finite. A substitution θ is also denoted by the set $\{x_1/t_1, \ldots, x_n/t_n\}$, where $\{x_1, \ldots, x_n\} = Dom(\theta)$ and $t_i = \theta(x_i)$, $i = 1, \ldots, n$. The empty substitution is denoted by ε. The notion of substitution can be naturally extended to terms. The composition $\theta\sigma$ of the substitutions θ and σ is defined as the functional composition. The pre-ordering $\leq$ on substitutions is such that $\theta \leq \sigma$ iff there exists θ' such that $\theta\theta' = \sigma$.

The application of the substitution θ to the atom A is denoted by $A\theta$. The relation $A \leq A'$ (A is less instantiated than A') holds iff there exists θ such that $A\theta = A'$. A is called a variant of A' if there exist θ and θ' such that $A\theta = A'$, and $A'\theta' = A$; θ (θ') is called a renaming for A (A').

For a formula F, we use $\forall F$, $\exists F$ to denote respectively $\forall x_1, \ldots, \forall x_n F$ and $\exists x_1, \ldots, \exists x_n F$, where $x_1, \ldots, x_n$ are all the free variables occurring in F.

[1] Deriving this formula is called in [3] "amalgamation of NAF and NAI". For the definition of an instantiating tree see section 3 below.

For a given language L, $\mathrm{Comp}_L(P)$ is the Clark completion of the program P over the language L, and CET_L the Clark equality theory for L, included in $\mathrm{Comp}_L(P)$. When the language is known, the Clark completion of P is indicated by $\mathrm{Comp}(P)$. Usually, it is the language whose function and predicate symbols are those occurring in the program P and the considered query Q. We will refer to it as the language of P and Q.

We will use the notation $|\overline{x}|$ to indicate the length of the tuple $\overline{x}$, and the symbol '$\equiv$' for syntactic identity (of terms, etc.).

3 Negation As Instantiation

The basic observation leading to the idea of *negation as instantiation* is that fresh constant symbols, that is constant symbols which do not belong to the language of the program and the goal under consideration, can play the role of existentially quantified variables. In fact, if every branch of an SLD-tree for $P \cup \{\leftarrow A\}$ either fails or instantiates some of the variables of A, then for a ground substitution η instantiating all variables to distinct fresh constants, the corresponding SLD-tree for $P \cup \{\leftarrow A\eta\}$ finitely fails. Thus, by the soundness of NAF, we can deduce $\neg A\eta$, and finally $\exists\neg A$. Therefore, it is sufficient to extend the underlying language by infinitely many constant symbols, to obtain the appropriate reference theory for validly inferring formulas like $\exists\neg A$. As shown in [3], the Negation As Instantiation (NAI) rule is, indeed, sound and complete with respect to the completion of P, $Comp_{L^*}(P)$, over the extended language L^*. The formal definition of the notion of *instantiation*, used in the above informal description of NAI, is given by means of the property *Inst*.

Definition 3.1 (Instantiation) *Let $V = \{x_1, \ldots, x_n\}$, θ be a substitution and p a predicate symbol of arity n.*

$$Inst(\theta, V) \Leftrightarrow p(x_1, \ldots, x_n)\theta \not\leq p(x_1, \ldots, x_n).$$

In words, the property $Inst(\theta, V)$ holds iff θ is not a renaming for $p(x_1, \ldots, x_n)$, ($p(x_1, \ldots, x_n)\theta$ is not a variant of $p(x_1, \ldots, x_n)$).

Note that if θ instantiates V, then it instantiates any $V' \supseteq V$. An equivalent definition that will come in handy in the proofs and definitions given further on, is as follows.

Proposition 3.2

$Inst(\theta, V)$ *iff*
- *there exists $x \in V$ such that $x\theta \notin Var$, or*
- *there exist $x, y \in V$ such that $x\theta \equiv y\theta \in Var$.*

Proof The (if) part is obvious. For the (only if) part, assume by contradiction that for all $x, y \in V$, $x\theta$ and $y\theta$ are two distinct variables. Then, $\neg Inst(\theta, V)$ holds. □

Definition 3.3 (Instantiating SLD-tree) *Let P be a program, Q a query, $V \subseteq var(Q)$. Let TR be an SLD-tree for P and Q. Then TR instantiates V iff for every branch ξ of TR one of the following holds:*

- *ξ finitely fails, or*
- *$Inst(\theta_1 \cdots \theta_k, V)$ holds, where $\theta_1 \cdots \theta_k$ are the substitutions labeling the first k edges of ξ, for some $k \geq 1$.*

This definition differs from (but is equivalent to) the original one. Note that if TR instantiates V then the set of those its nodes whose accumulated substitutions do not instantiate V is finite (by König's lemma). Thus in the definition above there exists an n such that, for all branches, $n > k \geq 1$. So it is sufficient to inspect the tree only to some restricted depth, in order to check that it instantiates V. This is why the original definition calls such tree "finitely instantiating".

The operational semantics for NAI is defined by the Failure by Finite Instantiation set (the FFI set), consisting of all atoms A, for which there exists an SLD-tree which instantiates the variables occurring in A in the the sense explained above. This has been shown to correspond to the set of atoms whose existentially quantified negation is a logical consequence of $Comp_{L^*}(P)$, where L^* is the language of P enriched with infinitely many new constant symbols.

Theorem 3.4 (*Soundness and completeness of the NAI rule, [3]*)

$$Comp_{L^*}(P) \models \exists \neg A \text{ iff } A \in FFI.$$

3.1 Strong soundness and completeness of NAI

Actually, the existence of an instantiating tree for P and Q implies more than just $\exists \neg Q$. In [3, 4] it is shown that it implies $\exists \overline{x} \neg Q$ provided the tree instantiates $\overline{x}$. (This usage of NAI is called there amalgamation of NAI and NAF). We show something more:

Theorem 3.5 (**Strong soundness of NAI**) *Let P be a program, Q a query, $\overline{x}$ a tuple of variables occurring in Q. Let $\overline{y}$ be the remaining variables of Q. If there exists an SLD-tree for P and Q that instantiates $\overline{x}$ then*

$$Comp_{L^*}(P) \models \exists \overline{x} \forall \overline{y} \neg Q$$

For a proof of this theorem, see its generalization, Theorem 4.10. Clearly, $\exists \overline{x} \forall \overline{y} \neg Q$ implies $\forall \overline{y} \exists \overline{x} \neg Q$, but not vice-versa. Hence, the notion of an instantiating tree is not sufficient to infer formulas of the form $\forall \overline{y} \exists \overline{x} \neg Q$. The way of inferring such formulas will be presented in the next section.

Now, we show that the reverse of Theorem 3.5 holds for the NAI rule.

Theorem 3.6 (**Strong completeness of NAI**) *Let P be a program, Q a query, and $\overline{x} \subseteq var(Q)$. Let $\overline{z}$ be the remaining variables of Q.*
If $Comp_{L^*}(P) \models \exists \bar{x} \forall \bar{z} \neg Q$, then there exists an SLD-tree for $P \cup \{\leftarrow Q\}$ which instantiates $\bar{x}$.

For a proof of Theorem 3.6, observe that an SLD-tree for $P \cup \{\leftarrow A\}$ which instantiates $\bar{x}$ is an SLD-tree for $P \cup \{\leftarrow A\}$ which instantiates $\bar{x}$ w.r.t. the empty set, in the sense explained in Section 4.1. Then, Theorem 4.13 applies.

4 A new notion of instantiation

The notion of instantiation defined in [3] does not allow to derive as much negative information as possible. There exist queries Q for which $\exists \overline{x} \neg Q$ is true in the completion semantics (equivalently, $\forall \exists \overline{x} \neg Q$ is true) and yet there are no SLD-trees for Q that instantiate $\overline{x}$.

Example 4.1 *Consider the program*

$$P \quad = \quad \{ \quad r(z,z) \leftarrow \quad \}$$

Observe that $Comp_L(P) \models \exists x \neg r(x,y)$. However, $Comp_L(P) \not\models \exists x \forall y \neg r(x,y)$. Hence, by the soundness theorem (Theorem 3.5) there does not exist an SLD-tree for $P \cup \{\leftarrow r(x,y)\}$ that instantiates x. So, we are unable to infer $\exists x \neg r(x,y)$.

Example 4.2 *Consider the program*

$$P' \quad = \quad \{ \quad r(z,f(z)) \leftarrow \quad \}$$

Observe that $Comp_L(P) \models \exists x \neg r(x,y)$. Again, the amalgamation rule cannot infer the formula $\exists x \neg r(x,y)$, as the SLD-tree for $P \cup \{\leftarrow r(x,y)\}$ does not instantiate $\{x\}$, according to the definition of Inst.

The point is that a substitution which links the existentially quantified variables $\overline{x}$ and the free variables $\overline{y}$, or the free variables with some terms containing $\overline{x}$, should be considered as instantiating $\overline{x}$. In other words, we want to define a notion of "relative instantiation" such that the substitution $\vartheta = \{x = z, y = z\}$ of Example 4.1, as well as the substitution $\vartheta' = \{x = z, y = f(z)\}$ of Example 4.2, turn out to be instantiating $\{x\}$ with respect to $\{y\}$.

4.1 Weak instantiation

Definition 4.3 (**Weak instantiation**) *Let $V = \{x_1, \ldots, x_n\}$, $W = \{y_1, \ldots, y_m\}$ be two disjoint sets of variables, and let θ be a substitution. We say that θ instantiates V w.r.t. W, in symbols $Winst(\theta, V, W)$, iff*

- *$Inst(\theta, V)$, or*
- *$V\theta$ and $W\theta$ have a common variable.*

Note that for $W = \emptyset$, $Winst(\theta, V, W)$ is equivalent to $Inst(\theta, V)$, (weak instantiation subsumes instantiation).

The following proposition provides an alternative definition of weak instantiation.

Proposition 4.4 *A substitution θ instantiates V w.r.t. W iff*

- *there exists $x \in V$ such that $x\theta$ is not a variable, or*
- *there exist $x, y \in V$ such that $x\theta \equiv y\theta$ is a variable, or*

- *there exist $x \in V$ and $y \in W$ such that $x\theta$ is a variable occurring in $y\theta$.*

Definition 4.5 *Let P be a program, G a goal, $V, W \subseteq Var$ two disjoint sets of variables. Let TR be an SLD-tree for $P \cup \{G\}$. Then TR instantiates V w.r.t. W iff for every branch ξ of TR one of the following holds:*

- *ξ finitely fails, or*
- *$Winst(\theta_1 \cdots \theta_k, V, W)$ is true, where $\theta_1, \ldots, \theta_k$ are the substitutions labeling the first k edges of ξ for some $k \geq 1$.*

TR is called a *weakly instantiating* tree for $P \cup \{G\}$.
Observe that this definition subsumes both that of a finitely failed SLD-tree, ([1]) and that of an instantiating SLD-tree ([3]). In fact, if $V \cup W$ is the set of the variables occurring in Q, then the case $V = \emptyset$ corresponds to the case of a finitely failed tree (the predicate *Winst* is *false*); on the other hand, when $W = \emptyset$ a weakly instantiating tree is just a finitely instantiating tree under the Definition 3.2 of [3].

4.2 Negation by Weak Instantiation

The notion of weak instantiation can be used to derive negative information from programs. Namely, for a program P, a query Q and disjoint tuples $\overline{x}$, $\overline{y}$ of variables of Q, if an SLD-tree for P and Q instantiates $\overline{x}$ w.r.t. $\overline{y}$ then we can infer $\exists \overline{x} \neg Q$. This is justified by the fact that the formula $\exists \overline{x} \neg Q$ is true in the completion semantics of P. For the details see Theorem 4.10. We call this rule the Negation by Weak Instantiation (NWI) rule. We show some examples to clarify its possible use.

Example 4.6 *Let plus be a predicate whose third argument is the sum of the others. It can be defined by the following program*

$$PLUS = \{\ plus(x, 0, x) \leftarrow,\quad plus(x, s(y), s(z)) \leftarrow plus(x, y, z)\ \ \}.$$

$PLUS \cup \{\leftarrow plus(x, s^2(0), y)\}$ has an SLD-tree where the goal variable x is instantiated w.r.t. y (y results instantiated to $s^2(x)$). From the NWI rule, we can derive $\exists x \neg plus(x, s^2(0), y)$. Indeed, $Comp(PLUS) \models \exists x \neg plus(x, s^2(0), y)$.

Note that x is not instantiated according to the original notion of instantiation. In fact, in this case, the formula $\exists x \forall y \neg plus(x, s^2(0), y)$ is not true.

Now we show an example where the NWI rule behaves like the amalgamation rule.

Example 4.7 *Consider the program PLUS of Example 4.6.*
We have that $Comp(PLUS) \models \exists x \neg plus(x, y, s^n(0))$. From the NWI rule, we can derive $\exists x \neg plus(x, y, s^n(0))$: in fact, $PLUS \cup \{\leftarrow plus(x, y, s^n(0))\}$ has an SLD-tree where the goal variable x results instantiated w.r.t. y.

Note that in the latter example, $Inst(\theta_i, \{x\})$ holds for all $i = 1, \ldots, n$, where θ_i is the composition of the substitutions labeling the first $i+1$ edges of the i-th branch. Indeed the formula

$$\exists x \forall y \neg plus(x, y, s^n(0))$$

(hence $\forall y \exists x \neg plus(x, y, s^n(0))$) is true.

The NWI rule can be efficiently implemented by suitably extending the implementation of the NAF rule. In Prolog this is typically obtained by means of a *cut–fail* combination. The well-known program is:

$$\begin{array}{lll} \mathtt{not}(G) & \leftarrow. & \\ \mathtt{not}(G) & \leftarrow & G, !, fail. \end{array}$$

In order to implement NWI, the idea is as follows. For a goal $G(\overline{x}, \overline{y})$, generate $|\overline{x}|$ new function symbols $\overline{f}$ of arity $|\overline{y}|$, and then apply `not` to the goal G' obtained from G by replacing all occurrences of the variables $\overline{x}$ by $\overline{f(\overline{y})}$.

Note that, as in the NAF case, this implementation works correctly provided that the *cut–fail* combination is applied to *ground* goals. This requirement is always satisfied in the case of the NAI rule, as *all* variables of the goal are replaced by new constant symbols.

4.3 Correctness of NWI

A natural way to prove soundness of negation as weak instantiation seems as follows. From an SLD-tree for A instantiating $\overline{x}$ w.r.t. $\overline{y}$ construct a finitely failed tree for $A\{\overline{x}/\overline{f}(\overline{y})\}$, where $\overline{f}$ are $|\overline{x}|$ new function symbols of arity $|\overline{y}|$. However, we found the technical details of the proof difficult and here we present another proof.

Lemma 4.8 *Let Q be a node of an SLD-tree, and let $Q_1, \ldots, Q_n$, $(n \geq 0)$ be its children. Then*

$$Comp(P) \models Q \leftrightarrow \bigvee_{i=1}^{n} \exists \overline{w}\ (\overline{v} = \overline{v}\theta_i \wedge Q_i)$$

where θ_i are the mgu's corresponding to Q_i, $\overline{w}$ are the "new" variables and $\overline{v}$ are "all the variables".
(Here $\overline{v} = \overline{v}\theta_i$ is a conjunction of equalities).

This is a version of Lemma 15.3 of [9] and of Lemma 4.1 of [5], and the proof is similar.

The soundness of negation as weak instantiation is implied by the following lemma.

Lemma 4.9 *Let P be a program, Q a query, $\overline{x}, \overline{y}$ a sequence of distinct variables. Let language L_0 contain exactly the functors of P and Q. Assume that there exists an SLD-tree for P and Q weakly instantiating $\overline{x}$ w.r.t. $\overline{y}$ such that variables of $\overline{x}, \overline{y}$*

are not used in the resolution[2]. *Then in any model of Comp(P) formula* $\neg Q$ *is true in any valuation* ν *in which the values of* $\overline{x}$ *are*

1. *distinct,*
2. *"new", i.e. not in the range of the interpretations of the functors from* L_0,
3. *for every* $x \in \overline{x}$ *and* $y \in \overline{y}$ *and for any term* $t[x]$ *of language* L_0 *and containing* x, *the value* $\nu(y)$ *differs from* $\mu(t[x])$ *for any valuation* μ *such that* $\mu(x) = \nu(x)$.

Proof Without lack of generality we may assume that all the variables are ordered and if an mgu contains a binding v/w then w precedes v in the ordering. We will also assume that the variables of $\bar{x}$ precede those of $\bar{y}$ in this ordering and those of $\bar{y}$ precede the remaining ones.

The proof is by induction on the number of nodes in the tree for which the corresponding accumulated substitution does not instantiate $\overline{x}$ w.r.t. $\overline{y}$. (Let us call such nodes non-instantiating). It follows from König's lemma that this number is finite.

Consider an arbitrary model of Comp(P). (We will write $\models_\nu \neg Q$, not mentioning the model). Consider the root Q of the SLD-tree, its children $Q_1, \ldots, Q_n$ ($n \geq 0$) and the corresponding mgu's $\theta_1, \ldots, \theta_n$. Assume that the lemma holds for all the trees with the number of non-instantiating nodes less than the number of such nodes in the tree under consideration. Then $\models_\nu \neg Q$ iff

$$\models_\nu \bigwedge_i \forall \overline{w}(\neg(\overline{v} = \overline{v}\theta_i) \vee \neg Q_i) \qquad (1)$$

where $\overline{w}$ and $\overline{v}$ are as in the previous lemma. For $i = 1, \ldots, n$ we have two cases:

1. If θ_i instantiates $\overline{x}$ w.r.t. $\overline{y}$, we show that $\models_\nu \forall \overline{w} \neg(\overline{v} = \overline{v}\theta_i)$. We have three cases:

   (a) For some x from $\overline{x}$, term $x\theta_i$ is not a variable. Equation $x = x\theta_i$ is a conjunct from the conjunction of equations $\overline{v} = \overline{v}\theta_i$. As the main functor of $x\theta_i$ occurs in P or Q, the equation is false in any valuation in which the value of x is the same as in ν.

   (b) For some x, x' from $\overline{x}$, $x\theta_i$ and $x'\theta_i$ are the same variable. In any valuation that differs from ν only on variables from $\overline{w}$, the values of x and x' are distinct (by condition 1). So $x = x\theta_i \wedge x' = x'\theta_i$ is false is false in any such valuation.

   (c) For some x from $\overline{x}$ and y from $\overline{y}$, $x\theta_i$ is a variable occurring in term $y\theta_i$. We may assume that $x\theta_i \equiv x$, otherwise $x\theta_i$ is another variable from $\overline{x}$ and the previous case applies. Now by condition 3, in any valuation that differs from ν only on variables from $\overline{w}$, the value of y differs from the value of $y\theta_i$. So the equation $y = y\theta_i$ is false.

[2]No variable of $\overline{x}, \overline{y}$ occurs in the renamed clauses used in the SLD-derivations that are the branches of the tree.

2. θ_i does not instantiate $\overline{x}$ w.r.t. $\overline{y}$. Note that $\overline{x}\theta_i \equiv \overline{x}$. First we show that the subtree rooted at Q_i instantiates $\overline{x}$ w.r.t. $vars(\overline{y}\theta_i)$.

   Consider any non failing branch of the subtree. It has a node with the accumulated substitution φ such that $\theta_i\varphi$ instantiates $\overline{x}$ w.r.t. $\overline{y}$. So there exists a variable x from $\overline{x}$ such that either (a) $x\theta_i\varphi \equiv x\varphi$ is not a variable or (b) $x\theta_i\varphi \equiv x'\theta_i\varphi$ is a variable for some $x' \in \overline{x}$, so $x\varphi \equiv x'\varphi$ is a variable; or (c) $x\theta_i\varphi \equiv x\varphi$ is a variable that occurs in $y\theta_i\varphi$, for some $y \in \overline{y}$. Hence φ instantiates $\overline{x}$ w.r.t. $vars(\overline{y}\theta_i)$.

   Consider a valuation π that differs from ν only on the variables from $\overline{w}$. It remains to show that

   $$\models_\pi \neg(\overline{v} = \overline{v}\theta_i) \vee \neg Q_i.$$

   Assume that $\models_\pi \overline{v} = \overline{v}\theta_i$ (otherwise the conclusion is immediate). Obviously, π satisfies conditions 1 and 2 of the lemma.

   Assume that π does not satisfy condition 3. So $\pi(z) = \mu(t[x])$ for some x from $\overline{x}$, some variable z occurring in $\overline{y}\theta_i$, some term $t[x]$ and some valuation μ such that $\mu(x) = \pi(x) = \nu(x)$. Let $s[z]$ be a term from the tuple $\overline{y}\theta_i$, such that z occurs in $s[z]$. $s[z] \equiv y\theta_i$, for some $y \in \overline{y}$.

   We may assume that $vars(s[z]) \cap vars(t[x]) \subseteq \{x\}$. (Otherwise we can rename the variables of $t[x]$ and modify μ accordingly). Let μ' be a valuation such that $\mu'(s[z]) = \pi(s[z])$ and $\mu'(t[x]) = \mu(t[x])$. Such a valuation exists as $\pi(x) = \mu(x)$. Now we have $\mu'(x) = \nu(x)$ and $\mu'(z) = \pi(z) = \mu(t[x]) = \mu'(t[x])$. Then $\nu(y) = \pi(y) = \pi(s[z]) = \mu'(s[z]) = \mu'(s[t[x]])$; contradiction with condition 3 for ν.

   So π satisfies conditions 3 of the lemma. By the inductive assumption, $\models_\pi \neg Q_i$.

Thus, we showed that each conjunct of the conjunction in (1) is true in ν. This completes the proof. □

Theorem 4.10 (**Strong soundness**) *Let* P *be a program,* Q *a query,* $\overline{x}, \overline{y}$ *a sequence of distinct variables occurring in* Q. *Let* $\overline{z}$ *be the remaining variables of* Q. *If there exists an SLD-tree for* P *and* Q *weakly instantiating* $\overline{x}$ *w.r.t.* $\overline{y}$, *then*

$$Comp_L(P) \models \forall \overline{y} \exists \overline{x} \forall \overline{z} \neg Q$$

where the set of functors of L *contains the functors of* P *and of* Q *and* $|\overline{x}|$ *distinct* $|\overline{y}|$*-ary functors that do occur neither in* P *nor in* Q.

Obviously, $\text{Comp}_L(P) \models \forall \overline{y} \exists \overline{x} \forall \overline{z} \neg Q$ implies $\text{Comp}_L(P) \models \exists \overline{x} \neg Q$.

Proof

From Lemma 4.9. Note that any model of $\text{Comp}_L(P)$ is a model of Comp(P). We show that for any model of CET_L, for any valuation ν_0 there exists a valuation ν_1 that agrees with ν_0 on the variables outside of $\overline{x}$, such that any valuation ν that agrees with ν_1 on the variables outside of $\overline{z}$ satisfies the conditions of the lemma.

Let $\overline{x} = x_1, \ldots, x_k$, let $f_1, \ldots, f_k$ be the "new" functors. Consider an arbitrary valuation ν_0. Let $\nu(x_i) = \nu_0(f_i(\overline{y}))$, for $i = 1, \ldots, k$; let $\nu(z)$ be arbitrary for any z from $\overline{z}$ and let $\nu(v) = \nu_0(v)$ for any other variable v.

Conditions 1 and 2 of the lemma obviously hold. Consider an x_i, a variable y from $\overline{y}$ and a term $t[x_i]$. Let μ be a valuation such that $\mu(x_i) = \nu(x_i)$. We have to show that $\nu(y) \neq \mu(t[x_i])$.

Consider a renaming ρ such that $\overline{y}\rho$ is a tuple of variables not occurring elsewhere. Take a valuation μ' that agrees with μ on the variables of $t[x_i]$ and such that $\mu'(v\rho) = \nu(v)$ for any $v \in \overline{y}$. So $\mu'(x_i) = \mu(x_i) = \nu(x_i)$ and $\mu'(f_i(\overline{y}\rho)) = \nu(f_i(\overline{y}))$. From this we obtain $\mu'(x_i) = \mu'(f_i(\overline{y}\rho))$, as $\nu(x_i) = \nu(f_i(\overline{y}))$. Now we have $\mu(t[x_i]) = \mu'(t[x_i]) = \mu'(t[f_i(\overline{y}\rho)]) \neq \mu'(y\rho)$ (from CET_L). As $\mu'(y\rho) = \nu(y)$, we obtain $\mu(t[x_i]) \neq \nu(y)$.

From the lemma $\neg Q$ is true for valuation ν. □

Note that the requirement that the variables of $\overline{x}, \overline{y}$ occur in Q can be weakened. It is sufficient to require that these variables are not used in the resolution in the tree. Also the requirement on the language can be modified. For instance, instead of $|\overline{x}|$ new $|\overline{y}|$-ary functors we may use $|\overline{x}|$ new constants and a new binary functor.

Note also that Theorem 3.5 (strong soundness of negation as instantiation) follows immediately from Theorem 4.10 with $\overline{y} = \emptyset$. In this case, the language L contains $|\overline{x}|$ distinct constant symbols. Thus L is indeed a subset of the language L^* used in Section 3.

4.4 Completeness of NWI

In this section we show that negation by weak instantiation is complete w.r.t. $\mathrm{Comp}_L(P)$, for any fair selection rule. In contrast with the correctness theorem, this result holds for any L containing the symbols of the program and the query. In our proof we use the Lloyd-Topor transformation described in [10] and a constructive negation method (the SLDFA-resolution of [5]), which is sound and complete for the Kunen semantics.

Definition 4.11 *Let P be a program , Q a query, $F = \forall\overline{y}\exists\overline{x}\forall\overline{z}\neg Q$, and p, q, r new predicate symbols. Consider the following general logic program*

$$P' = P \cup \{\ p \leftarrow \neg q(\overline{y}),\quad q(\overline{y}) \leftarrow \neg r(\overline{x}, \overline{y}),\quad r(\overline{x}, \overline{y}) \leftarrow Q \qquad \}$$

We call P' the Lloyd-Topor transformation of P with respect to F.

Clearly, by the definition of the completion we have that $\neg p \leftrightarrow \neg\exists\overline{y}\neg\exists\overline{x}\neg\exists\overline{z}Q \leftrightarrow \forall\overline{y}\exists\overline{x}\forall\overline{z}\neg Q$ holds w.r.t. the new program (it is a logical consequence of $\mathrm{Comp}(P')$).

The completeness of the negation by weak instantiation follows by the next lemma, whose proof is omitted due to space limitation.

Lemma 4.12 *Let P be a program, Q a conjunction of atoms, $\overline{x}, \overline{y}$ a sequence of distinct variables occurring in Q, and $\overline{z}$ the remaining variables of Q. Let F be $\forall\overline{y}\exists\overline{x}\forall\overline{z}\neg Q$ and P' the Lloyd-Topor transformation of P with respect to F.*

If there exists an SLDFA finitely failed tree for P' and query p then there exists an SLD-tree for P and Q (via the same selection rule) that weakly instantiates $\overline{x}$ w.r.t. $\overline{y}$.

Theorem 4.13 (Completeness of NWI) *Let P be a program, Q a query, $\overline{x}, \overline{y}$ a sequence of distinct variables occurring in Q, and $\overline{z}$ the remaining variables of Q. Let L be any language whose set of function symbols contains those of the language of P and Q, and has at least two elements.*
If $\mathrm{Comp}_L(P) \models \forall\overline{y}\exists\overline{x}\forall\overline{z}\neg Q$, then for any fair selection rule there exists an SLD-tree for P and Q weakly instantiating $\overline{x}$ w.r.t. $\overline{y}$.

Proof

$\mathrm{Comp}_L(P) \models \forall\overline{y}\exists\overline{x}\forall\overline{z}\neg Q$ iff $\mathrm{Comp}(P') \models \neg p$ iff $\mathrm{Comp}(P') \models_3 \neg p$ (as P' is call consistent and query $\neg p$ is strict with respect to P').

By the completeness of the SLDFA-resolution (cf. [5, Theorem 5.1]), if $\mathrm{Comp}(P') \models_3 \neg p$ then for any fair selection rule there exists an SLDFA finitely failed tree for p. Then, by Lemma 4.12, for the same selection rule there exists an SLD-tree for P and Q that weakly instantiates $\overline{x}$ w.r.t. $\overline{y}$.

□

5 Conclusions and future work

In this paper we studied what it means that in an SLD-tree certain variables are instantiated in a certain way. This work is a continuation of [3, 4]. First we showed that if an SLD-tree with the root Q instantiates (in the sense of [3]) some variables $\overline{x}$, then this implies not only $\exists\overline{x}\neg Q$, but also $\exists\overline{x}\forall\overline{z}\neg Q$ (where $\overline{z}$ are the remaining variables of Q: cf. Theorem 3.5). Then, we introduced a new notion of weak instantiation that makes it possible to derive formulae of the form $\exists\overline{x}\neg Q$ even when $\exists\overline{x}\forall\overline{z}\neg Q$ does not hold. The semantics of reference is given by Clark's completion over a certain extended language. We proved both soundness and completeness of negation as weak instantiation.

In line with the approach of [6, 7], the Failure by Weak Instantiation can be seen as one of the so-called *observable properties* of a program P. The set of atoms A for which the $\forall\exists\forall$-closure of $\neg A$ can be inferred from P represents yet another *failure set* for the program P, along with the standard finite failure set FF, the failure by instantiation set FFI of [3], etc. (see [4] for a classification of the semantical characterizations of the various operational properties of a logic program). The correctness and the completeness results shown in this paper give a model-theoretic characterization of the new observable. We plan to provide it with an equivalent fixpoint characterization. The fixpoint operator will be (a suitable modification of) the immediate consequence operator, T_c, of the C-semantics ([6, 7]).

In this paper we dealt with definite programs only. In the context of this work it is natural to consider general programs and goals in which subformulae of the form $\exists\bar{x}\forall\bar{z}\neg Q$ may occur, in addition to positive and negative literals. We believe that the results presented in this paper could be easily extended to such general programs and goals. The semantics of interest would be the three-valued completion semantics of Kunen ([8]).

6 Acknowledgments

The research of Alessandra Di Pierro was carried out during a post-doc stay at the Department of Computer and Information Science, Linköping University, in the context of the HCM project "Logic Program Synthesis and Transformation" (CHRX-CT93-0414).

The research of Włodzimierz Drabent was partly supported by Polish Academy of Sciences and by Swedish Research Council for Engineering Sciences (dnr 221-93-942).

References

[1] K. R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[3] A. Di Pierro, M. Martelli, and C. Palamidessi. Negation as Instantiation. *Information and Computation*, 120(2):263–278, August 1995.

[4] A. Di Pierro. Negation and Infinite Computations in Logic Programming. PhD thesis, Universitá di Pisa, 1994. Technical Report 3/94.

[5] W. Drabent. What is failure? an approach to constructive negation. *Acta Informatica*, 32:27–59, 1995.

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

[7] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.

[8] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

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

[10] J. W. Lloyd and R. W. Topor. Making Prolog more Expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

On Extended Logic Languages supporting Program Structuring*

Rosa Arruabarrena+, Marisa Navarro+

Abstract

In this paper we study some logic programming languages which extend the positive Horn clauses, permitting more expressiveness. Firstly, we present a single syntax for the programming languages that we study. This syntax is an extension of those of the languages presented in [GMR 92] and in [Mil 89]. Concretely, the following logical connectives are added to be used in goals and clause bodies: implication, disjunction and existential and universal quantifiers. This extension of Horn clause language allows to structure programs by considering implications D⊃G as "blocks". According to the semantics associated to these blocks we can obtain several extensions, characterising each one by a different visibility rule (static or dynamic scope rules) and by a different mode of predicate definitions (extension or overriding), as in conventional programming languages. Our main aim is to make a comparison among these languages, in terms of their operational semantics, and to implement them. The functionality of two of these languages is given by means of examples, establishing the main differences between them and giving some hints on how they have been implemented. We also consider two variants of these two languages (with overriding of predicate definitions instead of extension), comparing them to the previous ones and commenting the implementation adjustments for them.

1. INTRODUCTION

In [NM 88, Mil 89, GMR 92] the syntax of positive Horn clauses is extended by permitting implication goals. This allows to structure programs in the following way: An implication goal D⊃G can be seen as a block where D are the local definitions related to G. According to the associated semantics, different block types will be obtained, being characterised each extension by a different visibility rule for locally defined clauses. In [GMR 92] "open blocks" and "static scope rules" are mostly examined, while in [Mil 89] an approach handling "open blocks" and "dynamic scope rules" is shown.

---

* This work has been partially supported by projects CICYT TIC95-1016-C02-02 and UPV 141226-EA209/94

+ Dept. Lenguajes y Sistemas Informáticos, Fac. Informática, UPV-EHU, 20.009 Donostia, Spain
e-mail: {jiparsar,jipnagom}@si.ehu.es, phone: 34-43 21 80 00, fax: 34-43 21 93 06

In this paper, we first present the syntax of the programming languages that we are going to use. The syntax is an extension of the language in [GMR 92] (by allowing the use of the disjunction and universal quantification) and an extension of the language in [Mil 89] (by universal quantification). Concretely, the following logical connectives are used in goals and in clause bodies: implication, disjunction and existential and universal quantifiers. Although we have only one syntax, we get two different languages, since different interpretations associated to each approach (static or dynamic scope rules) yield different results. [Mil 90] already considered this extended language (with universal quantification) as a form of endowing the language with "data abstractions", where definite clauses were called "first-order hereditary Harrop formulae".

In the mentioned papers some implementation suggestions were pointed out, but the extension task to cover the new connectives is not trivial. First of all, for each goal $G \equiv D \supset G'$ we have taken into account which is the world P where G is going to be solved and how this world changes (in each approach) after introducing the local definitions D. In the dynamic case, the new world will be the union of the previous world P and the new definitions D (this union constitutes a new environment), whereas in the static case the new world will be implemented by a stack of block definitions, where the new definitions D will be its top. In the later case, when an atomic goal A is executed, only a part of the current world (that is, a part of the stack of blocks) will be considered, depending on where the predicate A was statically defined.

Next, the fact of permitting existential quantification in goals complicates the implementation. At each moment along the computation we must know which are the "free variables" introduced in the world, in order to use them properly. These free variables come from the fact that implication goals can be existentially quantified, for instance, $\exists x(p(x) \supset q(x))$. Moreover, allowing universal quantification in a goal [Mil 90, Nad 93] involves keeping track of the existential variables that have appeared previously in the goal and forming "dynamic" Skolem functions of these variables when universal quantifiers are encountered since, for example, $\exists z\, \forall u\, q(z,u)$ and $\forall u\, \exists z\, q(z,u)$ are different goals. Lastly, we add the disjunction in goals, $(p \vee q)$, which can be easily implemented. All these facts are covered by the implemented resolution procedure and some examples are given in order to show its functionality.

Both languages, with dynamic scope rule and with static scope rule, have been implemented in the logic language Eclipse, running on a Sun machine. This implementation intends to be useful mainly to establish the outstanding differences between both defined languages. It can also be used as an extension of Prolog.

These two languages allow to extend predicate definitions when adding a new block. Other approach that can be taken is that a new definition of a predicate overrides its previous definition. We also consider this variant (for both languages) with "overriding" of predicate definitions and we comment the adjustments to be made to the implementation for these new languages.

The paper is organised as follows. In section 2 we fix the syntax of our languages and the two visibility rules for open blocks. In sections 3 and 4 we define the operational semantics of the dynamic scope language and the static scope language, based on [Mil 90] and [GMR 92], respectively. In section 5 some examples are considered to illustrate the functionality of both languages and to give some hints on how they have been implemented. In section 6 two new languages (with overriding) are shown and compared to the previous ones. We conclude with some future work.

2. PRELIMINARIES

We shall start describing the syntax shared by our languages. This syntax is an extension of the positive Horn clauses, by adding in goals as well as in the clause bodies the following logical connectives: implication ($\supset$), disjunction ($\vee$), existential ($\exists$) and universal ($\forall$) quantifiers.

In the following we shall assume that A is a metavariable for representing an *atomic* formula, G for a *goal* formula and D for a *definite clause*. The last two will be defined recursively as:

$$G := A \mid G1 \wedge G2 \mid \exists x G \mid D \supset G \mid \forall x G \mid G1 \vee G2$$
$$D := A \mid D1 \wedge D2 \mid \forall x D \mid A :\!- G$$

A *program* P will be a set of closed definite clauses. In a definite clause A:- G the body G can be an implication goal, i.e. $D \supset G'$. In this case, $D \supset G'$ will be considered as a block, understanding that G' is a goal and D is a set of defined clauses local to G'. As G' can contain other implications as well, any number of nested blocks are permitted. Note also that there are two implication connectives, namely $\supset$ and :-, in goals and in definite clauses, respectively. The meaning of both connectives can be the same or different depending on the underlying logic.

In addition to this, a block structured language needs the explicit use of quantifiers to fix the scope of the variables. Besides, this allows the use of non-local variables in locally defined clauses. For this reason, we will write explicitly all the quantifiers in the clauses, in spite of the fact that it could complicate the reading of some examples.

The following program which calculates the reverse of a list is an example (from [Mil 89]) that uses an implication as a block.

Example 2.1

$$\forall L\, \forall K\, (\, \text{reverse}(L,K) :\!-\, (\ \forall K\, (\text{rev1}([],K,K)) \wedge \forall X\, \forall L\, \forall K\, \forall B\, (\, \text{rev1}([X|L],K,B) :\!-\, \text{rev1}(L,K,[X|B])\,)\ \supset \text{rev1}(L,K,[])\ ).$$

The code for the predicate *rev1* is only accessible when the predicate *reverse* is being evaluated. ■

According to the meaning that we relate to an implication goal, $D \supset G$, different types of blocks can be considered (see [BLM 94, GMR 92]):

Closed blocks: A block is closed if a goal $D \supset G$ is derivable from P when G is derivable from D, disregarding the content of P. In other words, to derive G, only clauses in

D can be used, not those in P. This idea of blocks can be used as a form of incorporating modules in logic programming [GMR 92].

Open blocks: A block D⊃G is open if, when proving G, both local definitions to G (the clauses in D) and the clauses in the external world (P) can be used. This idea of blocks provides a tool for structuring logic programs, usual in other conventional languages. Nevertheless, as the set of clauses that define a predicate (or procedure) can occur in different blocks, a language with open blocks requires scope rules for locally defined clauses. Concretely, there are two feasible alternatives:

Dynamic scope rule: The set of clauses which can be used to solve a goal G depends on the sequence of goals generated until that moment in the proof containing G. This set can only be determined dynamically. For instance, to solve the goal D⊃(G∧ (D'⊃G')) from a program P means solving G in the "world" P∪D, which is the union of the clauses in P and the clauses in D, and solving G' in the world P∪D∪D'.

Static scope rule: The set of clauses which can be used to solve a goal G depends on the program's block structure. So, this set can be determined statically. To solve an atomic goal A defined in a block, only the clauses defined in that block or in the external enclosing blocks can be used. This requires to establish explicitly the order of these blocks.

These rules will be formally specified in sections 3 and 4. In order to have an intuitive idea of both rules, we show the following example:

Example 2.2

Let P be the program { $\forall x(q(x):\text{-}r(x))$,
$s:\text{-}\ (\ (\forall x(p(x):\text{-}\exists y q(y)) \wedge r(a)) \supset p(b)\ )$ } and G be the goal s.

Due to the dynamic scope rule, G can be derived from P in the following way: s is derivable from P if and only if $p(b)$ is derivable from the program P', obtained as the union of P and set of local clauses { $\forall x(p(x):\text{-}\exists y q(y))$, $r(a)$}. To solve $p(b)$ from P' means solving $\exists y q(y)$ from P', which is true, since $q(a)$ can be derived from P' (from the rule $\forall x(q(x):\text{-}r(x))$ and the fact $r(a)$).

Nevertheless, according to the static scope rule, G can not be derived from P. Once again, s can be derived from P if and only if $p(b)$ can be derived from the program P "augmented" with the local clauses { $\forall x(p(x):\text{-}\exists y q(y))$, $r(a)$}. As before, solving $p(b)$ means solving $\exists y q(y)$. But, since the predicate q is defined only in the external block (or main program), the local definitions are not visible for q. Therefore $\exists y q(y)$ has to be proved *only* from the clauses in P. Then, there is no proof. ■

3. THE DYNAMIC SCOPE PROGRAMMING LANGUAGE

In this section, we consider a first language, taken from [Mil 90], with the syntax given in section 2 and whose operational semantics reflects the use of open blocks with visibility rules of dynamic scope. We shall use $Pl\text{-}_d G$ to mean that a (closed) goal G can be operationally derived from a program P. In the rest of the paper, we use the subscript d within the derivation symbol (|-) to indicate the dynamic scope language. The underlying logic of this language is the intuitionistic logic and, in particular, the connectives :- and ⊃ are both interpreted as the intuitionistic implication (see [Mil 89]).

While defining the operational semantics, to avoid variable renaming and substitutions, we follow [Mil 89] using the notation [P]. As usual, given a program P, [P] is defined as the smallest set of formulas satisfying the following recursive conditions:

1. $[P] \supseteq P$.
2. If $D1 \wedge D2 \in [P]$ then $D1 \in [P]$ and $D2 \in [P]$.
3. If $\forall x D \in [P]$ then $[x/t]D \in [P]$ for all terms t.

Informally, [P] contains all the formulas that can be obtained from the clauses in P by instantiation and conjunction. [x/t]D denotes the result of replacing t for free occurrences of x in D.

A *world* is the set of program clauses that exist at a certain moment when deriving a goal. The initial world will consist of the clauses of a program P. Given a world P and a closed goal G, $Pl\text{-}_d G$ is defined by induction on the structure of G, by the following rules (where we assume that A represents an atomic formula):

D-rules

(1) $Pl\text{-}_d A$, if $A \in [P]$.
(2) $Pl\text{-}_d A$, if there is a formula $A\text{:-}G \in [P]$ and $Pl\text{-}_d G$.
(3) $Pl\text{-}_d G1 \wedge G2$, if $Pl\text{-}_d G1$ and $Pl\text{-}_d G2$.
(4) $Pl\text{-}_d G1 \vee G2$, if $Pl\text{-}_d G1$ or $Pl\text{-}_d G2$.
(5) $Pl\text{-}_d \exists x G$, if there is some term t such that $Pl\text{-}_d [x/t]G$.
(6) $Pl\text{-}_d \forall x G$, if $Pl\text{-}_d [x/a]G$ for a new constant term a.
(7) $Pl\text{-}_d D \supset G$, if $P \cup D Pl\text{-}_d G$.

Due to (7), the world may change when deriving a goal. That is, to prove a goal D⊃G in a world P requires to prove G in the world obtained by augmenting the set of clauses of P with the set of local clauses D. Consequently, each goal will be solved in a world, which is the union of previous occurred block definitions. So, special care has to be taken to restore the world P after having solved G from P∪D. This means that the clauses D, which have been loaded into memory to solve G, will have to be unloaded. For example, to solve D⊃((D'⊃G')∧G) from a program P, firstly the clauses in D will have to be loaded, later will do the same with the D' clauses in order to solve G'. After this proof has been calculated, the memory will have to be restored, that is, the locally defined clauses D' will have to be unloaded from memory to solve G from only P∪D. Other remarkable point about implementation would be not to repeat clauses in the memory (though a multiset can be viewed as a correct implementation of a set) to avoid uncontrolled growth of the stored code. In particular, it is important when handling recursive programs as the following example.

Example 3.1

This program calculates the depth of n-ary trees, being "empty_ntree/0" and "ntree/2" the n-ary tree constructors:

```
depth( empty_ntree, 0).
∀R ∀SubTrees_L ∀DT ∀Max_D
    ( depth( ntree(R, SubTrees_L), DT):-
        (   ( depth_list( [], 0) ∧
              ∀X ∀DX ∀Y ∀DY ∀DTL ( depth_list( [X|Y], DTL):-
                                     depth(X, DX) ∧
                                     depth_list(Y, DY) ∧
                                     max(DX, DY, DTL)
                                 )
            ) ⊃ depth_list( SubTrees_L, Max_D)
        ) ∧
        DT is 1+ Max_D
    ).
```

4. THE STATIC SCOPE PROGRAMMING LANGUAGE

We define now a second language, as an extension of the language in [GMR 92], with the syntax given in section 2 and whose operational semantics reflects the use of open blocks with visibility rules of static scope. We shall use $P|-_s G$ to mean that a (closed) goal G can be operationally derived from a program P. In the rest of the paper, we use the subscript *s* within the derivation symbol (|-) to indicate the static scope language. In the underlying logic of this language the connectives :- and ⊃ are differently interpreted, while ⊃ is the intuitionistic implication :- is the classical one (see [GM 94, BLM 94]).

Similarly to the previous case, when deriving a goal, a world M obtained by adding to the initial program (D0) some block definitions (D1,...,Dn) has to be considered. However, in this case the world can not be built as the union of these sets of clauses, but as the ordered sequence of these blocks. Now the world M is denoted by the sequence D0!D1!...!Dn. This sequence can be seen as a stack of block definitions, where ! is the "push" operator, that is, M!D denotes a stack of block definitions with D at its top. The notation [P] is extended to [M], which is defined as the union of [Di], for all Di in M.

Given a world M and a closed goal G, $M|-_s G$ is defined by induction on the structure of G, by the following rules (where we assume that A represents an atomic formula):

S-rules

(1) $M|-_s A$, if $A \in [M]$.
(2) $D0!D1!D2!...!Dn|-_s A$, if there is a formula $A:-G \in [Di]$ and $D0!D1!...!Di|-_s G$, for some i ($0 \le i \le n$).
(3) $M|-_s G1 \wedge G2$, if $M|-_s G1$ and $M|-_s G2$.
(4) $M|-_s G1 \vee G2$, if $M|-_s G1$ or $M|-_s G2$.
(5) $M|-_s \exists x G$, if there is some term *t* such that $M|-_s [x/t]G$.
(6) $M|-_s \forall x G$, if $M|-_s [x/a]G$ for a new constant term *a*.
(7) $M|-_s D \supset G$, if $M!D|-_s G$

The block definitions are added to the world by means of rule (7). Due to (2), the world is explicitly described as a stack of block definitions. When proving an atomic goal A, if a rule A:-G defined in Di is chosen then the body G of the rule must be evaluated in D0!D1!...!Di, without using other rules defined along Di+1,...,Dn. This guarantees the static scope visibility, since the block definitions that have been added later are not visible for G. As it is necessary to identify which are the clauses belonging to each Di in the world D0!D1!...!Dn, special care has to be taken to handle them all in memory. In addition to this, the problem mentioned in the dynamic approach with respect to the recursive programs does not happen here, since multiple instances of the same block cannot exist at all (see [GMR 94]).

As in [GMR 92], rule (2) produces nondeterminism: if there were more than one candidate block, any could be selected. In terms of implementation, however, to search for all possible refutations of a goal will imply to consider all the indexes *i* which verify the conditions of the rule. We start the search from the block on the top of the stack and continue downwards.

Remark:
The rules of the dynamic scope language (D-rules) have been written following the notation of the papers [GMR 92, Mil 90] but they could be reformulated using the stack, instead of the union. In this new formulation, the rules of the dynamic scope language are the same of those given for the static scope language (S-rules), except rule 2 which has to be formulated in the following way:

(2') $D0!D1!D2!...!Dn|-_d A$, if there is a formula $A:-G \in [Di]$ for some i ($0 \le i \le n$) and $D0!D1!D2!...!Dn|-_d G$

That is, the body G of the rule defined in Di can be solved now using rules of any element of the whole stack. This formulation shows more clearly the differences between both operational semantics. In [GM 94] two subsets of these languages have been integrated into a single modal language.

5. IMPLEMENTATION ISSUES AND EXAMPLES

In this section we will focus on the implementation. Now by means of examples we comment some of the most relevant points that we have taken into account when implementing the system in order to compute properly the operational semantics defined for each language.

5.1 Dynamic versus static scope

Depending on the language, given a program and a goal formula, we can obtain different results. A simple example that illustrates the difference between both languages is the following:

Example 5.1.1

Program: {q :- p.}
Goal: ?- p⊃q

The inference process of the goal from the program will succeed in one language and will fail in the other one. The steps in both programming languages are the following:

* with dynamic scope

World	Goal
{q :- p.}	?- p⊃q
{q :- p.}∪{p.}	?- q
{q :- p.}∪{p.}	?- p
{q :- p.}∪{p.}	?- [].

the goal is derivable.

* with static scope

World	Goal
{q :- p.}	?- p⊃q
{q :- p.}!{p.}	?- q
{q :- p.}	?- p
	failure

the goal is not derivable ■

As it can be observed, at each moment it is necessary to know in which world has the goal to be proved. Thus, pairs (World, Goal) must be handled along the inference process.

Notice that when using the dynamic scope language, a world (where a goal has to be solved) is a set of program clauses which come from the union of the clauses in the previously occurred blocks along the inference process. In order to avoid repeated clauses in memory, as it was mentioned in section 3, we use the Eclipse predicate "variant/2". In addition to this, to restore the world W corresponding to the goal D⊃G after solving G from W∪D, we copy the world W in a file. Moreover, we maintain a "stack of files" since nested blocks can occur. In contrast, when using the static scope language, a world is a sequence of set of clauses where each set corresponds to the clauses of a previously occurred block. When implementing, we copy each block in a file and the sequence of blocks in a stack of files. Each time a goal D⊃G has to be solved, a new file is created containing D and it is pushed on the stack of files. The obtained stack will be the current world where G will have to be proved and, afterwards, the previous world will be restored by popping the stack. In more detail, in each level *i* of the file stack which implements the world D0!...!Dn we have a triple where the first element is a file name (corresponding to the block Di), the second is the list of all the predicates defined in D0!...!Di and the third is the list of all predicates defined in Di. These lists permit to determine fast where predicates have been defined without searching for them along the clauses in the files.

5.2 Existential goals

When a goal ∃xG has to be proved, firstly, every appearance of the existentially quantified variable *x* is marked and the quantifier dropped from the goal. We mark the existential variable by replacing it by a new variable, *vex*_N. After the replacement, the new goal to be proved will be [*x*/*vex*_N]G. Let us examine the case ∃wG, where G is an implication goal, for instance, ∃w(p(w)⊃G'). After transforming the goal into p(*vex*_N)⊃G', the world is augmented with {p(*vex*_N).} and the new goal is G'. Then, when *vex*_N is instantiated in the refutation process of G', at that very moment the instantiation value replaces each occurrence of *vex*_N in G' and in the world's clauses. This is illustrated by the following example:

Example 5.2.1

Program: {q :- p(a)∧p(b).}
Goal: ∃w(p(w)⊃q)

* Using the dynamic scope programming language:

World	Goal	
{q :- p(a)∧p(b).}	?- ∃w(p(w)⊃q)	
{q :- p(a)∧p(b).}	?- p(*vex*_1)⊃q	% w is marked
{q :- p(a)∧p(b).}∪{p(*vex*_1).}	?- q	
{q :- p(a)∧p(b).}∪{p(*vex*_1).}	?- p(a)∧p(b)	
	{*vex*_1<-a}	% *vex*_1 is instantiated to *a*
{q :- p(a)∧p(b). }∪{p(*a*).}	?- p(b)	
	failure	

The inference process fails because we have p(*a*) in the world. This fits the operational semantics of the language.

* Using the static scope programming language: The answer is the same as above.

World	Goal	
{q :- p(a)∧p(b).}	?- ∃ w(p(w)⊃q)	
{q :- p(a)∧p(b).}	?- p(*vex*_1)⊃q	% w is marked
{q :- p(a)∧p(b).} ! {p(*vex*_1).}	?- q	
{q :- p(a)∧p(b).}	?- p(a)∧p(b)	
	failure	

■

In terms of implementation, a list of triples containing information of appeared existential variables is kept. When an existentially quantified variable is found in a goal, the process of marking variables will replace the occurrences of this variable by a new Eclipse atom *vex*_N which will uniquely identify this existential variable. This atom will be handled as a special variable by our unification algorithm. The variable and its associated identifier are stored as the two first components of a new triple in the list of existential variables, in the above example Lvex= [(w,*vex*_1,_)]. The third component of the triple will be the value that the associated identifier will get by unification, that is, Lvex= [(w,*vex*_1,*a*)]. In more detail, not only the pairs (World, Goal) are kept when deriving a goal but also the list of triples of all appeared existential variables.

Apart from this, although the goal ∃xG is closed, our system returns all the possible values for *x* that verify G, similarly to the Prolog language (the user have to indicate with ";" if he/she wants more answers). For this aim we use the list Lvex where each existential variable has its associated answer.

5.3 Universal goals

Universal and existential quantifiers can be alternated in goals. Thus in an universal goal, ∀xG, it is necessary that the variable *x* is replaced by an adequate Skolem function, that is, by a Skolem function of the existential variables that have previously occurred in the goal. The use of such a function ensures that these existential variables can not be instantiated by a term containing the instantiation for *x* . The Skolemization process has to be "dynamic", since an "static Skolemization" can not be sound for the underlying logic (see [Nad 93]). We consider two cases:

A) If both, the goal $\forall$xG and the world M, are closed (that is, there are no existential variables till the moment), the way the goal G is proved is by replacing *x* by a "new" constant (*cte*_N) and prove [x/*cte*_N]G in M. In implementation terms, this implies that new constant symbols (*cte_N*) have to be generated. For instance:

Example 5.3.1

Program: {$\forall$x q(f(x),x).}
Goal: $\forall$u $\exists$z q(z,u)

The steps of the inference process are the following:

World	Goal	
{$\forall$x q(f(x),x).}	?- $\forall$u $\exists$z q(z, u)	
{$\forall$x q(f(x),x).}	?- $\exists$z q(z, *cte*_1)	% *cte*_1 is a new constant for u
{$\forall$x q(f(x),x).}	?- q(*vex*_1, *cte*_1)	% the variable z is marked
	{*vex*_1 <-f(*cte*_1), x <- cte_1 }	
{$\forall$x q(f(x),x).}	?- []	

In this way (in both languages) the goal succeeds. The answer given by our system is u=*cte*_1 and z= f(*cte*_1) that represents the solution z= f(u). This could be directly obtained undoing the replacements that have been made by the Skolem process. ■

B) If either the goal $\forall$xG or the world M is not closed, then the variable *x* depends on other variables *vex*_1,....,*vex*_k which have appeared existentially quantified in the previous goals along the proof. The goal G has to be proved by replacing *x* by a new term *vfa*_N(*vex*_1,....,*vex*_k) which is formed by a "new" function symbol *vfa*_N applied to these existential variables, that is, we prove [x/*vfa*_N(*vex*_1,....,*vex*_k)]G in M. An example of this is the following:

Example 5.3.2

Program: {$\forall$x q(f(x),x).}
Goal: $\exists$z $\forall$u q(z, u)

Again, independently of the chosen language, the steps of the inference process are:

World	Goal	
{$\forall$x q(f(x),x).}	?- $\exists$z $\forall$u q(z,u)	
{$\forall$x q(f(x),x).}	?- $\forall$u q(*vex*_1,u)	% the variable z is marked
{$\forall$x q(f(x),x).}	?- q(*vex*_1,*vfa*_1(*vex*_1))	
	% *vfa*_1(*vex*_1) is a new term for u	
	failure	

In implementation terms, this implies that new functions symbols (*vfa_N*) have to be generated. If the system has "occur-check" then the substitution *vex*_1= f(*vfa*_1(*vex*_1)) will not be allowed, and the inference process will fail. This fits the operational semantics since from {$\forall$x q(f(x),x).} it is not possible to derive $\exists$z $\forall$u q(z,u) (neither in $\vdash_s$ nor in $\vdash_d$). In our system the occur check process is under implementation. ■

In [Nad 93] instead of using Skolem functions the adopted solution involves labeling constants and variables to constrain unification to achieve more efficiency.

6. LANGUAGES WITH OVERRIDING

The two languages that have been described in the previous sections allow to extend predicate definitions when adding a new block to the world, that is, the clauses which define a predicate *p* can occur in different blocks. Other approach consist in considering that, when adding a new block, the definition of a predicate *p* overrides the definitions of *p* in the blocks previously added to the world [MP 89]. These two new languages "with overriding" can be defined similarly to the static scope language. We consider then the S-rules. Again, the difference will be in rule (2) since, among the blocks containing the definitions of a predicate, only the definitions that are in the outermost block can be considered. Subsequently, definitions of the same predicate that were in lower levels in the stack would remain in hiding. In the following pred(A) denotes the predicate symbol of the atom A.

The rules of the static scope language with overriding, denoted SO-rules, will be as S-rules, except rule (2) which will be now:

(2") D0!D1!D2!...!Dn$\vdash_{so}$A, if there is a formula A:-G$\in$ [Di] and D0!D1!...!Di$\vdash_{so}$G, being i = max{j | there is a formula A':-G'$\in$ [Dj], pred(A')=pred(A)}

In the same way, the rules of the dynamic scope language with overriding, denoted DO-rules, will be as S-rules, except rule (2) which will be now (see remark in section 4):

(2''') D0!D1!D2!...!Dn$\vdash_{do}$A, if there is a formula A:-G$\in$ [Di] and D0!D1!...!Di$\vdash_{do}$G, being i = max{j | there is a formula A':-G'$\in$ [Dj], pred(A')=pred(A)}

As it has been mentioned earlier, from the implementation point of view, for both languages presented in section 3 and 4 we have to consider all indexes *i* satisfying rule (2). In fact, we start the search from the block on the top of the stack and continue downwards. For these new languages with overriding, simply we have to limit the search to find the biggest of such indexes.

7. CONCLUSIONS AND FUTURE WORK

This paper deals mainly with two programming languages which extend the positive Horn clause logic by introducing some connectives ($\supset$, $\vee$,$\forall$ and $\exists$) in goals and in clause bodies. We are especially interested in studying the differences between these two logic programming languages. Both of them consider implications as "open blocks" to structure programs, but while the first one has a dynamic scope rule for visibility of locally defined clauses, as in Miller's proposal [Mil 89, MIL 90], the second one has a static scope rule. This is, in fact, an extension of the language proposed in [GMR 92]. In order to compare both languages we study their operational semantics, in sections 3 and 4, and then establish the basis for their implementation. We have implemented both languages in the logic

programming language Eclipse, which has permitted using the backtracking process and the logic variables of the underlying system. In section 5 we show, by means of examples, the most relevant points to take into account when implementing, and we give some hints on how the implementations of both languages have been carried out. We also consider two variants of the previous languages, in section 6, with "overriding" (instead of extension) of predicate definitions. We give the operational semantics of these variants and we comment the adjustments to be made to our implementations to handle these two new cases.

From the logical point of view, the basis of the first language (the language with dynamic scope) is the "intuitionistic logic". In [Mil 89] a fixpoint semantics and a proof-theoretic semantics are given for this language. For the static scope language we would like to have a similar result, that is, a model-theoretic and a proof-theoretic semantics for it. [GMR 92] give a fixpoint semantics together with a model-theoretic semantics for their language that might be extended to ours. Nevertheless, in the case of the language with static scope it is not clear the underlying logic of the programming language. In [GM 94] the followed approach is to interpret the block languages within modal logic S4. However, our idea is to establish the underlying logic, with its proof calculus and the corresponding model-theory, and to obtain the static scope programming language as a subset of formulas in such logic, without translation to other logics.

Acknowledgments

We would like to thank the referees for their useful comments and remarks on the previous version of this paper. Following them the tittle has changed to the current one.

References

[BLM 94] Bugliesi, M.; Lamma, E.; Mello, P.
"Modularity in Logic Programming"
Journal of Logic Programming, vols. 19 y 20, pp. 443-502, 1994.

[GMR 92] Giordano, L.; Martelli, A.; Rossi, G.
"Extending Horn Clause Logic with implication goals"
Theoretical Computer Science, vol. 95, pp. 43-74, 1992.

[GMR 94] Giordano, L.; Martelli, A.; Rossi, G.
"Structured Prolog: A Language for Structured Logic Programming"
Software -- Concepts and Tools, vol. 15, pp. 125-145, 1994.

[GM 94] Giordano, L.; Martelli, A.
"Structuring Logic Programs: a Modal Approach"
Journal of Logic Programming, vol. 21, pp. 59-94, 1994.

[Mil 89] Miller, D.
"A Logical Analysis of Modules in Logic Programming"
Journal of Logic Programming, vol. 6, pp. 79-108, 1989.

[Mil 90] Miller, D.
"Abstractions in Logic Programming"
Logic and Computer Science, pp. 329-359, Academic Press, 1990.

[MP 89] Monteiro, L.; Porto A.
"Contextual Logic Programming"
Proc. 6th Int. Conf. of Logic Programming, Lisbon, pp. 284-299, 1989

[Nad 93] Nadathur, G.
"A Proof Procedure for the Logic of Hereditary Harrop Formulas"
Journal of Automated Reasoning, vol. 11, pp. 115-1145, 1993.

[NM 88] Nadathur, G.; Miller, D.
"An Overview of λProlog"
Proc. 5th Int. Conf. and Symp. on Logic Programming, pp. 810-827, MITT Press, 1989.