Transforming normal logic programs to constraint logic programs

Kanchana Kanchanasut* and Peter J. Stuckey

Abstract


A normal logic program \( P \) is transformed to another \( P' \) by a bottom-up computation on the positive component of \( P \) while the negative counterpart is left untouched. The declarative semantics of \( P \) is given by the completion of \( P' \). The completed predicate definitions in \( P' \), if they do not contain local variables, can be used as a basis for expanding each negated atom in the bodies of \( P'' \). We show that for a class of programs where every negative subgoal can be expanded, the resultant program \( P' \) is a definite logic program with equality and disequality constraints. If the program falls outside this class, the resultant program may be executed using constructive negation.

Our proposed scheme provides an effective sound and complete query-answering system for the well-founded models of a class of programs whose positive part has a finite \( P'' \) and whose clauses satisfy the positive groundedness property defined herein.

1. Introduction

In logic programming, when negation is introduced in queries, the meaning of a program is based upon the Clark completion [7] of the original program, which intuitively reads implication signs as equivalence signs. SLDNF-resolution [17] provides a sound and complete proof procedure for definite logic programs, but when negated atoms are allowed in the body of a program clause, the completeness of SLDNF is lost. In addition, for the case of two-valued logic, the Clark completion
$\text{comp}(P)$ of a program $P$ can be inconsistent. Fitting [11] and Kunen [16] propose three-valued interpretations of $\text{comp}(P)$ as models of logic programs which can give meanings in three-valued logic to programs even when they are inconsistent in two-valued logic. However, $\text{comp}(P)$ also has other drawbacks even without the presence of negated atoms in the program, as shown in [25, 33], where there are infinite looping SLDNF-derivations for $P$.

Recent approaches to giving declarative semantics, that capture the intended meaning of normal programs tend to adopt the stable model semantics [12] as the natural two-valued semantics for normal programs. For three-valued interpretation, the well-founded semantics [33] has received much attention. As in the Fitting semantics, both approaches only consider Herbrand models. The well-founded semantics coincides with the stable model semantics for cases where programs have two-valued well-founded models. Ross [26] gave a procedural semantics which is sound and complete with respect to the well-founded model of nonfloundering programs. But his procedural semantics is not effective and at this stage we are not aware of any effective operational model for well-founded semantics for programs beyond those which are function-free (DATALOG).

We propose an operational model for a query-answering system for normal programs with respect to well-founded model semantics which is based on transforming the original program to a new program upon which the query is applied. It uses both the bottom-up and top-down computational models; the bottom-up computation during a partial evaluation phase and a top-down SLD-resolution at run-time. The idea is to eliminate negated atoms so that SLD is sound and complete with improved run-time performance. We first apply a program transformation which evaluates all the positive atoms in the bodies of the program. Each positive atom in the body of the program gets expanded until there are no more positive atoms left in the bodies of clauses. In other words, we generalize a $T_p$ operator defined in [8] to nonground programs. The transformed program $P^o$ consists of clauses whose body consists only of equality constraints and negated atoms. We then take the Clark completion of $P^o$ and use its negated predicate definitions to expand the negated subgoals in $P^o$. There are negated subgoals whose negated definitions cannot be used for expansion due to the appearances of local variables. For cases where a program has all its negative atoms expanded, its transformed version will be in a form of a constraint logic program $P'$ over the Herbrand universe with equality and disequality constraints and only positive atoms in the body. This class of constraints has been shown to be decidable [19] and several implementations of a constraint logic programming (CLP) system for this structure exist, for example, CLP(FT) [28] and Chan and Wallace's [6] meta-interpreter for handling negated goals written in SEPIA [21].

An example below demonstrates the two phases of our transformation. It shows how our transformation can overcome the floundering problem associated with the SLDNF-resolution.
Example 1.1. Let $P$ be

\[
q(x) \leftarrow \neg p(x)
\]

\[
p(a)
\]

\[
p(x) \leftarrow s(y) \land \neg r(y, x)
\]

\[
s(b)
\]

\[
r(b, c)
\]

Given a query $\leftarrow q(x)$, Prolog will fail and SLDNF flounders. $P^\alpha$ is

\[
q(x) \leftarrow \neg p(x)
\]

\[
p(a)
\]

\[
p(x) \leftarrow \neg r(b, x)
\]

\[
s(b)
\]

\[
r(b, c)
\]

\[\text{comp}(P^\alpha)\] is

\[
q(x) \leftarrow \neg p(x)
\]

\[
p(x) \leftarrow x = a \lor r(b, x)
\]

\[
s(x) \leftarrow x = b
\]

\[
r(z, x) \leftarrow z = b \land x = c
\]

Expanding all the negated atoms by their negated definitions, $P'$ becomes

\[
q(x) \leftarrow \neg p(x)
\]

\[
p(a)
\]

\[
p(x) \leftarrow \neg r(b, x)
\]

\[
s(b)
\]

\[
r(b, c)
\]

\[
\neg p(x) \leftarrow x = a \lor r(b, x)
\]

\[
\neg r(b, x) \leftarrow x \neq c
\]

Now if we ask $\leftarrow q(x)$, an SLD-derivation on $P'$ will succeed with the answer $x = c$.

Before describing the transformation, we give a brief review of the declarative semantics for normal programs in Section 2. In Section 3, we formally describe our transformation. We then describe the class of programs for which our system provides
a sound and complete query evaluation procedure in Section 4. We describe tech-
niques to overcome the theoretical limitations which makes our scheme applicable to
many programs in practice.

2. Background

2.1. Terminologies

Throughout this paper, a program means a normal logic program as defined in [17],
where negative literals can occur in the body of any clause, unless otherwise indicated.
We restate the definition here for readability.

We assume that the language $L$ has a countable alphabet of constants, function
symbols and predicate symbols and a countable number of variables.

A term is defined as follows:

- a variable is a term,
- a constant is a term,
- a function symbol with terms as arguments is a term.

An atom is an $n$-ary predicate with terms as arguments. A literal is an atom or the
negation of an atom.

A normal program clause is a clause of the form

$$A \leftarrow L_1, \ldots, L_n,$$

where $A$ is an atom and $L_i$ are (positive or negative) literals.

A normal program is a set of normal program clauses.

From now on we refer to normal programs as programs and we use $P$ to denote an
arbitrary, but fixed program.

The Herbrand universe of a program $P$, denoted by $U_P$, is the set of all variable-free
terms that may be formed from the constants and function symbols appearing in $P$. If
there are no constants in $P$ then we treat $P$ as if it had a single extra constant symbol.

The Herbrand base of a program $P$, denoted by $HB_P$, is the set of all variable-free
atoms that may be formed from the constants and function symbols appearing in $P$.

The Herbrand instantiation [33] of a logic program, denoted by $G_P$, consists of the set
of ground clauses

$$A\theta \leftarrow B_1\theta, \ldots, B_n\theta$$

for each clause in $P$ of the form

$$A \leftarrow B_1, \ldots, B_n$$

and ground substitution $\theta$.

A constraint logic programming [13] system CLP($\mathcal{D}$) exists in the context of
a particular structure $\mathcal{D}$ which determines the meaning of the function and relation
symbols. Constraints in the structure are relations upon terms of the structure. An
(atomic) constraint takes the form $r(t_1, ..., t_n)$, where $r$ is an $n$-ary relation symbol defined by $\mathcal{D}$. For example, the following are constraints in the domain of finite trees (which is the only domain we are interested in for the purposes of this paper):

$$x = f(y, z), \quad \forall z(y \neq g(z, a)), \quad \forall u, v(z \neq h(y, u, v)).$$

Constraint logic programs differ from logic programs by allowing constraints in bodies of rules and goals. A (definite) constraint logic program is, thus, a finite set of rules of the form

$$A \leftarrow C, B_1, ..., B_n,$$

where $A$ and $B_i$, $1 \leq i \leq n$, are atoms, and $C$ is a conjunction of constraints. We shall also come across normal constraint logic programs in the paper, where $B_i$, $1 \leq i \leq n$ are literals.

An (SLD) derivation step in CLP($\mathcal{D}$) for a definite program $P$ takes a goal

$$\leftarrow C, B_1, ..., B_i, ..., B_n,$$

where $B_i$ is a selected atom, and clause

$$B \leftarrow C', D_1, ..., D_m$$

from $P$, where $\{B = B_i\} \land C \land C'$ is satisfiable in domain $\mathcal{D}$, and results in the goal

$$\leftarrow \{B = B_i\} \land C \land C', B_1, ..., B_{i-1}, D_1, ..., D_m, B_{i+1}, ..., B_n,$$

where $\{B = B_i\}$ is a set of constraints equating the arguments of atoms $B$ and $B_i$. A computation rule determines at each step which atom (if any) is selected. A derivation tree for goal $G$ (for some computation rule) is the tree rooted at $G$ where the children of each node $G'$ are the goals obtained from a derivation step applied to $G'$. A derivation of $G$ is a branch in the search tree of $G$. A derivation is successful if it is finite and its last goal contains no atoms. A derivation is finitely failed if it is finite but not successful. The success set of a program $P$, denoted by $SS(P)$, is the set of all ground atoms which have successful derivations. The finite failure set of a program $P$, $FF(P)$, is the set of all ground atoms for which all derivations are finitely failed. For more details see [13].

We can define the Herbrand instantiation, $G_P$, of a normal constraint logic program $P$ when the structure $\mathcal{D}$ is that of the finite trees. $G_P$ consists of the set of ground clauses

$$A \theta \leftarrow B_1 \theta, ..., B_n \theta$$

for each clause in $P$ of the form

$$A \leftarrow C, B_1, ..., B_n$$

and ground substitution $\theta$ such that $C \theta$ is true in $\mathcal{D}$. 
2.2. Semantics of normal programs

An early approach to understanding negation in normal logic programs is by the program completion introduced by Clark [7]. The meaning of a program is given by its completed definition which is simply a first-order formula. The corresponding proof procedure for this scheme is SLDNF-resolution which is SLD augmented by a nonmonotonic rule called negation as failure. Problems related to this approach are: there are programs whose Clark completion is inconsistent and there are also programs with consistent Clark completion which have unintuitive models.

Another approach to the question of negation is to identify a model that a program is “intended” to mean. This approach has led to the introduction of classes of programs for which unique “intended” models exist, for example, stratified programs with unique iterated least models [1] and locally stratified programs with unique perfect models [24]. Stratified programs are ones where recursion through negation is forbidden. Locally stratified programs are programs whose stratification requirement is defined based on priority relationship on ground atoms instead of the predicate symbols as for the case of stratified programs.

Without the stratification restrictions, van Gelder et al. [33] has given a three-valued well-founded semantics which can associate meaning to all logic programs. It differs from the three-valued Fitting semantics [11] in that atoms appearing in a positive cycle of dependencies are considered false. Each program is associated with a unique well-founded model. Under certain conditions, the three-valued well-founded semantics can yield two-valued models in which case they are equivalent to the stable models proposed by Gelfond and Lifschitz [12]. Whenever they coincide, it is obvious that the stable model is unique as well. However, there are programs with unique stable models which do not have two-valued well-founded models. In [9], a sufficient condition for the existence of unique stable models called sufficient stratification is defined, which also ensures a two-valued well-founded model.

In this paper, our transformation preserves the well-founded semantics of the original program \( P \). In the case where \( P \) is sufficiently stratified we can also show that our soundness and completeness results apply as well to unique stable models.

3. Transformation of \( P \) to \( P' \)

We divide our transformation into two phases: the bottom-up computation on positive atoms (\( P \) to \( P'' \)) and the expansion of the negative atoms (\( P'' \) to \( P' \)).

3.1. Positive transformation

Our transformation from \( P \) to \( P'' \) is a generalization of an operator \( T_P \) of [8] (also independently defined in [3]) which operates on ground normal program clauses mapping one ground quasi-interpretation to another. The \( T_P \) operator in [8] is
virtually identical to the $T_p$ operator as defined for definite programs [32] but operates over the domain of ground quasi-interpretations instead of the Herbrand interpretations. Ground quasi-interpretation contains a set of ground clauses whose body consist only of negated atoms. For the case of definite programs, they are identical to the Herbrand interpretations which makes the two $T_p$ operators coincide precisely.

Instead of operating on ground clauses, our transformation operates on the program clauses themselves. We first reproduce the definitions of $T_p$ from [8], together with its important properties.

**Definition.** A ground quasi-interpretation\(^1\) for $P$ is a set of ground program clauses of the form $A \leftarrow \neg B_1, \ldots, \neg B_n$, where $A, B_i$ are ground atoms in $HB_p$.

The set of all ground quasi-interpretations for $P$ is denoted by $GQI_p$. It is clear that $GPI_p$ is a complete lattice w.r.t. set inclusion.

Let $C$ be the ground clause $A \leftarrow \neg B_1, \ldots, \neg B_n$, $A_1, \ldots, A_m$ with $n \geq 0$, $m \geq 0$ and let $C_i$ be ground clauses $A_i \leftarrow \neg B_{i1}, \ldots, \neg B_{in}$, with $1 \leq i \leq m$ and $n_i \geq 0$. Then $T_c(C_1, \ldots, C_m)$ is the following clause

$$A \leftarrow \neg B_1, \ldots, \neg B_n, \neg B_{i1}, \ldots, \neg B_{in}, \ldots, \neg B_{m1}, \ldots, \neg B_{mn}.$$  

We now introduce the transformation $T_p$ on ground quasi-interpretations. Let $G_P$ be the Herbrand instantiation of $P$, and $GQ$ be a ground quasi-interpretation for $P$:

$$T_p : GQI_p \rightarrow GQI_p.$$  

**Theorem 3.1** (Dung and Kanchanasut [8]). $T_p$ is continuous.

We define the semantic kernel\(^2\) $SK$ of $P$ as follows:

Let

$$SK_n(P) = T_p^n(\emptyset),$$

$$SK(P) = \bigcup_{n \geq 1} SK_n(P)$$ (The least fixpoint of $T_p$)

Let $p$ be a predicate of $P$ and $\{C_1, C_2, \ldots\}$ be the set of clauses in $SK(P)$ whose heads are atoms with predicate symbol $p$. $C_i$ is a clause of the form $p(T) \leftarrow \neg B_1, \ldots, \neg B_n$. Then the Clark completion of $p$ is

$$\forall(p(x) \leftarrow E_1 \lor \cdots \lor E_m \lor \cdots).$$

\(^1\) In [8] a ground quasi-interpretation is called a quasi-interpretation.

\(^2\) It is denoted as $LFP$ in [8].
where the right-hand side is a (possibly infinite) disjunction. Every $E_i$ is of the following form:

$$\tilde{x} = \bigwedge i \neg B_i \land \cdots \land \neg B_n.$$  

An infinite disjunction is true w.r.t. an interpretation if one or more of its elements is true w.r.t. this interpretation. Clark's completion of $SK(P)$, which is called the fixpoint completion of $P$, denoted by $fixcomp(P)$, is a collection of the completed definitions of predicates of $P$ together with Clark's equality theory.

**Theorem 3.2** (Dung and Kanchanasut [8]). (a) Every Herbrand model of $P$ is a model of $SK(P)$.  
(b) Every Herbrand model of the fixpoint completion of $P$ $fixcomp(P)$ is a model of the Clark's completion of $P$, $comp(P)$.

**Note:** In general, the reverse of part (b) of Theorem 3.2, does not hold.

Other important properties of $SK(P)$ that are discussed in [8, 9] are the following:

A two-valued interpretation of $SK(P)$ gives the stable model semantics while a three-valued interpretation gives the well-founded model semantics.

**Definition.** $P$ is sufficiently stratified if the priority relation on the ground atoms of $SK(P)$ is well-founded. That is to say $SK(P)$ is locally stratified.

**Theorem 3.3** (Dung and Kanchanasut [8]). (a) Every Herbrand model of $fixcomp(P)$ is a stable model of $P$ and vice versa.
(b) If $P$ is sufficiently stratified then there exists a unique stable model for $P$.

The next theorem relates the three-valued interpretation of $fixcomp(P)$ to the well-founded semantics. We follow van Gelder et al. [33] in representing three-valued Herbrand interpretations of a program $P$ as consistent sets of literals whose atoms are from $HB_P$. Further, we define $\neg \cdot A$ to be the literal $\neg A$ if $A$ is an atom, or the literal $B$ if $A$ is the literal $\neg B$ for some atom $B$. The Fitting semantics of a normal program (or normal constraint program) $P$ is defined as the least fixpoint of the monotonic operator $\Phi_p$ that maps three-valued interpretations to three-valued interpretations defined as follows.

**Definition.**

$$\Phi_p(I) = \{ A \mid \text{there exists a (ground) clause in } G_P \text{ of the form}$$

$$A \leftarrow L_1, \ldots, L_n \text{ such that } \{ L_1, \ldots, L_n \} \subseteq I \}$$

$$\cup \{ \neg \cdot A \mid \text{for each (ground) clause in } G_P \text{ of the form}$$

$$A \leftarrow L_1, \ldots, L_n \exists j \leq n, \neg \cdot L_j \in I \}.$$
The Fitting semantics gives a three-valued interpretation of the completion of a program, hereafter called Fitting models.

**Theorem 3.4** (Dung and Kanchanasut [9]). *The Fitting model of fixcomp(*P*) is the well-founded model of *P*.*

We now define a transformation \( \mathcal{F}_P \) which is a generalization of \( T_P \) to nonground atoms. This bears some similarity to the fixpoint operator of the nonground semantics for definite programs [10]. It will be shown that the nonground semantics kernels share the above properties of the ground ones. We extend the definition of ground quasi-interpretations to a general case as follows.

**Definition.** A *negative clause* \( NC \) is a clause of the form
\[
A \leftarrow \neg B_1, \ldots, \neg B_n.
\]

Let \( [NC] \) be the set of ground instances \( NC\theta \) of a negative clause \( NC \)
\[
A\theta \leftarrow \neg B_1\theta, \ldots, \neg B_n\theta.
\]

We extend the \([\;]\) notation to sets of negative clauses in the obvious manner.

**Definition.** A *quasi-interpretation* for \( P \) is a set of negative clauses over the alphabet of \( P \).

Let the set of all quasi-interpretations for \( P \) be denoted by \( QI_P \). We define the following relation \( \preceq \) on \( QI_P \).

**Definition.** \( A \preceq B \), where \( A \) and \( B \in QI_P \) iff \( [A] \subseteq [B] \).

Clearly, \( A \preceq B \) and \( B \preceq A \) iff \( [A] = [B] \). When this is the case, we say that \( A \) and \( B \) belong to the same equivalence class. It is easy to show the following lemma.

**Lemma 3.5.** If \( \bar{X} \) is a directed subset of \( QI_P/\sim \) (set of equivalence classes of \( QI_P \)) then \( [\text{lub} \bar{X}] = \text{lub} [\bar{X}] \).

**Definition.** Let \( \gamma : QI_P/\sim \to GQI_P \) be defined by \( \gamma(S) = [S] \).

**Lemma 3.6.** \( \gamma \) is a bijection.

**Corollary 3.7.** \( QI_P/\sim \) with respect to \( \preceq \) is a complete lattice.

Our operator \( \mathcal{F}_P \) is defined on \( QI_P \) in the same manner as \( T_P \) on \( GQI_P \). Let \( C \) be a clause of \( P \) of the following form:
\[
A \leftarrow \neg B_1, \ldots, \neg B_n, A_1, \ldots, A_m.
\]
with $n$ and $m \geq 0$.

Let $NC_i$ be negative clauses of the following form:

$$A_i' \leftarrow \neg B_1, \ldots, \neg B_{n_i}.$$ 

with $1 \leq i \leq m$ and $n_i \geq 0$.

Let $\theta$ be a most general substitution such that $A_1 \theta = A_1', \ldots, A_m \theta = A_m'$. If no such $\theta$ exists then $\mathcal{T}_c(NC_1, \ldots, NC_m)$ is the empty clause, otherwise $\mathcal{T}_c(NC_1, \ldots, NC_m)$ is the following clause:

$$A_1 \theta \leftarrow \neg B_1 \theta, \ldots, \neg B_n \theta, \neg B_1 \theta, \ldots, \neg B_{n_i} \theta, \ldots, \neg B_m \theta, \ldots, \neg B_{m_{n_i}} \theta.$$ 

We now formally define the transformation $\mathcal{T}_p$ on quasi-interpretations.

**Definition.**

$$\mathcal{T}_p : QI \rightarrow QI_p,$$

$$\mathcal{T}_p(Q) = \{ \mathcal{T}_c(NC_1, \ldots, NC_m) | C \in P \text{ and } NC_i, 1 \leq i \leq m \}$$

are renamed apart copies of elements of $Q$.

**Lemma 3.8.** For any quasi-interpretation $I$, $T_p([I]) = [\mathcal{T}_p(I)]$.

**Proof.** Clearly, $[\mathcal{T}_c(NC_1, \ldots, NC_m)] = \cup C'([NC_1, \ldots, NC_m])$, where $C'$ ranges over ground instances of $C$. \qed

**Lemma 3.9.** $\mathcal{T}_p$ is continuous on $QI_p$.

**Proof.** For any directed subset $\bar{X}$ of $QI_p$,

$$[[\text{lab } \mathcal{T}_p(\bar{X})]] = \text{lab } [[\mathcal{T}_p(\bar{X})]] = \text{lab } (T_p([\bar{X}])) = T_p(\text{lab }([\bar{X}])) = \mathcal{T}_p(\text{lab }([\bar{X}]))$$

Since $\gamma$ is a bijection, $\mathcal{T}_p(\text{lab }([\bar{X}])) = \text{lab } (\mathcal{T}_p(\bar{X}))$. \qed

Since $\mathcal{T}_p$ is continuous, $\mathcal{T}_p^\omega(\emptyset)$ reaches the least fixpoint at or before $\omega$ steps. We denote the program given by the least fixpoint of $\mathcal{T}_p^\omega$ by $P^\omega$. $P^\omega$ is the result of the first phase of our transformation, and corresponds to evaluating all the positive information in the program. We can see the close relationship between $P^\omega$ and the original program $P$ from the following theorem.
Proposition 3.10. Every model of $comp(P^\omega)$ is a model of $comp(P)$.

Proof. Let $head(C)$ denote the head of a clause $C$ and $body(C)$ denote its body. Let $I$ be a model of $comp(P^\omega)$.

Consider a clause $C' \in P^\omega$ of the form
\[ A \leftarrow \neg B_1 \theta, \ldots, \neg B_n \theta, \neg B_{i_1} \theta, \ldots, \neg B_{i_n} \theta, \ldots, \neg B_m \theta, \ldots, \neg B_{m_n} \theta, \]
where $C' \in T_C(C_1, \ldots, C_m)$, where $C \in P$ is of the form
\[ A \leftarrow \neg B_1, \ldots, \neg B_n, A_1, \ldots, A_m, \]
and $C_i \in P^\omega$ is of the form
\[ A_i \leftarrow \neg B_{i_1}, \ldots, B_{i_n}, \]
and $\theta$ is the mgu of $A_1 = A_1' \land \cdots \land A_m = A_m'$. Let $\alpha$ and $\gamma$ be assignments of elements in the domain of $I$ to variables.

(a) First we show that $I$ models $P$. Suppose $I \not= body(C)\gamma$; then $I \models A \alpha \gamma$ and, since $I \models comp(P^\omega)$, there exists a clause $C_i \in P^\omega$ such that $I \models \neg B_i \gamma, \ldots, \neg B_m \gamma$. Thus,
\[ I \models \neg B_1 \gamma, \ldots, \neg B_n \gamma, \neg B_{i_1} \gamma, \ldots, \neg B_{i_n} \gamma, \ldots, \neg B_m \gamma, \ldots, \neg B_{m_n} \gamma, \]
Since $I$ models Clark’s axioms, $A_1 = A_1' \land \cdots \land A_m = A_m'$ is unifiable; thus, there exists $C' \in T_C(C_1, \ldots, C_m)$ as above and its instance $\alpha$ such that $body(C)\gamma = body(C')\alpha$. Hence, $I \models A \gamma = A \alpha \gamma$.

(b) Suppose $I \models A \gamma$; then, since $I \models comp(P^\omega)$, there exists $C' \in P^\omega$ as above and its instance $\alpha$ such that $A_1 = A_1 \alpha \gamma$. Now $I \models body(C')\alpha$ and, since $I \models C_i$ and $I \models (\neg B_i \theta, \ldots, \neg B_m \theta)\alpha$, $I \models A_1 \theta \alpha$. Hence, $I \models body(C)\theta \alpha$, i.e. $I \models body(C)\gamma$. \[ \square \]

Since $[P^\omega] = SK(P)$, its Clark completion yields the same set of Herbrand models as $SK(P)$. It follows from Theorem 3.3(a) that the Herbrand models of $comp(P^\omega)$ are equivalent to the stable models [12] of $P$. Similarly, it follows from Theorem 3.4 that the Fitting model of $P^\omega$ is the well-founded model of $P$.

Lemma 3.11. Every Herbrand model of $comp(P^\omega)$ is a stable model of $P$.

Lemma 3.12. The least fixpoint of $\Phi_{p^\omega}$ is the well-founded model of $P$.

In general, $P^\omega$ will contain infinitely many clauses, but if $\mathcal{F}_P(\theta)$ reaches its fixpoint in a finite number of steps then $P^\omega$ is finite, since each step only produces finitely many negative clauses. From now on we assume that $P^\omega$ is finite; this assumption is essential for the application of the negation operation $N(\ )$ defined in Section 3.2. Obviously, programs $P$ whose Herbrand universe is made up entirely of constants (DATALOG programs) have finite $P^\omega$. In Section 4.2 we discuss some other techniques of avoiding building infinite $P^\omega$. 

3.2. Negative expansion

$P^\alpha$ consists of clauses which only have negated atoms in the body. With the aim of obtaining a definite constraint program $P'$, we introduce the following program transformation $N(\cdot)$ that expands negated atoms and replaces them where possible by positive atoms, and define $P' = N(P^\alpha)$. The technique is basically a version of that proposed by Sato and Tamaki [27] for transforming definite programs to their dual programs (see also [2]). Their technique of taking the negation of a completed definition can be applied to programs with negated atoms (as Chan and Wallace did in [6]), and is also the method used in the constructive negation operational scheme of Chan [5].

We define our transformation over arbitrary programs. Given an arbitrary program $P$, we construct a normal constraint logic program $N(P)$ over the structure of the Herbrand Universe of $P$, involving equality and quantified disequality constraints. The approach we outline can be considerably improved, by removing redundancy from the resulting clauses, but for the simplicity of proofs we consider a very straightforward form.

Consider the Clark completion of $P$. For each predicate $p$ in $P$ where the clauses in $P$ are of the form

$$p(\bar{s}) \leftarrow B_i, \quad 1 \leq i \leq n.$$ 

Let $E_i = \bar{x} = \bar{s}_i \wedge B_i$. The completed definition of $p$ is then

$$p(\bar{x}) \leftarrow \exists Y_1 E_1 \lor \exists Y_2 E_2 \lor \cdots \lor \exists Y_n E_n,$$

where $Y_i$ are the variables in $E_i$ not in $\bar{x}$.

We call a variable $y \in Y_i$ a local variable [22] if it does not occur in $\bar{x} = \bar{s}_i$. If no variables in each $E_i$ of $p$ are local, we say $p$ has a local-variable-free definition. For the special case that there are no clauses in $P$ with predicate $p$ in the head (i.e. $\text{comp}(P)$ includes $\forall \bar{x} \neg p(\bar{x})$) we also say that $p$ has a locally variable-free definition. Define $\text{Free}(P)$ as the set of predicates which have a locally variable-free definition in $\text{comp}(P)$.

We now are able to define the operation $\text{Not}(p)$, where $p$ is a predicate with a local-variable-free definition, that returns a set of clauses defining the predicate $\neg p$ which we shall show is equivalent to the negation of $p$. This operation is essentially identical to the procedure used by Chan [5] to negate answer goals in his constructive negation derivation procedure. The operation is based on the following equivalence implied by the Clark’s equality theory.

$$\forall \bar{x} (\neg \exists \bar{y}, \bar{z} (x = s \wedge Q) \leftrightarrow \forall \bar{y} (x \neq s) \lor \exists \bar{y} (x = s \wedge \neg \exists \bar{z} Q),$$

where $\bar{y}$ are the variables in $s$ not in $\bar{x}$ and the variables of $Q$ are a subset of $\{\bar{x}, \bar{y}, \bar{z}\}$.

Consider the complete definition of $p$ to be

$$p(\bar{x}) \leftrightarrow \exists Y_1 E_1 \lor \exists Y_2 E_2 \lor \cdots \lor \exists Y_n E_n.$$
where $Y_i$ are the variables in $E_i$ not in $\bar{x}$. A disjunct takes the form

$$E_i \equiv x_1 = s_i^1 \land \cdots \land x_n = s_i^n \land B_i,$$

where $Y_i^j$ are the variables in $s_i^j$ not in $\bar{x}$ and any of $s_i^k$ for $1 \leq k < j$. Negating the disjunct using the above property, we obtain

$$\forall Y_1^1 x_1 \neq s_1^1$$

$$\forall \exists Y_1^1 x_1 = s_1^1 \land \forall Y_1^n x_2 \neq s_1^2$$

$$\vdots$$

$$\forall \exists Y_1^1 x_1 = s_1^1 \land \cdots \land \exists Y_1^n x_n = s_1^n \land \neg B_i.$$

Note that the formula $\neg B_i$ does not involve any quantifiers, since we have restricted ourselves to local-variable-free definitions; hence, we can place it in disjunctive normal form without creating universal quantifiers. If we negate each disjunct and place the conjunction in disjunctive normal form then we obtain a formula of the form

$$\exists Y_1^1 D_1 \lor \cdots \lor \exists Y_r D_r$$

and we can define the clauses of $\text{Not}(p)$ to be $\text{not}(\bar{x}) \leftarrow D_i, 1 \leq i \leq r$. When $p$ does not appear in the head of any clause in $P$ we define $\text{Not}(p)$ as the single clause $\text{not}(\bar{x}) \leftarrow$.

**Example 3.13.** Let the completed definition of $p(x, y)$ be

$$p(x, y) \leftarrow \neg x \rightarrow y$$

$$\forall \exists x = f(z) \land y = g(z) \land \neg p(y, z).$$

Its negated definition, $\text{Not}(p)$ in clausal form is

$$\text{not}(x, y) \leftarrow \neg x \neq y \land \forall x \neq f(z)$$

$$\text{not}(x, y) \leftarrow \neg x \neq y \land x = f(z) \land y \neq g(z)$$

$$\text{not}(x, y) \leftarrow \neg x \neq y \land x = f(x) \land y = g(z) \land p(y, z)$$

Note that the only explicit quantification occurs in quantified disequalities.

We show by example that when the definition of $p$ is not local-variable-free then $\text{Not}(p)$ may, in general, have no clausal form.

**Example 3.14.** Let $P$ be

$$p(x) \leftarrow \neg r(x, y) \land \neg s(x, y)$$

Then the negation of definition of $p$ is

$$\neg p(x) \leftarrow \forall y (r(x, y) \lor s(x, y))$$

and the right-hand side cannot be written in a clausal form.
If we allowed the negation of definitions which are not local-variable-free and used the transformation of Lloyd and Topor [18] for removing quantifiers from the final programs, we obtain clauses similar to those from which we started. Hence, we do not attempt to negate such definitions. As we will see this means we cannot eliminate all negative literals from a program.

Define $NOT(P)$ as the set of clauses $\{not(p)|p\in Free(P)\}$. Define the operation $Sub(S, C)$ that takes a set of predicates $S$ and a clause $C$ and returns the clause $C'$ where each negative literal $\neg p(\tilde{s})$ appearing in the body, where $p\in S$, is replaced by the literal $notp(\tilde{s})$.

The negation operation $N(\cdot)$ applied to a program $P$ is defined as the program $N(P) \equiv \{Sub(Free(P), C)|C\in P \cup NOT(P)\}$. That is, we take the program $P$ and add the negative clauses, and then substitute out the negative literals that have a positive representation. If every predicate in $P$ is local-variable-free then, clearly, $N(P)$ is a definite program, since all negative literals are substituted for. Otherwise, some negative literals remain.

The program $P'$ is the result of applying the negation transformation to the program $P^\omega$ obtained from the original program $P$ using the positive transformation, i.e. $P' = N(P^\omega)$.

**Example 3.15.** Let $P$ be the program

\[
p(x, y) \leftarrow x = y
\]

\[
p(x, y) \leftarrow x = f(z) \land y = g(z) \land \neg p(y, z)
\]

$Free(P)$ consists of the predicate symbol $p$, and the negated version of $P$, $NOT(P)$ in clausal form is

\[
notp(x, y) \leftarrow x \neq y \land \forall z (x \neq f(z))
\]

\[
notp(x, y) \leftarrow x \neq y \land x = f(z) \land y \neq g(z)
\]

\[
notp(x, y) \leftarrow x \neq y \land x = f(z) \land y = g(z) \land p(y, z)
\]

After substitution, we obtain the definite constraint program $N(P)$

\[
p(x, y) \leftarrow x = y
\]

\[
p(x, y) \leftarrow x = f(z) \land y = g(z) \land notp(y, z)
\]

\[
notp(x, y) \leftarrow x \neq y \land \forall z (x \neq f(z))
\]

\[
notp(x, y) \leftarrow x \neq y \land x = f(z) \land y \neq g(z)
\]

\[
notp(x, y) \leftarrow x \neq y \land x = f(z) \land y = g(z) \land p(y, z)
\]
3.3. Correctness

We show that the negation operation $N(P)$ preserves the Fitting semantics of $P$. With this result we show that the Fitting semantics of the program $P'$ is the well-founded semantics of the original program $P$.

Lemma 3.16. For any ordinal $k$,
\begin{enumerate}[(a)]
\item $\phi_p \uparrow k \models p(\tilde{s}) \iff \phi_{N(P)} \uparrow k \models p(\tilde{s})$
\item $\phi_p \uparrow k \models \neg p(\tilde{s}) \iff \phi_{N(P)} \uparrow k \models \neg p(\tilde{s})$
\end{enumerate}
for each atom $p(\tilde{s}) \in HB_P$ and
\begin{enumerate}[(c)]
\item $\phi_p \uparrow k \models p(\tilde{s}) \iff \phi_{N(P)} \uparrow k \models \neg \text{not } p(\tilde{s})$
\item $\phi_p \uparrow k \models \neg p(\tilde{s}) \iff \phi_{N(P)} \uparrow k \models \text{not } p(\tilde{s})$
\end{enumerate}
for each atom $p(\tilde{s}) \in HB_P$ such that $p \in \text{Free}(P)$.

Proof. By induction.

The base cases are trivial. If $k$ is a limit ordinal then for each $p(\tilde{s}) \in \Phi_p \uparrow k$ there exists $j < k$ such that $p(\tilde{s}) \in \Phi_p \uparrow j$ and, thus, $p(\tilde{s}) \in \Phi_{N(P)} \uparrow k$ by the induction hypothesis. The other cases are similar.

Otherwise, $k = k' + 1$ is a successor ordinal. We give the proofs for cases (a) and (d) only, since (b) and (c) are similar. Assume that clauses in $P$ are written in the form
\[ p(\tilde{x}) \leftarrow \tilde{x} = \tilde{i}(\tilde{y}), B(\tilde{x}, \tilde{y}) \]
where $B(\tilde{x}, \tilde{y})$ is a conjunction of literals involving variables $\tilde{x}$ and $\tilde{y}$. If $p$ has a local-variable-free definition then, clearly, there is only one ground substitution $\alpha$ such that $\tilde{x} \alpha = \tilde{s}$ since then $\tilde{i}(\tilde{y}) \alpha = \tilde{s}$ and, hence, $\tilde{y} \alpha = \tilde{s}'$ for some fixed $\tilde{s}'$.

(a) $p(\tilde{s}) \in \Phi_p \uparrow k' + 1 \iff p(\tilde{s}) \in \Phi_{N(P)} \uparrow k' + 1$: We have
\[ p(\tilde{s}) \in \Phi_p \uparrow k' + 1 \iff \text{there exists a clause in } P \text{ of the form} \]
\[ p(\tilde{x}) \leftarrow \tilde{x} = \tilde{i}(\tilde{y}), B(\tilde{x}, \tilde{y}) \]
and ground substitution $\alpha$ such that $\tilde{x} \alpha = \tilde{s}$ and $\phi_p \uparrow k' \models B(\tilde{x}, \tilde{y}) \alpha$.

$\iff$ there exists a clause in $\Phi_{N(P)}$ of the form
\[ p(\tilde{x}) \leftarrow \tilde{x} = \tilde{i}(\tilde{y}), B'(\tilde{x}, \tilde{y}) \]
and ground substitution $\alpha$ such that $\tilde{x} \alpha = \tilde{s}$ and $\phi_{N(P)} \uparrow k' \models B'(\tilde{x}, \tilde{y}) \alpha$ by the induction hypothesis, noting that $B'(\tilde{x}, \tilde{y})$ differs from $B(\tilde{x}, \tilde{y})$ only in the replacement of negative literals $\neg q(\tilde{s})$ by literals $\text{not } q(\tilde{s})$.

$\iff p(\tilde{s}) \in \Phi_{N(P)} \uparrow k' + 1$.
We have
$$\neg p(\tilde{s}) \in \Phi_P \uparrow k \iff \text{not}p(\tilde{s}) \in \Phi_{N(P')} \uparrow k$$
We have
$$\iff \text{for each clause in } P \text{ (note they must be local variable free)}$$
of the form
$$p(\tilde{x}) \leftarrow \tilde{x} = \tilde{t}_i(\tilde{y}_i), B_i(\tilde{x}, \tilde{y}_i)$$
and ground substitution $\alpha$ such that $\tilde{x}\alpha = \tilde{s}$ either
$$\tilde{t}_i(\tilde{y}_i)\alpha \text{ or } \Phi_P \uparrow k' \models \neg B_i(\tilde{x}, \tilde{y}_i)\alpha.$$}
$$\iff \text{for each clause in } P \text{ of the form}$$
$$p(\tilde{x}) \leftarrow \tilde{x} = \tilde{t}_i(\tilde{y}_i), B_i(\tilde{x}, \tilde{y}_i)$$
and ground substitution $\alpha$ such that $\tilde{x}\alpha = \tilde{s}$ then
$$\Phi_P \uparrow k' \models \neg (\tilde{x} = \tilde{t}_i(\tilde{y}_i) \land B_i(\tilde{x}, \tilde{y}_i))\alpha.$$ Let $E_i(\tilde{x})$ represent
$$\exists \tilde{y}_i(\tilde{x} = \tilde{t}_i(\tilde{y}_i) \land B_i(\tilde{x}, \tilde{y}_i))$$
$$\iff \Phi_P \uparrow k' \models (\neg E_1(\tilde{x}) \land \cdots \land \neg E_m(\tilde{x}))\alpha \text{ for each ground substitution } \alpha \text{ such that } \tilde{x}\alpha = \tilde{s}$$
$$\iff \Phi_P \uparrow k' \models (\neg E_1(\tilde{x}) \land \cdots \land \neg E_m(\tilde{x})) \text{ (since each } E_i(\tilde{x}) \text{ has at most one solution for the } \tilde{y}_i \text{ variables when } \tilde{x} = \tilde{s}. \text{ But by the construction of } \text{Not}(p)$$
$$\forall \tilde{x} \models (\neg E_1(\tilde{x}) \land \cdots \land \neg E_m(\tilde{x})) \iff (\bar{E}_1(\tilde{x}) \lor \cdots \lor \bar{E}_m(\tilde{x}))$$
where $\bar{E}_j$ are the bodies for clauses of the form
$$\text{not}p(\tilde{x}) \leftarrow \bar{E}_j(\tilde{x}) \text{ in } \text{Not}(p).$$
$$\iff \Phi_P \uparrow k' \models (\bar{E}_1(\tilde{s}) \lor \cdots \lor \bar{E}_m(\tilde{s}))$$
$$\iff \exists j \text{ such that } \Phi_P \uparrow k' \models \bar{E}_j(\tilde{s})$$
$$\iff \exists j \text{ such that } \Phi_{N(P')} \uparrow k' \models \bar{E}_j(\tilde{s}) \text{ (by the induction hypothesis)}$$
$$\iff \text{there exists a clause in } N(P) \text{ of the form}$$
$$\text{not}p(\tilde{x}) \leftarrow \bar{E}_j(\tilde{x})$$
and substitution $\beta$ such that $\tilde{x}\beta = \tilde{s}$ and $\Phi_{N(P')} \uparrow k' \models \bar{E}_j(\tilde{s})\beta$
(where $\bar{E}_j(\tilde{x})$ and $\bar{E}_j(\tilde{s})$ differ only by substitution of negative literals $\neg q(\tilde{s})$ by literals $\text{not}q(\tilde{s})$)
$$\iff \text{not}p(\tilde{s}) \in \Phi_{N(P')} \uparrow k' + 1.$$ Theorem 3.17 (Correctness). $\text{lfp}(\Phi_{N(P')}) = \text{lfp}(\Phi_P)$ restricting to the predicates of $P$. Proof. Directly from Lemma 3.16. We wish to apply the transformation to programs $P^\alpha$ whose Fitting semantics captures the well-founded semantics of the original program. Recall that we name the resulting program $P' = N(P^\alpha)$. In this case we have the following important corollary.
Corollary 3.18. If $P$ has finite $P^{\omega}$ then $\text{lfp}(\Phi_{P})$ is the well-founded semantics of $P$ restricting to the predicates of $P$.

Proof. From Lemma 3.12 and Theorem 3.17. 

Until the end of this section we assume that $P^{\omega}$ is local-variable-free and, hence, $P'$ is a definite constraint program. For definite constraint programs (over the domain of finite trees) we have the following relationship between the $\Phi_{p}$ operator of Fitting and the $T_{p}$ operator of van Emden and Kowalski [32].

Proposition 3.19 (Fitting [11]). If $P$ is a definite program then

$$\Phi_{p} \uparrow k = T_{p} \uparrow k \cup \{ \neg A \mid A \notin T_{p} \downarrow k \}$$

for any ordinal $k$.

We know that for definite programs $P'$, SLD-resolution (for constraint logic programs involving equalities and disequalities over the Herbrand universe) is sound and complete with respect to $\text{comp}(P')$ for success and finite failure due to the following results of Jaffar and Lassez [13].

Theorem 3.20 (Jaffar and Lassez [13]).

$$\text{comp}(P') \models \neg G \iff G \in T_{p} \downarrow \omega \iff G \in \text{SS}(P'),$$

$$\text{comp}(P') \models \neg G \iff G \notin T_{p} \downarrow \omega \iff G \in \text{FF}(P').$$

Unfortunately, this does not correspond, in general, to the Fitting semantics of the program $P'$ because $\text{lfp}(\Phi_{P'}) \neq \Phi_{p} \uparrow \omega$. The canonical programs [14] are those programs where the greatest fixpoint of $T_{p}$ is $T_{p} \downarrow \omega$. Let $A \models H B$ be true if every Herbrand model of $A$ models $B$; then we have the following result from Jaffar and Lassez.

Theorem 3.21 (Jaffar and Lassez [13]).

$$\text{comp}(P') \models_{H} G \iff G \in T_{p} \uparrow \omega \iff G \in \text{SS}(P'),$$

If $P'$ is canonical then

$$\text{comp}(P') \models_{H} \neg G \iff G \notin T_{p} \downarrow \omega \iff G \in \text{FF}(P').$$

Hence, the canonical programs are exactly those programs for which SLD resolution is complete for finite failure with respect to the Fitting semantics, or equivalently the Herbrand consequences of $\text{comp}(P')$. All practical programs are canonical, and it seems unlikely that $P'$ will not be canonical. We can, however, construct examples where $P'$ is not canonical.
Example 3.22. The following program $P(=P^\omega)$ is not local-variable-free because of the first clause, but the first clause is, in fact, not required for the transformation:

$$
p(a) \leftarrow \neg q(x)
q(x) \leftarrow \neg r(x)
\begin{align*}
    r(f(x)) & \leftarrow q(x)
\end{align*}
$$

$P'$ is

$$
p(a) \leftarrow \neg q(x)
q(x) \leftarrow \neg r(x)
\begin{align*}
r(f(x)) & \leftarrow \neg q(x)
\neg q(x) & \leftarrow r(x)
\neg r(x) & \leftarrow \forall z, x \neq f(z)
\neg r(x) & \leftarrow x = f(z) \land q(z)
\end{align*}
$$

There are no Herbrand models of $P'$ that make $p(a)$ true but $p(a)$ does not finitely fail; thus, $P'$ is not canonical.

We now show that for the class of programs which we are principally interested in, our transformation always produces canonical programs.

Lemma 3.23. If $P^\omega$ is finite and local-variable-free then $P'$ is canonical.

Proof. From Theorem 3.17, it suffices to show that $P^\omega$ is canonical since $\Phi_{P'}^{\uparrow}\omega = \Phi_P^{\uparrow}\omega$ and $\text{lp}(\Phi_{P'}) = \text{lp}(\Phi_P)$. Suppose $P^\omega$ is not canonical; then there exists a literal $q$, where $q \in \Phi_{P'}^{\uparrow}(\omega + 1) - \Phi_{P'}^{\uparrow}\omega$. It is easy to show that if $q$ is a positive literal this cannot occur; so, assume that $q = \neg p$. Since $P^\omega$ is finite and local-variable-free, there are only finitely many clauses in $G_P$, with head $p$. Since $p$ is false in $\Phi_{P'}^{\uparrow}(\omega + 1)$, each body of these clauses is false in $\Phi_{P'}^{\uparrow}\omega$, but, since there are only finitely many atoms in the body, this implies that each body was false at some finite stage $\Phi_{P'}^{\uparrow}k$ and, hence, $\neg p \in \Phi_{P'}^{\uparrow}(k + 1) \subseteq \Phi_{P'}^{\uparrow}\omega$. Contradiction. \[\square\]

Theorem 3.24. For $P$ with finite and local-variable-free $P^\omega$,

(a) SLD-resolution on $P'$ is sound and complete for success.

(b) SLD-resolution on $P'$ is sound and complete for finite failure, with respect to the well-founded model of $P$.

Proof. $P'$ contains only positive atoms and equality and disequality constraints, and is canonical by Lemma 3.23. Hence, by Corollary 3.18 and Theorem 3.21 the result follows. \[\square\]
Corollary 3.25. For sufficiently stratified $P$ with finite and local-variable-free $P^\omega$,
(a) SLD-resolution on $P'$ is sound and complete for success.
(b) SLD-resolution on $P'$ is sound and complete for finite failure, with respect to the unique stable model of $P$.

Programs $P'$ which result from nonlocal-variable-free $P^\omega$ may include negative literals in the body. In this case, we suggest the use of constructive negation as an operational semantics. Chan's constructive negation scheme [5] is only known to be sound with respect to the three-valued consequences of the completion, but there are other schemes [30] that are known to be both sound and complete. As in the above discussion (and see [16]), the (atomic) three-valued consequences of the completion of a program $P'$ only coincide with the Fitting semantics when $\text{lfp}(\Phi_P) = \Phi_P.\uparrow^\omega$, in other words, for canonical normal programs. Unlike in the above discussion, we cannot dismiss the possibility of noncanonical programs, so we are left with the following result, where we use SLD-CNF to stand for a sound and complete implementation of constructive negation.

Theorem 3.26. For $P$ with finite and canonical $P^\omega$,
(a) SLD-CNF resolution on $P'$ is sound and complete for success.
(b) SLD-CNF resolution on $P'$ is sound and complete for finite failure, with respect to the well-founded model of $P$.

4. Restrictions

In this section we discuss for which original programs $P$ we can ensure the conditions of local-variable freeness and finiteness of $P^\omega$.

4.1. Local-variable freeness

We can characterize those programs $P$ whose positive expanded version $P^\omega$ will contain only local-variable-free clauses as follows.

Definition. Let $P^+$ be the program $P$ with negative literals removed. A program $P$ is positive-grounded if for each clause $C$ in $P$

$$A \leftarrow B_1, \ldots, B_n, \neg B_{n+1}, \ldots, \neg B_m,$$

and each successful derivation $\leftarrow B_1, \ldots, B_n$ from $P^+$ with answer substitution $\theta$ then all variables in $C\theta$ appear in $A\theta$.

Positive groundness is not a syntactic condition. It is clear that if a program is positive-grounded it is admissible, i.e. every variable appearing in a clause appears in
a positive literal or in the head (see [17] for more details). Sufficient conditions for positive groundness include allowedness [17], where every variable in a clause must appear in a positive body literal, and mode correctness [29].

**Theorem 4.1.** $P$ is positive-grounded iff all clauses of $P^+$ are local-variable-free.

**Proof.** The proof proceeds by induction. We show $P$ is positive-grounded for each BF-derivation [34] of length $\leq n$ in $P^+$ (which are exactly those captured in $\mathcal{F}_p^+(\emptyset)$) iff all clauses in $\mathcal{F}_p^{n+1}(\emptyset)$ are local-variable-free. Concurrently, we show $A \in \mathcal{F}_p^+(\emptyset)$ iff $A \leftarrow \neg B \in \mathcal{F}_p^+(\emptyset)$.

Base Case ($n=0$): Trivial.

**Induction step:** Take a clause $C$

$$A \leftarrow B_1, \ldots, B_n, \neg B_{n+1}, \ldots, \neg B_m$$

in $P$, the corresponding clause in $P^+$ is

$$A \leftarrow B_1, \ldots, B_n.$$

Let $B'_i \in \mathcal{F}_p^+(\emptyset)$ and $B_i \leftarrow \neg B_i$ be the corresponding elements of $\mathcal{F}_p^+(\emptyset)$ and let $B_i \theta = B_i \emptyset$ for $1 \leq i \leq n$. Now, by the induction hypothesis, each $B_i \leftarrow \neg B_i$ is local-variable-free; hence, all variables in $\neg B_i \emptyset$ appear in $B_i \emptyset$. Then $A \emptyset \in \mathcal{F}_p^{n+1}(\emptyset)$ and, similarly, the clause $C' \in \mathcal{F}_c(B'_1 \leftarrow \neg B'_1, \ldots, B'_n \leftarrow \neg B'_n)$,

$$A \emptyset \leftarrow \neg B'_1 \emptyset, \ldots, \neg B'_n \emptyset, \neg B_{n+1} \emptyset, \ldots, \neg B_m \emptyset$$

is in $\mathcal{F}_p^{n+1}(\emptyset)$. Let $\emptyset$ be an answer substitution for a BF-derivation of $B_1, \ldots, B_n$ of length $\leq n$. $\emptyset$ is positive grounding for $C$ iff

- all variables in $C \emptyset$ are in $A \emptyset$ iff
- all variables in $B_i \emptyset$ and $\neg B_i \emptyset$ are in $A \emptyset$ iff
- all variables in $\neg B'_i \emptyset$ and $\neg B_i \emptyset$ are in $A \emptyset$ iff
- $C'$ is local-variable-free. 

**Example 4.2.** The following program $P$ (from [12]) is not locally stratified but is sufficiently stratified and positive-grounded:

\[
p(a, b)\]
\[
q(x) \leftarrow p(x, y) \land \neg q(y)\]

$P^+$ becomes

\[
p(a, b)\]
\[
q(a) \leftarrow \neg q(b)\]
Then $P'$ becomes

$$
\begin{align*}
p(a, b) \\
q(x) &\leftarrow x = a \land \neg q(b) \\
\neg q(x) &\leftarrow x \neq a \\
\neg q(x) &\leftarrow q(b)
\end{align*}
$$

The following program demonstrates that positive-grounded programs are a larger class than flounder-free programs.

**Example 4.3.** Let $P$ be

$$
\begin{align*}
even(0) &\leftarrow \\
\even(s(x)) &\leftarrow \neg \even(x)
\end{align*}
$$

$P$ is positive-grounded but not flounder-free for nonground goal $\even(y)$. $P'$ is the program

$$
\begin{align*}
even(0) &\leftarrow \\
\even(s(x)) &\leftarrow \neg \even(x) \\
\neg \even(s(x)) &\leftarrow \even(x)
\end{align*}
$$

For this program the goal $\even(y)$ will correctly find answers.

This means that by applying our transformation to a normal program we can achieve a constraint program that can successfully execute more goals than the original program using SLDNF. In fact, we can show that positive-grounded programs are exactly those that are flounder-free for all ground queries.

**Proposition 4.4.** $P$ is positive-grounded iff every SLDNF derivation for a ground query is flounder-free.

**Proof.** We show that $P^\omega$ is local-variable-free iff every SLDNF derivation for a ground query is flounder-free.

$\Rightarrow$: Suppose to the contrary that in an SLDNF derivation for the ground goal $p(\vec{s})$ using $P$, we arrive at a goal \( \neg B_1, \ldots, \neg B_n \), where no literal can be selected. Examining the derivation for $p(\vec{s})$ it is easy to see that there must exist a clause

$$
p(\vec{t}) \leftarrow \neg B'_1, \ldots, \neg B'_n, \neg B'_{n+1}, \ldots, \neg B'_m
$$

in $\mathcal{F}_P^{\omega}(\emptyset)$ for some finite $k$ and substitution $\theta$, where $\vec{s} = \vec{t}\theta$ and $B_i = B'_i\theta$, $1 \leq i \leq n$. Now since this clause appears in $P^\omega$ it must be local-variable-free. Hence, each $B_i$ must be ground since $\theta$ grounds all the variable in the head. Contradiction.
\( \Leftarrow: \) Suppose to the contrary that \( P'^\omega \) contains a nonlocal-variable-free clause \( C \) given by
\[
p(\tilde{t}) \leftarrow \neg B_1, \ldots, \neg B_n.
\]
Then there exists a finite \( k \) such that \( C \in \mathcal{T}_k(\theta) \). Now each derivation for ground goal \( p(\tilde{t}) \theta \), where \( \theta \) ground the variables in \( \tilde{t} \) is flounder-free, so there exists a derivation that selects only negative literals when there is no positive literal which reaches a goal \( \neg B_1 \theta, \ldots, \neg B_n \theta \). Since this derivation does not flounder, each of the literals \( \neg B_1 \theta, \ldots, \neg B_n \theta \) must be ground. Contradiction. \( \square \)

The positive groundness restriction can be loosened in a number of ways. There are often predicates whose negated definitions are not required by the transformation and, hence, we do not require the clauses defining them in \( P'^\omega \) to be local-variable-free. Similarly, predicates for which \( P'^\omega \) contains clauses containing local variables may still be eliminated if the part of \( P'^\omega \) they depend on is finite and stratified. In this case we can replace negated goals \( \neg p(\tilde{s}) \) by the negated (finite) set of answers to the query \( p(\tilde{s}) \) given by Chan's original constructive negation scheme \([4]\). See Example 4.5. Note that if \( P \) is stratified then we do not lose stratification in the positive transformation, e.g. \( p \leftarrow q, q \leftarrow \neg r \) becomes \( p \leftarrow \neg r \).

**Example 4.5.** The second clause of the following program contains a local variable but \( p(x) \) has a finite set of answers \( p(x) \leftrightarrow x \neq a \).

\[
\begin{align*}
r(x) & \leftarrow \neg p(x) \\
p(x) & \leftarrow \neg q(x, y) \\
q(a, a) & \\
q(a, f(x)) &
\end{align*}
\]

Hence, it can be transformed to

\[
\begin{align*}
r(x) & \leftarrow \neg p(x) \\
p(x) & \leftarrow \neg q(x, y) \\
\neg p(x) & \leftarrow x = a \\
\neg q(x, y) & \leftarrow x \neq a \\
q(a, a) & \\
q(a, f(x)) &
\end{align*}
\]

4.2. Finite representation

We need to keep our intermediate program \( P'^\omega \) finite, in order to be able to apply the negation operation, and this implies a finite \( \mathcal{T}_P \uparrow \omega \). It is quite obvious that \( P'^\omega \) may
be infinite when we have positive recursion involving function symbols. This includes most practical programs. We show that in these cases, we may be able to omit the bottom-up expansion of some positive literals to ensure a finite fixpoint of $\mathcal{F}_P$.

**Example 4.6.** The following program has an infinite $P^\omega$.

$$
q(x), \text{even}(x) \\
\text{even}(0) \\
\text{even}(s^2(x)) \leftarrow \text{even}(x)
$$

We can avoid the infinite $P^\omega$ by replacing the literal $\text{even}(x)$ in the body by $\neg \neg \text{even}(x)$ as follows:

$$
q(x) \leftarrow \text{even}(x) \\
\text{even}(0) \\
\text{even}(s^2(x)) \leftarrow \neg \neg \text{even}'(x) \\
\text{even}'(x) \leftarrow \neg \text{even}(x)
$$

In this case $P^\omega$ is the following program:

$$
q(0) \\
q(s^2(x)) \leftarrow \neg \neg \text{even}'(x) \\
\text{even}(0) \\
\text{even}(s^2(x)) \leftarrow \neg \text{even}'(x) \\
\text{even}'(x) \leftarrow \neg \text{even}(x)
$$

In effect, we have not used the literal $\text{even}(x)$ in the positive transformation. We can achieve the same effect by not expanding the literal in the bottom-up expansion. This results in an equivalent $P^\omega$ of the form

$$
q(0) \\
q(s^2(x)) \leftarrow \text{even}(x) \\
\text{even}(0) \\
\text{even}(s^2(x)) \leftarrow \text{even}(x)
$$

**Example 4.7.** For a more practical example consider the following program for the Yale shooting problem. It results in an infinite $P^\omega$ because of infinite positive recursion in the predicates $\text{dead}$ and $\text{loaded}$.
If we omit their expansion in the positive transformation, $P^\omega$ is finite and the final program $P'$ is:

\[
\begin{align*}
\text{dead}(T) & \leftarrow \text{dead}(T0), \text{next}(T0, T) \\
\text{dead}(T) & \leftarrow \text{shoot}(T0), \text{next}(T0, T) \\
\text{loaded}(0) & \\
\text{loaded}(T) & \leftarrow \text{loaded}(T0), \neg \text{shoot}(T0), \text{next}(T0, T) \\
\text{shoot}(T) & \leftarrow \text{loaded}(T), \neg \text{dead}(T), \neg \text{asleep}(T) \\
\text{asleep}(0) & \\
\text{next}(T, s(T)) &
\end{align*}
\]

We try to prevent infinite $P^\omega$ by replacing positive literals that may cause infinite positive recursion by equivalent negative literals, transforming the original program $P$ to another program $P_2$ before the positive transformation is applied. If $p(s)$ is an atom in the body of some clause that may cause infinite positive recursion then we replace it by the literal $\neg p(s)$ and add the rule $p'(\bar{x}) \leftarrow \neg p(\bar{x})$ to the program $P_2$. We
call the atoms \( p(\bar{s}) \) that are so modified the \textit{changed} atoms and \( p'(\bar{s}) \) their \textit{replacing} atoms. This \textit{double negation} transformation ensures that none of the changed atoms \( p(\bar{s}) \) is expanded in the positive transformation of \( P_2 \). After applying the negative expansion on the resulting \( P'_2 \), each \( \neg p'(\bar{s}) \) is effectively replaced by \( p(\bar{s}) \). We must be careful to ensure that the program \( P_2 \) resulting from the double negation transformation is positive-grounded, because even if \( P \) is positive-grounded this property may be destroyed by the transformation.

The above transformation does not, in general, preserve the well-founded semantics. The following program

\[
p \leftarrow p
\]

is transformed to

\[
p \leftarrow \neg p'
\]
\[
p' \leftarrow \neg p
\]

where the well-founded model is not preserved. This is because we have replaced positive dependencies by (double) negative dependencies. In order for the transformation to preserve the well-founded models, we require that for each ground instance of a changed atom, the well-founded model of \( P \) can be constructed without using an unfounded set containing a ground instance of a changed atom. In this case, a modified construction for \( P_2 \) will give the same answer. We claim that the well-founded model is almost always conserved in most cases since for most of the time we are trying to change recursion like

\[
p(f(x)) \leftarrow p(x)
\]

to

\[
p(f(x)) \rightarrow \neg p'(x)
\]
\[
p'(x) \leftarrow \neg p(x)
\]

where, clearly, no positive loops (at the ground level) have become negative loops. We present a sufficient condition for ensuring \( P \) and \( P_2 \) have the same well-founded models.

\textbf{Definition.} The \textit{predicate dependency graph} \( D_P \) of a program \( P \) is a directed graph whose nodes are predicate symbols and whose edges represent the relation \textit{refer to} between predicate symbols of \( P \). An edge \((p, q)\) exists and is \textit{positive (negative)} iff there is a clause \( C \) in \( P \) in which \( p \) is the predicate symbol in the head of \( C \) and \( q \) is the predicate symbol of a positive (negative) literal in the body of \( C \).

A strongly connected component (SCC) \( S_1 \) of \( D_P \) is \textit{lower} than another, \( S_2 \), if there is a path from \( S_2 \) to \( S_1 \).

The \textit{positive dependency graph} \( D^+_P \) is the subgraph of the predicate dependency graph consisting of only the positive edges.
Let \( S_q(\mathcal{S}_q^+) \) be the set of predicates in the same SCC of \( D_p(\mathcal{D}_p^+) \) as the predicate \( q \).

Let \( P \) be a program and \( M \) a three-valued interpretation of \( P \). The ground program \( P/M \) is defined as the set of clauses in \( \mathcal{G}_P \) whose body literals are not false in \( M \).

Let \( P(S) \) denote the rules in \( P \) whose heads are in the set of predicates \( S \).

For simplicity, we restrict ourselves to the case where there is a single changed atom \( p(\delta) \) whose replacing atom is \( p(\delta) \). We can apply the following proposition multiple times to handle multiple changed atoms.

**Proposition 4.8.** Let \( M \) be the well-founded model of the program \( P(T) \), where \( T \) is the set of predicates in SCC of \( D_p \) strictly lower than the \( \mathcal{S}_q^+ \). Let \( C = P_2(\mathcal{S}_q^+ \cup \{p'\}) \). If \( C/M \) is locally stratified then the well-founded model of \( P \) is the same as the well-founded model of \( P_2 \) (restricting to the predicates of \( P \)).

**Proof (Sketch).** Note that \( M \) defines exactly the well-founded model of \( P \) restricted to the predicates in \( T \). Also the clauses in \( P(T) \) appear in \( P_2 \) unchanged. Clearly, if the well-founded model of \( P \) and \( P_2 \) differ, they must differ on the predicates of \( \mathcal{S}_q^+ \), since these are the only predicates where we may have replaced positive loops with negative loops. But since \( C/M \) is locally stratified, the well-founded model construction never required a positive loop in its inferences. Hence, the well-founded model of \( P \) and \( P_2 \) agree on the predicates of \( P \). \( \square \)

For example, in the program of Example 4.7 we changed the recursive call to \( \text{dead} \). The predicates in the same SCC as \( \text{dead} \) are \( S_{\text{dead}} = \{\text{dead, loaded, shoot}\} \) but the only predicate in the same positive SCC is \( S_{\text{dead}}^+ = \{\text{dead}\} \). Clearly, any positive loops that are converted to negative loops must occur in the same positive SCC. Examine the rules for \( \text{dead} \) and \( \text{dead}' \) in \( P_2 \).

\[
\text{dead}'(T) \leftarrow \neg \text{dead}(T)
\]
\[
\text{dead}(T) \leftarrow \neg \text{dead}'(T0), \text{next}(T0, T)
\]
\[
\text{dead}(T) \leftarrow \text{shoot}(T0), \text{next}(T0, T)
\]

The lower SCC of \( D_p(\mathcal{P}(T)) \) include the predicate \( \text{next} \), and any instantiation of these clauses satisfying the well-founded model for \( \text{next} \) gives us a locally stratified program.

Checking the conditions for the above proposition is quite closely related to detecting positive loops in top-down execution of definite programs. Many of the analysis techniques for determining that a definite program is loop-free (e.g. \([23, 31]\)) can be modified to this purpose.
5. Discussion

5.1. Related work

Ross [26] defined global SLS-resolution as a top-down procedural semantics corresponding to the well-founded semantics. Global SLS-resolution is not effective, nor could it hope to be for arbitrary programs, and suffers the same floundering problems as SLDNF. The positive transformation from $P$ to $P^o$ can be related to global SLS-resolution. Ross restricts computation rules to be positivistic, i.e. selecting only positive literals until only negative literals remain. Our positive transformation can be seen as computing, in a bottom-up manner, the positivistic nonlooping derivations appearing in Ross's SLP trees. Even when restricted to positive grounded programs with finite $P^o$, global SLS-resolution is not effective (unless some kind of loop checking is employed and it is certainly not clear how this may be done efficiently) and may flounder. Hence, it cannot answer all the programs and queries that our approach can.

The negative expansion technique discussed in this paper was originated by Sato and Tamaki [27]. It has appeared in many guises since then. Chan and Wallace [6] proposed a treatment of negation during partial evaluation time by expanding negated subgoals or eliminating them in order to improve run-time efficiency. They applied the negation technique of Sato and Tamaki [27] to eliminate negated atoms whenever possible. The remaining negated atoms are treated by constructive negation [5] during run-time. The constructive negation procedure itself has a strong relationship to the negation technique of Sato and Tamaki. Since they use the Sato and Tamaki transformation, Chan and Wallace [6] are restricted to local-variable-free clauses of the original program.

Barbuti et al. [2] define a similar transformation which is applicable only for definite programs with a domain closure assumption. It is again not complete for clauses with local variables. They propose an interesting operational scheme to treat queries containing local variables, but unfortunately to ensure completeness they must invoke an instantiating predicate which, in the worst case, will exhaustively ground the local variables. With this transformation, the treatment of disequality also runs into problems because negative equality goals may return an infinite number of answers even when a unique answer exists. They point out that the CLP approach to disequality, as in our scheme, removes this problem. Mancarella et al. [20] extended the transformation of Barbuti et al. to normal logic programs, again with the restriction that all clauses must be local-variable-free. Independently, they showed that this transformation preserves the three-valued consequences of the completion of the original program, a weaker result than our Lemma 3.16. This gives them a sound and complete operational semantics (with respect to three-valued consequences of the completion) for programs that do not include local variables.

Because both the above schemes apply the transformation to the original program, while we apply our transformation to $P^o$, they are unable to eliminate all negations
from positive grounded programs. When applied to $P_0^\alpha$ they yield similar programs to $P_1^\alpha$, though neither paper showed that Fitting’s semantics were preserved, which is essential for the soundness and completeness of our scheme with respect to the well-founded semantics of $P$.

We propose the use of a constructive negation technique [5, 30] to handle any remaining negated atoms. In fact, we can view the negative expansion step as compiling the steps of constructive negation into a definite program. Our compiled version has all or most negated atoms eliminated, making it more efficient than handling negated subgoals at run time.

5.2. Conclusion

We have presented a scheme which involves two transformations: the first evaluates partially normal programs to obtain their semantic kernels, the second replaces negative literals by positive literals where possible. Their combination yields the fundamental contribution of this work: a method of translating normal logic programs to constraint logic programs, that gives us an effective operational procedure for evaluating the well-founded model of the original program. For programs where not all negative literals can be removed, we propose the use of a constructive negation technique [5, 30] to handle any remaining negative literals. In fact, we can view the negative expansion step as compiling the steps of constructive negation into a definite program. Our compiled version eliminates the need for these steps at run-time, and allows much of the redundancy that arises from these transformation to be eliminated, thus making it more efficient than handling negated subgoals at run-time. The scheme can readily be extended to handle original programs $P$ which are normal constraint logic programs over the domain of finite trees, rather than just normal logic programs.

One interesting question which arises out of this work is the semantics given by the three-valued consequences of $\text{comp}(P_0^\alpha)$ or, equivalently, Kunen’s semantics [16] applied to $P_0^\alpha$. In fact, this is exactly the semantics we compute, but we have restricted ourselves to canonical programs $P_0^\alpha$, where Fitting’s and Kunen’s semantics coincide. If we remove the canonical restriction, it appears that we are computing the three-valued (non-Herbrand) well-founded consequences of $P$, which is the fixpoint completion semantics [8] for the case of nonground semantic kernels.

References

Transforming normal logic programs


