On Negation As Instantiation

Alessandra Di Pierro[1] and Włodzimierz Drabent[2]

[1] Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy
dipierro@dimi.unipi.it
[2] IPIPAN, Polish Academy of Sciences, Ordona 21, Pl – 01-237 Warszawa,
and IDA, Linköping University, S - 581 83 Linköping
wdr@ida.liu.se

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 $\forall_W \exists_V \forall_Z \neg Q$, where Z 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 $\mathrm{Comp}(P) \models \neg Q$ (or, equivalently, $\mathrm{Comp}(P) \models \forall\neg Q$).

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 the 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, $\mathrm{Comp}_{L^*}(P) \models \exists\overline{x}\neg Q$.[3] Actually, as shown in Section 3.1, the existence of such

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

a tree implies that $\text{Comp}_{L^*}(P) \models \exists\overline{x}\forall\overline{y}\neg Q$ (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 $\text{Comp}_{L^*}(P) \models \exists\overline{x}\neg Q$, 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 $\text{Comp}_L(P) \models \exists\overline{x}\neg Q$ also in cases when $\text{Comp}_L(P) \not\models \exists\overline{x}\forall\overline{y}\neg Q$. Thus, we define a negation rule based on the new notion of weak instantiation and prove both its soundness and completeness with respect to $\text{Comp}_L(P)$. 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, 11] 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.

For a given language L, $\text{Comp}_L(P)$ is the Clark completion of the program P over the language L, and CET_L the Clark equality theory for L, included in $\text{Comp}_L(P)$. When the language is known, the Clark completion of P is indicated by $\text{Comp}(P)$. 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 1. (Instantiation) Let $V = \{x_1, \ldots, x_n\}$, θ be a substitution (and p be an auxiliary 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 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.

(if) Obvious.
(only if) By contradiction, assume that for all $x, y \in V$, $x\theta$ and $y\theta$ are two distinct variables. Then, $\neg Inst(\theta, V)$ holds.

□

Definition 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 of 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 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 4. (*Soundness and completeness of the NAI rule,* [3])

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

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 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 15. 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 5 holds for the NAI rule.

Theorem 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 $Comp_{L^*}(P) \models \exists\bar{x}\forall\overline{z}\neg Q$, then there exists an SLD-tree for $P \cup \{\leftarrow Q\}$ which instantiates $\bar{x}$.

For a proof of Theorem 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 18 applies.

4 A new notion of instantiation

In this section we first discuss some limitations of negation as instantiation. In the following subsections we introduce a more general notion of weak instantiation, show how it can be used to derive negative information from programs and discuss its semantics. Then we study soundness and completeness properties of negation as weak 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 1. Consider the program

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

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 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 2. Consider the program

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

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 1, as well as the substitution $\vartheta' = \{x = z, y = f(z)\}$ of Example 2, turn out to be instantiating $\{x\}$ with respect to {y}.

4.1 Weak instantiation

Definition 7. (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 8. *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 and y are distinct, and $x\theta \equiv y\theta$ 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 9. (Weakly instantiating SLD-tree) Let P be a program, Q a query, $V, W \subseteq Var$ two disjoint sets of variables. Let TR be an SLD-tree for P and Q. 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 and Q.
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 15. We call this rule the Negation by Weak Instantiation (NWI) rule. We show some examples to clarify its possible use.

Example 3. Let *plus* be a predicate whose third argument is the sum of the others. It can be defined by the following program.

$$\begin{aligned} PLUS = \{\, & plus(x, 0, x) \leftarrow, \\ & plus(x, s(y), s(z)) \leftarrow plus(x, y, z) \,\}. \end{aligned}$$

$PLUS \cup \{\leftarrow plus(x, s^2(0), y)\}$ has an SLD-tree, TR, that instantiates the goal variable x w.r.t. y (y is bound 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 TR does not instantiate x 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. Consider the program *PLUS* of Example 3.
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 which instantiates the goal variable x 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.

Negation by Weak Instantiation may be implemented by using $|\overline{x}|$ new function symbols $\overline{f}$ of arity $|\overline{y}|$. Then, under a given selection rule, query $Q' = Q\{\overline{x}/\overline{f(\overline{y})}\}$ finitely fails iff the SLD-tree for Q instantiates $\overline{x}$ w.r.t. $\overline{y}$. Finite failure of Q' can be established in a standard way, by the usual (unsound) implementation of negation as failure (built-in \+ of Prolog).

Note that for such implementation to be correct it is necessary to use unification with occur check. (Otherwise, for instance, query $p(f(y), y)$ "succeeds" with program $\{p(v, v)\}$, although the SLD-tree for $p(x, y)$ instantiates x w.r.t. y).

4.3 The logical semantics of the instantiation

In this section we give a logical characterization of the predicate *Winst* (and then of *Inst*, as a particular case), in terms of the models of *CET*.

The fact we will exploit is that an accumulated substitution θ for (a prefix of) a branch of an SLD-tree, when rewritten as an equation set, results in a *solved form* equation set, θ being an idempotent substitution ([12]). We recall the definition of solved form equation set as given in [12].

An equation set is *solved* if it has the form $\{x_1 = t_1, \ldots, x_n = t_n\}$, and the x_i's are distinct variables which do not occur in the right hand side of any equation. The variables x_i are called *eliminable*; the remaining variables are called *parameters*. A variable x_i *depends* on a variable y, if y is a variable in t_i in the solved form.

In [12] an idempotent substitution $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ is associated to the equation set $\mathtt{eqn}(\theta) = \{v_1 = t_1, \ldots, v_n = t_n\}$, which is clearly in solved form.

Let V be a set of variables. For an equation in $\mathtt{eqn}(\theta)$ of the form $x = v$, where x, v are variables, consider the following transformation. Replace all the occurrences of v by x in the right hand sides of the other equations in $\mathtt{eqn}(\theta)$, and $x = v$ by $v = x$. Let $\mathtt{e}(\theta)$ be the equation set obtained from $\mathtt{eqn}(\theta)$ by repeatedly replacing, as described above, equations of the form $x = v$, where $x \in V$ and v is a variable not in V, until no such equations remain. Clearly, if θ does not instantiate V, then this procedure terminates. Moreover, $\mathtt{e}(\theta)$ is in solved form and equivalent to $\mathtt{eqn}(\theta)$.

Lemma 10. *Let θ be an idempotent substitution, V, W two disjoint sets of variables, and let $\mathtt{e}(\theta)$ be defined as above. Let $Z = var(\theta) \setminus (V \cup W)$. Then*

$$\neg\, Winst(\theta, V, W) \Rightarrow \quad \begin{array}{l} \textit{in } \mathtt{e}(\theta) \\ - \textit{ all } x \in V \textit{ are parameters, and} \\ - \textit{ only the variables in } Z \textit{ depend on the variables in } V. \end{array}$$

Proof.
If $\neg\, Winst(\theta, V, W)$, then θ does not instantiate V. So $\mathtt{e}(\theta)$ is well-defined. By Proposition 8,

$$\neg\, Winst(\theta, V, W) \text{ iff } \begin{array}{l} - \text{ for all } x, x' \in V, x\theta \text{ and } x'\theta \text{ are distinct variables, and} \\ - \text{ for all } x \in V, \text{ and } y \in W, x\theta \text{ does not occur in } y\theta. \end{array}$$

Thus, in $\mathtt{eqn}(\theta)$ all equations of the form $x = t$, $x \in V$ are such that t is a variable not in V. Moreover, if for some $x_1, x_2 \in V$, and variables v_1, v_2, $x_1 = v_1$ and $x_2 = v_2$ occur in $\mathtt{eqn}(\theta)$, then $v_1 \neq v_2$. So, in $\mathtt{e}(\theta)$ a variable $x \in V$ can only appear in equations of the form $v = x$, or in the right hand side of equations $z = t$, derived from some equation $z = t'$ occurring in $\mathtt{eqn}(\theta)$, with t' containing $x\theta$. This means that all $x \in V$ are parameters in $\mathtt{e}(\theta)$. Moreover, by the hypothesis, the latter equations cannot be of the form $y = y\theta$ with $y \in W$. Therefore, in $\mathtt{e}(\theta)$, only the variables in Z depend on the variables in V.

□

Proposition 11. *Let θ, V, W, Z be as in Lemma 10. Let L be a language containing the function symbols occurring in θ. Assume that L contains at least two function symbols. Then*

$$Winst(\theta, V, W) \textit{ iff } CET_L \models \forall_W \exists_V \forall_Z \neg \mathtt{eqn}(\theta),$$

Proof.

(only if) Let M be a model of CET_L and ν_0 an arbitrary valuation of the variables in W. We show how to extend ν_0 to a valuation ν of the variables in $V \cup W$, such that $\forall_Z \neg \mathtt{eqn}(\theta)$ is true under ν in M. This means $M \models \forall_W \exists_V \forall_Z \neg \mathtt{eqn}(\theta)$.
By Proposition 8, $Winst(\theta, V, W)$ iff one of the following three cases occurs:

1. For some $x \in V$, $x\theta$ is not a variable.
   Choose ν such that the value $\nu(x)$ is not in the range of the interpretation of the main function symbol of $x\theta$. Then $\exists_Z x = x\theta$ is false under ν. As $x = x\theta$ occurs in $\mathtt{eqn}(\theta)$, we have that $M \models_\nu \forall_Z \neg \mathtt{eqn}(\theta)$.
2. For some $x, x' \in V$, $x \neq x'$, $x\theta$ and $x'\theta$ are the same variable.
   Take ν such that $\nu(x) \neq \nu(x')$. Then $\exists_Z x = x\theta, x' = x'\theta$ is false under ν. As $x = x\theta$ and $x' = x'\theta$ occur in $\mathtt{eqn}(\theta)$, we have that $\exists_Z \mathtt{eqn}(\theta)$ is false under ν.
3. For some $x \in V$ and $y \in W$, $x\theta$ is a variable occurring in $y\theta$.
   Consider an extension ν' of ν_0 such that the equation $y = y\theta$ is true under ν'. In a model of CET_L, a value of a term uniquely determines the values of its sub-terms (as $f(\overline{t}) = f(\overline{s})$ implies $\overline{t} = \overline{s}$). So $\nu'(x\theta)$ is unique. Now,

choose ν such that $\nu(x) \neq \nu'(x\theta)$. We have that $\exists_Z(x = x\theta, y = y\theta)$ is false under ν. Hence $\forall_Z \neg \mathtt{eqn}(\theta)$ is true under ν, as $y = y\theta$ occurs in $\mathtt{eqn}(\theta)$, and either $x = x\theta$ occurs in $\mathtt{eqn}(\theta)$, or x and $x\theta$ are the same variable.

(if) Let $\mathtt{e}(\theta)$ be as in Lemma 10. Suppose by contradiction $\neg\, Winst(\theta, V, W)$. Then by Lemma 10, we have that any $x \in V$ can only occur in $\mathtt{e}(\theta)$ as a parameter, and only variables in Z can depend on some $x \in V$. Let M be a model of CET_L and ν a valuation for the variables in W which makes $\exists_V \exists_Z \mathtt{e}(\theta)$ true. Then $\forall_V \exists_Z \mathtt{e}(\theta)$ is also true under ν, hence $M \models \exists_W \forall_V \exists_Z \mathtt{e}(\theta))$, i.e. $M \models \neg(\forall_W \exists_V \forall_Z \neg \mathtt{e}(\theta))$. This contradicts the hypothesis, as $\mathtt{e}(\theta)$ is equivalent to $\mathtt{eqn}(\theta)$.

□

Note that the assumptions on the language are required only in the "only if" part of the proof.

This result will be the base for proving the completeness of the weak instantiation.

4.4 Correctness of NWI

In this section we prove soundness of negation as weak instantiation w.r.t. the completion semantics. It is required that the underlying language has some "new" function symbols (a symbol of arity > 0 or infinitely many constants). Note that using such an extended language is quite usual in logic programming (for example conf. [9]). We may view any Prolog implementation as using a language with infinitely many function symbols (of any arity).

The next lemma is a generalization of the "only if" part of Proposition 11 to the case of $n \geq 1$ substitutions.

Lemma 12. *Let $\theta_1, \ldots, \theta_n$ be some idempotent substitutions, V, W two disjoint sets of variables, and $Z = \bigcup_{i=1}^n var(\theta_i) \setminus (V \cup W)$. Let L be a language containing the function symbols occurring in $\theta_1, \ldots, \theta_n$. Assume that L contains infinitely many constant symbols, or alternatively one non-constant function symbol which does not occur in $\theta_1, \ldots, \theta_n$. Suppose that for $i = 1, \ldots, n$ θ_i instantiates V with respect to W. Then $CET_L \models \forall_W \exists_V \bigwedge_{i=1}^n \forall_Z \neg \mathtt{eqn}(\theta_i)$.*

Proof.

Assume that $\theta_1, \ldots, \theta_n$ instantiate V with respect to W. Let M be a model of CET_L and ν_0 an arbitrary valuation of the variables in W. We show how to extend ν_0 to a valuation ν of the variables in $V \cup W$, such that $\bigwedge_i \forall_Z \neg \mathtt{eqn}(\theta_i)$ is true under ν in M. This means that $M \models \forall_W \exists_V \bigwedge_i \forall_Z \neg \mathtt{eqn}(\theta_i)$.

Consider an extension ν of ν_0 to the variables of $W \cup V$. Consider an i $(1 \leq i \leq n)$. By Proposition 8, one of the following three cases occurs:

1. For some $x \in V$, $x\theta_i$ is not a variable. Then, as in the proof of Proposition 11, we obtain that a sufficient condition for

$$M \models_\nu \forall_Z \neg \mathtt{eqn}(\theta_i) \tag{1}$$

is that the value $\nu(x)$ is not in the range of the interpretation of any function symbol occurring in θ_i.

2. For some $x, x' \in V$, $x \neq x'$, $x\theta$ and $x'\theta$ are the same variable. As in the proof of Proposition 11 we obtain that (1) holds if the values of the variables of V in ν are distinct.
3. For some $x \in V$ and $y \in W$, $x\theta_i$ is a variable occurring in $y\theta_i$. As in the proof of Proposition 11, there exists a value α_i (in the domain of M) such that if equation $y = y\theta_i$ is true under ν then $\nu(x\theta_i) = \alpha_i$. Value α_i depends only on valuation ν_0 and term $y\theta_i$, and is unique. If $\nu(x\theta_i) \neq \alpha_i$ then (1).

It remains to construct a valuation ν such that the abovementioned sufficient conditions for (1) are satisfied for all i.

Let S be the set of all α_i's defined above. Let $k = |V|$. If L has an infinite set of constants then there exist distinct constants $c_1, \ldots, c_k$ that do not occur in $\theta_1, \ldots, \theta_n$ and their values in M are outside S (as S and the substitutions are finite). Take a ν as above with the values of the variables from V being the values of $c_1, \ldots, c_k$ in M. The sufficient conditions are satisfied and we obtain $M \models_\nu \bigwedge_{i=1}^n \forall_Z \neg \mathtt{eqn}(\theta_i)$.

If L has a one-argument function symbol f not occurring in $\theta_1, \ldots, \theta_n$ then consider a value β in the domain of M and values $\beta_j = f_M^j(\beta)$ (where f_M is the interpretation of f in M and $j > 0$). Taking a ν as above with the values of the variables from V being (distinct) $\beta_{j_1}, \ldots, \beta_{j_k}$, not occurring in S, we again obtain $M \models_\nu \bigwedge_{i=1}^n \forall_Z \neg \mathtt{eqn}(\theta_i)$. Generalization of this reasoning for the arity of f being greater than 1 is obvious.

□

Lemma 13. *Let Q be a node of an SLD-tree, and let $Q_1, \ldots, Q_n$, ($n \geq 0$) be its children. Then*

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

where θ_i are the mgu's corresponding to Q_i and $\overline{w}$ are those variables of $\theta_1, \ldots, \theta_n$, $Q_1, \ldots, Q_n$ that do not occur in Q.

This is a version of Lemma 15.3 of [11] and of Lemma 4.1 of [5], and the proof is similar.

In the proof of the soundness theorem we use the notion of a *cross-section* (or a frontier) of an SLD-tree.

Definition 14. A *cross-section* of an SLD-tree is any finite set S of nodes of the tree such that every successful or infinite branch has exactly one node in S.

Theorem 15. (Strong soundness of NWI) *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. Let L be an extension of the language of P and Q, which has infinitely many constants or has a function symbol occurring neither in P nor in Q. If there exists an SLD-tree for P and Q weakly instantiating $\overline{x}$ w.r.t. $\overline{y}$, then*

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

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

Proof.
By the hypothesis there exists a cross-section $\{Q_i\}_{i\in[1,n]}$ of the SLD-tree for P and Q such that for all $i \in [1,n]$ $\;Winst(\theta_i, \overline{x}, \overline{y})$. From Lemma 13 by simple induction we obtain

$$\mathrm{Comp}_L(P) \models \neg Q \leftrightarrow \bigwedge_{i=1}^{n} \forall\overline{w}\ \neg(\mathtt{eqn}(\theta_i) \wedge Q_i).$$

By adding quantifiers on both sides of $\leftrightarrow$, we obtain

$$\mathrm{Comp}_L(P) \models \forall\overline{y}\exists\overline{x}\forall\overline{z}\neg Q \leftrightarrow \forall\overline{y}\exists\overline{x} \bigwedge_{i=1}^{n} \forall\overline{z},\overline{w}\ \neg(\mathtt{eqn}(\theta_i) \wedge Q_i). \qquad (2)$$

Now by Lemma 12 $CET_L \models \forall\overline{y}\exists\overline{x} \bigwedge_{i=1}^{n} \forall\overline{w},\overline{z}\ \neg\mathtt{eqn}(\theta_i)$, hence the formula to the right hand side of $\leftrightarrow$ in (2) is a logical consequence of $\mathrm{Comp}_L(P)$. Thus

$$\mathrm{Comp}_L(P) \models \forall\overline{y}\exists\overline{x}\forall\overline{z}\neg Q.$$

□

Note that Theorem 5 (strong soundness of negation as instantiation) follows immediately from Theorem 15 with $\overline{y} = \emptyset$.

4.5 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 our proof we use the Lloyd-Topor transformation [13] and a constructive negation method (the SLDFA-resolution of [5]), which is sound and complete for the Kunen semantics.

Definition 16. 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

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

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 $\mathrm{Comp}(P')$).

Lemma 17. *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. $\overline{y}$.

Comment: SLDFA resolution uses constraints instead of substitutions. These constraints are arbitrary first order formulae with $=$ as the only predicate symbol. The goals are of the form $\sigma, \overline{L}$ where $\overline{L}$ is a sequence of literals and σ is a satisfiable (w.r.t. CET) constraint. For the details cf. [5].

Proof. Let us assume a fixed selection rule throughout the proof.

Any SLDFA failed tree for p is of the form

$$\begin{array}{c} p \\ | \\ \neg q(\overline{y}) \end{array}$$

provided that there exist some computed answers $\delta_1, \ldots, \delta_n$ for $q(\overline{y})$ such that

$$CET \models \delta_1 \vee \ldots \vee \delta_n \tag{3}$$

(cf. [5, Definition 3.9.4]). No other SLDFA failed trees for p exist. A constraint δ_i is a computed answer for $q(\overline{y})$ iff it is obtained from an SLDFA-refutation of the form

$$q(\overline{y}) \qquad \neg r(\overline{x}, \overline{y}) \qquad \sigma_i$$

where δ_i is $\exists\overline{x}\sigma_i$ and σ_i is a fail answer for $r(\overline{x}, \overline{y})$. The latter means that $freevar(\sigma_i) \subseteq \overline{x} \cup \overline{y}$ and there exists an SLDFA failed tree for $\sigma_i, r(\overline{x}, \overline{y})$ for a given selection rule. Note that an SLDFA failed tree exists for $\sigma_i, r(\overline{x}, \overline{y})$ iff an SLDFA failed tree exists for σ_i, Q. Also, if there exists such a tree for σ, Q and for σ', Q then an SLDFA failed tree exists for $(\sigma \vee \sigma'), Q$ (this follows from soundness and completeness of SLDFA-resolution or by a simple construction).

So there exists an SLDFA failed tree T for σ, Q, where σ is $\sigma_1, \ldots, \sigma_n$ and from (3) we obtain

$$CET \models \forall\overline{y}\exists\overline{x}\sigma. \tag{4}$$

Consider the fixed selection rule. If a failed tree for σ, Q exists then σ can be computed as described in Section 6 of [5]. Consider the pre-failed tree T' for Q. (It is the same tree as the SLD-tree for Q, it differs only by the form of the goals. In SLDFA-resolution, instead of applying an mgu to a goal, the corresponding conjunction of equations is added to the goal.) Then there exists a cross-section

$$S = \{\ \rho_1, \overline{L}^1;\ \ldots;\ \rho_m, \overline{L}^m\ \}$$

of T' such that each non failed branch of T' has a goal in S, and such that

$$\sigma \rightarrow \bigwedge_{i=1}^{m} \forall\,\overline{z}\,\overline{v}\,\neg\rho_i$$

where $\overline{v}$ are the variables of the tree not occurring in Q (cf. [5, Section 6]). From (4) we have CET $\models \forall\overline{y}\exists\overline{x}\bigwedge_i \forall\overline{z}\,\overline{v}\neg\rho_i$. This implies that CET $\models \forall\overline{y}\exists\overline{x}\forall\overline{z}\,\overline{v}\neg\rho_i$, for $i = 1, \ldots, m$.

Let T'' be the SLD-tree for Q. Let θ_i be the accumulated substitution for the node of T'' corresponding to the node $\rho_i, \overline{L}^i$ of T' $(i = 1, \ldots, m)$. Then $\mathtt{eqn}(\theta_i)$ is equivalent to ρ_i w.r.t. CET (cf. eg. [6, 14]).

By Proposition 11, θ_i instantiates $\overline{x}$ w.r.t. $\overline{y}$. Then T'' instantiates $\overline{x}$ w.r.t. $\overline{y}$, as in every non failed branch of T'' there is a node corresponding to a node from S.

□

Theorem 18. (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 $Comp_L(P) \models \forall\overline{y}\exists\overline{x}\forall\overline{z}\neg Q$, 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 $\mathrm{Comp}_L(P') \models \neg p$ iff $\mathrm{Comp}_L(P') \models_3 \neg p$ (as P' is call consistent and query $\neg p$ is strict with respect to P'). If $\mathrm{Comp}_L(P') \models_3 \neg p$, then $\mathrm{Comp}_{L'}(P') \models_3 \neg p$ for any extension L' of L. Consider an L' with an infinite set of function symbols.

By the completeness of the SLDFA-resolution (cf. [5, Theorem 5.1]), if $\mathrm{Comp}_{L'}(P') \models_3 \neg p$ then for any fair selection rule there exists an SLDFA finitely failed tree for p. Then, by Lemma 17, 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 $\exists\overline{x}\forall\overline{y}\neg Q$ (where $\overline{y}$ are the remaining variables of Q; cf. Theorem 5). Then, we introduced a new notion of weak instantiation. Even when $\exists\overline{x}\forall\overline{y}\neg Q$ does not hold, weak instantiation makes it possible to derive formulae of the form $\exists\overline{x}\neg Q$ (or, equivalently, $\forall\overline{y}\exists\overline{x}\neg Q$) and of the form $\forall\overline{y}_1\exists\overline{x}\forall\overline{y}_2\neg Q$, (where $\overline{y}_1\overline{y}_2$ is $\overline{y}$). 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.

As shown in Section 4.5, the formulae derivable by negation as weak instantiation can also be computed using constructive negation and a program transformation. However, the latter solution, although more general, is more complicated and expensive. For instance, constructive negation uses quantified equational formulae instead of substitutions. In contrast, (weak) instantiation refers to the substitutions obtained by the standard SLD-resolution.

In line with the approach of [7, 8], the weak instantiation can be seen as one of the so-called *observable properties* of a program P. The set of atoms A for which a $\forall\exists\forall$-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. We expect to employ (a suitable modification of) the immediate consequence operator, T_c, of the C-semantics [7, 8].

In this paper we dealt with definite programs only. In the context of this work it is natural to consider programs and goals with negation. Definition 9 of a weakly instantiating tree can be applied to SLDNF-trees; it seems that the soundness proof from Section 4.4 can be easily generalized for this case. The semantics of interest is, as in the case of SLDNF-resolution [10, 15, 6], the three-valued completion semantics of Kunen [9]. Another generalization is to permit sub-formulae of the form $\forall\overline{y}\exists\overline{x}\forall\overline{z}\neg Q$ in programs and goals. In this case negative literals are to be treated by negation as failure and quantified negative literals by negation as instantiation. We believe that such operational semantics is sound w.r.t. Kunen's semantics. We expect a completeness property similar to those in [10, 15] or [6].

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, 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. W. Drabent. Completeness of SLDNF-resolution for Non-Floundering Queries. *Journal of Logic Programming*, 27(2):89–106, May 1996.
7. 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.
8. 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.
9. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.
10. K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7(3):231–245, 1989.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
12. J. L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.
13. J. W. Lloyd and R. W. Topor. Making Prolog more Expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.
14. C. Palamidessi. Algebraic properties of idempotent substitutions. In M. S. Paterson, editor, *Proc. of the 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 386–399. Springer-Verlag, Berlin, 1990.
15. R. F. Stärk. Input/output dependencies of normal logic programs. *Journal of Logic and Computation*, 4:249–262, 1994.