View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CWI's Institutional Repository

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Transforming acyclic programs

A. Bossi S. Etalle

Computer Science/Department of Software Technology

CS-R9369 1993

Transforming Acyclic Programs

Annalisa Bossi[1], Sandro Etalle[1,2].

[1] *Dipartimento di Matematica Pura ed Applicata, Università di Padova, Via Belzoni 7, 35131 Padova, Italy.*

[2] *CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands.*

email: bossi@zenone.math.unipd.it, etalle@cwi.nl

Abstract

An Unfold/Fold transformation system is a source-to-source rewriting methodology devised to improve the efficiency of a program. Any such transformation should preserve the main properties of the initial program: among them, termination. To this end, in the field of logic programming, the class of acyclic programs plays an important role, as it is closely related to the one of terminating programs. The two classes coincide when negation is not allowed in the bodies of the clauses.

In this paper it is proven that the Unfold/Fold transformation system defined by Tamaki and Sato preserves the acyclicity of the initial program. As corollaries, it follows that when the transformation is applied to an acyclic program, then finite failure set for definite programs is preserved; in the case of normal programs, all major declarative and operational semantics are preserved as well. These results cannot be extended to the class of left terminating programs without modifying the definition of the transformation.

AMS Subject Classification (1991): 68Q40, 68T15.
CR Subject Classification (1991): D.1.6, F.3.2, F.4.1, I.2.2, I.2.3.
Keywords and Phrases: Program's Transformation, Logic Programming, Termination, Terminating programs, Acyclic programs.
Note: this work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant n. 89.00026.69.

1. Introduction

1.1 Motivation

In this paper we focus on the unfold/fold transformation systems proposed by Tamaki and Sato [TS84].

As the large literature shows [TS84, KK88, Sek90, Sek91, Sek93, AD93], a lot of research has been devoted to proving the correctness of the system wrt the various semantics proposed for logic programs. However the question of the consequences of the transformation on the (universal) termination of the program has not yet been tackled.

Recall that a program is called *terminating* if all its SLDNF derivations starting in a ground goal are finite. In this paper we follow the approach to termination of Apt and Bezem [AB91]. They investigate the class of *acyclic* programs (introduced by Cavedon [Cav91]) and prove that it is closely related to the one of terminating programs. In fact we have that every acyclic

program is terminating [AB91] and that every definite, terminating program is acyclic [Bez93]; however, when negation is allowed in the bodies of the clauses, then there exist terminating programs which are not acyclic [AB91].

In this paper we prove that when the initial program is acyclic, then the resulting program is acyclic as well.

This has some obvious consequences on the preservation of termination and some semantic repercussions. In fact since acyclic programs are terminating, their Finite Failure Set coincides with the complement of their Success Set; it follows that in this case the transformation preserves the Finite Failure Set (for definite programs). A similar reasoning applies to all the major formalisms for programs with negation, namely Fitting's model, 2 and 3 valued ground logical consequence of the completion, and, in the non-floundering cases, the operational semantics based on the SLDNF-resolution: when the program is acyclic they all coincide and thus they are preserved by the transformation.

1.2 Structure of the Paper

Sections 2 and 3 contain the preliminaries on terminating and acyclic programs and on the Tamaki-Sato's unfold/fold transformation system. In section 4 we prove that the transformation preserves the acyclicity of the initial program; we also discuss the case in which the initial program is left terminating. In section 5 we give a brief summary of the semantic properties of acyclic programs and we show that they are preserved through the transformation.

1.3 Preliminaries

We assume that the reader is familiar with the basic concepts of logic programming; throughout the paper we use the standard terminology of [Llo87] and [Apt90]. We consider *normal programs*, that is finite collections of *normal rules*, $A \leftarrow L_1, \ldots, L_m.$ where A is an atom and $L_1, \ldots, L_m$ are literals. We say that a clause is *definite* if the body contains only positive literals (atoms); a definite program is then a program consisting only of definite clauses. Symbols with a $\sim$ on top denote tuples of objects, for instance $\tilde{x}$ denotes a tuple of variables $x_1, \ldots, x_n$, and $\tilde{x} = \tilde{y}$ stands for $x_1 = y_1 \wedge \ldots \wedge x_n = y_n$. We also adopt the usual logic programming notation that uses "," instead of $\wedge$, hence a conjunction of literals $L_1 \wedge \ldots \wedge L_n$ will be denoted by $L_1, \ldots, L_n$ or by $\widetilde{L}$. We denote by $Var(E)$ the set of all the variables in an expression E.

2. Termination

2.1 Terminating and Acyclic programs

The following notion is crucial.

Definition 2.1 A program is called *terminating* iff all its SLDNF-derivations starting from a ground goal are finite. □

Hence terminating programs are the ones whose SLDNF-trees starting in a ground goal are finite. We now present the approach to the issue of termination followed by Apt and Bezem [AB91].

Acyclic Programs Acyclic programs, form a natural subclass of the locally stratified ones, and have been studied by Apt and Bezem [AB91] and by Cavedon [Cav91]. To give their definition, first we need the following notion.

Definition 2.2 Let P be a program, a *level mapping* for P is a function $|\ |: B_P \to \mathbf{N}$ of ground literals to natural numbers, such that, for each $A \in B_P$ $|A| = |\neg A|$. □

Definition 2.3 Let $|\ |$ be a level mapping.

- A clause is *acyclic wrt* $|\ |$ iff for every ground instance $A \leftarrow L_1, \ldots, L_k$ of it, and for each i, $|A| > |L_i|$;
- A program P is *acyclic wrt* $|\ |$ iff all its clauses are. P is called *acyclic* if it is acyclic wrt some level mapping. □

Following Bezem [Bez93] level mappings can be generalized and hence applied to nonground atoms.

Definition 2.4 Let $|\ |$ be a level mapping.

- A literal L is called *bounded* wrt $|\ |$ if $|\ |$ is bounded on the set $[L]$ of ground instances of L.
- A goal is called *bounded* wrt $|\ |$ iff all its literals are. □

Example 2.5 [AP93] Consider the program *member*.

$$P = \{\ member(X, [Y|Xs]) \leftarrow member(X, Xs).$$
$$member(X, [X|Xs]).\ \}$$

We adopt the standard list notation and define the function $|\ |_l$, called *listsize* which assigns natural numbers to ground terms as follows:
$|t|_l = 1$ if t is not of the form $[x_1|x_s]$ (this takes also care of the case $t = [\,]$).
$|[x_1|x_s]|_l = 1 + |x_s|_l$.
We can now define the level mapping $|\ |$ for the *member* program: $|member(t, s)| = |s|_l$. It is easy to see that program *member* is acyclic wrt $|\ |$ and that if l is a list (by this we mean $l = [x_1, \ldots, x_n]$, where the x_i's need not be ground), then $member(t, l)$ is a bounded atom. □

We can now relate acyclic and terminating programs.

Theorem 2.6 [AB91] If P is acyclic and G is a bounded goal then all SLDNF derivations of $P \cup \{G\}$ are finite. □

In [AB91] is stated that the converse of Theorem 2.6 holds in the case that no SLDNF-derivation starting in a ground goal contains a goal with a nonground negative literal in it, and that since that condition is quite constraining, that the result itself is too weak to be formalized. However it is significant at least for the case that we restrict our attention to definite programs; in fact in [Bez93] we have the following.

Theorem 2.7 [**Bez93**] Let P be a definite program, then P is terminating iff P is acyclic. □

From the procedural point of view, acyclic programs enjoy the following important property: the two most prominent approaches, namely the SLDNF resolution, see Lloyd [Llo87], and the SLS resolution from Przymusinski [Prz89], coincide when applied to acyclic programs. For the semantic properties of acyclic programs we refer to section 5.

3. Unfold/Fold Transformations

We now give the definition of unfold/fold transformation sequence that was first given by Tamaki and Sato [TS84] for definite programs and then used by Seki [Sek90, Sek93] for normal programs. Here we present it as it is in [Sek93]. All definitions are given modulo reordering of the bodies of the clauses, and standardization apart is always assumed.

Definition 3.1 (initial program) We call a normal program P_0 an *initial program* if the following two conditions are satisfied:

(I1) P_0 is divided into two disjoint sets $P_0 = P_{new} \cup P_{old}$;

(I2) All the predicates which are defined in P_{new} occur neither in P_{old} nor in the bodies of the clauses in P_{new}. □

The predicates defined in P_{new} are called *new* predicates, while those defined in P_{old} are the *old* predicates. Clauses in P_{new} will also be referred to as *defining* clauses.

Example 3.2 [Sek93] Let P_0 be the following program

$$
\begin{array}{lll}
P_0 = DB \cup \{ & c_1: \ path(X,[X]) & \leftarrow node(X). \\
& c_2: \ path(X,[X|Xs]) & \leftarrow arc(X,Y), path(Y,Xs). \\
& c_3: \ goodlist([]). & \\
& c_4: \ goodlist([X|Xs]) & \leftarrow \neg bad(X), goodlist(Xs). \\
& c_5: \ goodpath(X,Xs) & \leftarrow path(X,Xs), goodlist(Xs). \ \ \}
\end{array}
$$

where predicates *node, arc* and *bad* are defined in DB by a set of unit clauses. Predicate $goodpath(X, Xs)$ can be employed for finding a path Xs starting from the node X which doesn't contain "bad" nodes. Let $P_{old} = \{c_1, \ldots, c_4\} \cup DB$ and $P_{new} = \{c_5\}$, thus *goodpath* is the only new predicate. □

Unfolding an atom in the body of a clause consists in applying a resolution step to the considered atom in all possible ways. This operation is basic to all the transformation systems.

Definition 3.3 (unfold) Let $cl : A \leftarrow H, \tilde{K}.$ be a clause of a normal program P, where H is an atom. Let $\{H_1 \leftarrow \tilde{B}_1, \ldots, H_n \leftarrow \tilde{B}_n\}$ be the set of clauses of P whose heads unify with H, by mgu's $\{\theta_1, \ldots, \theta_n\}$.

- *Unfolding an atom H in cl* consists of substituting cl with $\{cl'_1, \ldots, cl'_n\}$, where, for each i,
  $cl'_i = (A \leftarrow \tilde{B}_i, \tilde{K})\theta_i$.

$unfold\ (P, cl, H) \stackrel{\text{def}}{=} P \backslash \{cl\} \cup \{cl'_1, \ldots, cl'_n\}$. □

Example 3.2 (part 2) By unfolding the atom $path(X, Xs)$ in the body of c_5, we obtain

c_6 : $goodpath(X, [X]) \leftarrow node(X), goodlist([X])$.

c_7 : $goodpath(X, [X|Xs]) \leftarrow arc(X, Y), path(Y, Xs), goodlist([X|Xs])$.

Both clauses can be further further unfolded, and the resulting clauses are

c_8 : $goodpath(X, [X]) \leftarrow node(X), \neg bad(X)$.

c_9 : $goodpath(X, [X|Xs]) \leftarrow arc(X, Y), path(Y, Xs), \neg bad(X), goodlist(Xs)$.

Let $P_1 = \{c_1, \ldots, c_4, c_8, c_9\} \cup DB$. □

Folding is the inverse of unfolding when one single unfolding is possible. It consists in substituting a literal L for an equivalent conjunction of literals $\widetilde{K}$ in the body of a clause c. This operation is used in all the transformation systems in order to pack back unfolded clauses and to detect implicit recursive definitions. In the literature we find different definitions for this operation. This is due to the fact that it is not generally safe even for declarative semantics and its application must be restricted by some conditions which depend on the semantics we choose. Such conditions can be either a constraint on how to sequentialize the operations while transforming the program [TS84, KK88], or they can be expressed only in terms of properties of the program, independently from its transformation history[BC93, Mah87].

The transformation sequence and the folding operation are defined in terms of each other.

Definition 3.4 (transformation sequence) A *transformation sequence* is a sequence of programs $P_0, \ldots, P_n$, $n \geq 0$, such that each program P_{i+1}, $0 \leq i < n$, is obtained from P_i by unfolding or folding a clause of P_i. □

Definition 3.5 (folding) Let $P_0, \ldots, P_i$, $i \geq 0$, be a transformation sequence, $c : A \leftarrow \tilde{K}', \tilde{J}$. a clause in P_i and $d : D \leftarrow \tilde{K}$. a clause in P_{new}. Let X be the set of all the variables occurring in the clause d. If there exists a substitution τ whose domain is the set X, such that the following conditions hold:

(F1) $\tilde{K}\tau = \tilde{K}'$;

(F2) τ renames with fresh variables the variables in $\tilde{K}$ not in D;

(F3) d is the only clause in P_{new} whose head is unifiable with $D\tau$;

(F4) one of the following two conditions holds

1. the predicate in A is an old predicate;
2. c is the result of applying unfolding at least once to a clause in P_{new};

then *folding* $D\tau$ *in* c *in* P_i consists of substituting c' for c in P_i, where

$head(c') \stackrel{\text{def}}{=} A$

$body(c') \stackrel{\text{def}}{=} D\tau, \tilde{J}$.

$\text{fold}(P_i, D\tau, c) \stackrel{\text{def}}{=} (P_i \backslash \{c\}) \cup \{c'\}$. □

Example 3.2 (part 3) We can now fold the body of c_9, using c_5 as folding clause, the resulting program is $P_2 = DB \cup \{c_1, \ldots, c_4, c_{10}\}$, where c_{10} is the following clause:

$c_{10}: \ goodpath(X, [X|Xs]) \leftarrow arc(X, Y), \neg bad(X), goodpath(Y, Xs).$

Notice that because this operation the definition of *goodpath* is now recursive. □

The transformation enjoys the following important properties.

Theorem 3.6 Let $P_0, \ldots, P_n$ be a transformation sequence.

- If P_0 is a definite program then
  - [TS84] The least Herbrand models of the initial and final programs coincide.
  - [KK88] The computed answers substitution semantics of the initial and final programs coincide.
- If P_0 is a normal program, then
  - [Sek90] The Stable models of the initial and final programs coincide.
  - [Sek93] The Well-Founded models of the initial and final programs coincide.
  - [Sek89] Under a further mild assumption on the initial program; if the initial program is stratified then the final program is stratified and their Perfect models coincide.
  - [AD93] The *semantic kernel* of the initial and final program coincide; this implies also that the Stable model semantics, the preferred extension semantics, the stationary semantics and the stable theory semantics of the initial and the final programs coincide. □

However, the transformation does not preserve the Finite Failure set of the initial (definite) program. More precisely we have that the Finite Failure set of the final program is contained in the one of the initial program, but, in general, not vice-versa.

Modified Folding In order to make the paper more self-contained, we have to mention that the problem of the correctness of the operation wrt the Finite Failure Set was pointed out by Seki, who modified the applicability conditions of the folding operation as follows.

Definition 3.7 (modified folding) [Sek91] The *modified* folding operation is defined exactly as in Definition 3.5, with the exception of condition **(F4)**, which is replaced by the following

(F4') one of the following two conditions holds

1. the predicate in A is an old predicate;
2. all the atoms in $\tilde{K}'$ are the result of some previous unfold operation. □

This Definition first appeared in [Sek89]. It is easy to see that when **F4'** holds, then **F4** holds as well, hence that the *modified folding* operation enjoys all the properties that were proven for the folding operation. Seki proved that modified folding preserves the Finite Failure set of a definite program [Sek89, Sek91]; later on Sato, on a work that extends this definition to full first order programs [Sat90], proved the correctness of the system wrt Kunen's semantics.

4. Transforming Acyclic Programs

We now show that if the initial program of a transformation sequence is acyclic then the resulting program is acyclic as well.

Notation. Let $P_0, \ldots, P_n$ be the transformation sequence we are considering. Since P_0 is acyclic, then it is acyclic wrt some level mapping, say $||\ ||$, moreover, there in no loss of generality in assuming that $||\ ||$ does not take value zero on any atom. Let *nf* be the number of foldings that are going to be performed in the sequence (which we assume greater than zero), and let *maxargs* be the maximum number of literals that a body of a clause of P_0 contains, augmented by one. We also suppose that *maxargs*> 1, as it is not possible to perform any unfold or fold operations on a program consisting solely of unit clauses.

We now define a new level mapping $|\ |$ for P_0.

Definition 4.1 The level mapping $|\ |$ is defined as follows. Let A be a ground atom.

- If A is an *old* atom then we let $|A| = nf \cdot maxargs^{||A||}$.
- If A is an *new* atom then we distinguish two subcases:
  - (a) If A unifies with the head of only one clause of P_{new}, $N \leftarrow B_1, \ldots, B_n$, suppose that $A = N\theta$, since $B_1, \ldots, B_n$ are *old* atoms, we have that $|\ |$ is already defined on their ground instances, so we set $|A| = |N\theta| = sup\{\sum_{i=1}^{n} |B_i\theta\gamma| \ \mid \ Dom(\gamma) = Var(B_1\theta, \ldots, B_n\theta)\} + 1$.
  - (b) (This case is of no relevance for the proof, as, because of condition **(F3)**, we are interested in computing the level mapping of atoms that unify with the head of only one clause of P_{new}; but we do have to extend $|\ |$ in a consistent way). If A unifies with the head of a (non-unit) set of clauses $\{N_1 \leftarrow B_{1,1}, \ldots, B_{1,n(1)} \quad \ldots \quad N_j \leftarrow B_{j,1}, \ldots, B_{j,n(j)}\} \subseteq P_{new}$, suppose that $A = N_i\theta_i$, we define $|A| = sup\{\sum_{j=1}^{n(i)} |B_{i,j}\theta_i\gamma| \ \mid \ i \in [1, \ldots, j],\ Dom(\gamma) = Var(B_{i,1}\theta_i, \ldots, B_{i,n(i)}\theta_i)\}+1$. □

$|\ |$ is obviously a level mapping, as it is defined and finite on each ground atom.

In order to prove that each of the program in the transformation sequence is acyclic wrt $|\ |$ we need the following simple but technical lemma.

Lemma 4.2 For nonzero integers $nf, n, n_1, \ldots, n_k$, if $1 < k <$*maxargs* then

- if $n > sup\{n_1, \ldots, n_k\}$, then $nf \cdot maxargs^n > nf + \sum_{j=1}^{k} nf \cdot maxargs^{n_j}$

Proof.

$nf + \sum_{j=1}^{k} nf \cdot maxargs^{n_j} \ \leq \ nf + nf \cdot k \cdot maxargs^{sup\{n_j\}} \leq$

Since $k <$*maxargs*

$\leq nf + nf \cdot (maxargs - 1) \cdot maxargs^{sup\{n_j\}} \ = \ nf + nf \cdot maxargs^{sup\{n_j\}+1} - nf \cdot maxargs^{sup\{n_j\}} \leq$

Since *maxargs*> 0 and $n > sup\{n_j\}$,

$\leq nf \cdot maxargs^n + nf - nf \cdot maxargs^{sup\{n_j\}} \ = \ nf \cdot maxargs^n + nf \cdot (1 - maxargs^{sup\{n_j\}})$.

Since all integers are nonzero and $maxargs > 1$, $1 - maxargs^{sup\{n_j\}} < 0$. This proves the Lemma. □

Lemma 4.3 For each P_i in the transformation sequence the level mapping $|\ |$ satisfies the following.

(a) for each ground instance of a *defining* clause $H \leftarrow B_1, \ldots, B_k.$,
$|H| > |B_1| + \ldots + |B_k|$;

(b) for any other clause $H \leftarrow B_1, \ldots, B_k.$ in $Ground(P_i)$,
$|H| > |B_1| + \ldots + |B_k| + nf_i$.

Where for each j, nf_j is the number of folding operations that will be performed in the sequence from P_j to P_n.

Proof. The proof proceeds by induction on the index i.

Base Case: P_0.

Let $c : H \leftarrow B_1, \ldots, B_k.$ be a clause of $Ground(P_0)$. If $k = 0$ then the result holds trivially. So we assume $k > 0$. We have to distinguish two cases:

If H is a *new* predicate, then c is an instance of a *defining* clause, and condition (a) is then trivially satisfied by the definition of $|\ |$.

If H is an *old* predicate, then, since $||H|| > sup\{||B_j||\}$ and since $1 < k <$*maxargs*, the result follows from Lemma 4.2.

Induction Step: P_{i+1}.

For those clauses that P_i and P_{i+1} have in common, the result follows from the inductive hypothesis and the fact that $nf_{i+1} \leq nf_i$. Hence we can focus on those clauses that were introduced or modified in the last transformation step (from P_i to P_{i+1}). We distinguish upon the operation that has been used for going from P_i to P_{i+1}

Unfolding

Let
$d : H \leftarrow B', L_1, \ldots, L_h.$ be the unfolded clause, and
$c : B \leftarrow B_1, \ldots, B_k.$ be one of the unfolding ones.
Let also $\theta = mgu(B, B')$, then the resulting clause is
$H\theta \leftarrow B_1\theta, \ldots, B_k\theta, L_1\theta, \ldots, L_h\theta.$
Since $nf_{i+1} = nf_i$, in order to prove the thesis, we have to prove that, for each γ

$$|H\theta\gamma| > |B_1\theta\gamma| + \ldots + |B_k\theta\gamma| + |L_1\theta\gamma| + \ldots + |L_h\theta\gamma| + nf_i. \tag{4.1}$$

We have to distinguish two cases:

First we suppose that d is a *defining* clause, Then B is an old predicate and clause c satisfies condition (b), hence
$|B\theta\gamma| > |B_1\theta\gamma| + \ldots + |B_k\theta\gamma| + nf_i$.
On the other hand, clause d satisfies condition (a), hence
$|H\theta\gamma| > |B'\theta\gamma| + |L_1\theta\gamma| + \ldots + |L_h\theta\gamma|$.
Since $B'\theta\gamma = B\theta\gamma$ this proves (4.1).

Since we consider the case in which d is not a *defining* clause. Hence d satisfies condition (b), and we have that

$|H\theta\gamma| > |B'\theta\gamma| + |L_1\theta\gamma| + \ldots + |L_h\theta\gamma| + nf_i$.
Since clause c must satisfy either (a) or (b), we have that also
$|B\theta\gamma| > |B_1\theta\gamma| + \ldots + |B_k\theta\gamma|$.
Since $B'\theta\gamma = B\theta\gamma$ this proves again (4.1).

Folding

Suppose that:
$c:\ H \leftarrow B'_1, \ldots, B'_k, L_1, \ldots, L_h.$ is the folded clause of P_i,
$d:\ N \leftarrow B_1, \ldots, B_k$ is the folding clause of P_{new}.
Hence $(B'_1, \ldots, B'_k) = (B_1, \ldots, B_k)\tau$, and $H \leftarrow N\tau, L_1, \ldots, L_h.$ is the clause we add to P_{i+1}.

By **F4**, c is not a *defining* clause, hence its ground instances have to satisfy condition (b), that is, for each γ, $|H\gamma| > |B'_1\gamma| + \ldots + |B'_k\gamma| + |L_1\gamma| \ldots + |L_h\gamma| + nf_i$. Since $(B'_1, \ldots, B'_k) = (B_1, \ldots, B_k)\tau$, this implies that, for each γ,
$|H\gamma| > |B_1\tau\gamma| + \ldots + |B_k\tau\gamma| + |L_1\gamma| \ldots + |L_h\gamma|. + nf_i$,
where τ is a renaming on the variables in $W = Var(B_1, \ldots, B_k) \backslash Var(N)$. Let $Z = W\tau$, by the assumptions in **F2**, $Var(H, L_1, \ldots, L_h) \cap Z = \emptyset$. Hence we can split γ into two independent orthogonal substitutions: $\gamma = \gamma_{|Z}\gamma_{|\bar{Z}}$, where $\gamma_{|Z}$ is γ restricted to Z, and $\gamma_{|\bar{Z}}$ is γ restricted to the complement of Z. And we have that, for each γ,
$|H\gamma_{|\bar{Z}}| > |B_1\tau\gamma_{|\bar{Z}}\gamma_{|Z}| + \ldots + |B_k\tau\gamma_{|\bar{Z}}\gamma_{|Z}| + |L_1\gamma_{|\bar{Z}}| + \ldots + |L_h\gamma_{|\bar{Z}}| + nf_i$.
Since this holds for any choice of $\gamma_{|Z}$, for each γ
$|H\gamma_{|\bar{Z}}| > sup\{\sum_{i=1}^{k} |B_i\tau\gamma_{|\bar{Z}}\eta| \mid Dom(\eta) = Z\} + |L_1\gamma_{|\bar{Z}}| + \ldots + |L_h\gamma_{|\bar{Z}}| + nf_i$.
Now by **F3** d is the only clause whose head unifies with $N\tau$; it follows that, by the definition of $|\ |$, $|N\tau\gamma_{|\bar{Z}}| = sup\{\sum_{i=1}^{k} |B_i\tau\eta|\} + 1$, hence we have that, for each γ,
$|H\gamma_{|\bar{Z}}| > |N\tau\gamma_{|\bar{Z}}| + |L_1\gamma_{|\bar{Z}}| + \ldots + |L_h\gamma_{|\bar{Z}}| + nf_i - 1$.
Now the variables of Z do not occur in any atom of this clause we have that, for each γ
$|H\gamma| > |N\tau\gamma| + |L_1\gamma| + \ldots + |L_h\gamma| + nf_i - 1$
Since this is a folding step, $nf_{i+1} < nf_i$ and hence we have that (b) is satisfied in P_{i+1}. □

This implies immediately the desired conclusion

Corollary 4.4 Let $P_0, \ldots, P_n$ be a transformation sequence, then

(a) if P_0 is acyclic then P_n is.

In the case that P_0 is a definite program, this can be restated as follows

(b) if P_0 is definite and terminating, then P_n is.

Proof. It follows at once from Lemma 4.3 □

Transforming left-terminating programs One would like Corollary 4.4b to hold also in the case of *left terminating* programs, which are those programs whose LDNF (SLDNF with leftmost selection rule) derivations starting in a ground goal are finite. *Left terminating* programs form an important superclass of the terminating programs and, as pointed out by Apt and Pedreschi [AP93], there are natural left terminating programs that are not terminating. However, left-termination is not preserved by the transformation system. This is simply due to the fact that the definition of transformation sequence is given modulo reordering of the bodies of the clauses, and the operation of reordering itself does not preserve left-termination.

Example 4.5 Let $P_0 = P_{old} \cup P_{new}$, be the following program:

$$
\begin{array}{lll}
P_{old} = \{c_1 : & p & \leftarrow q(X), h(X). \\
 & q(s(0)). & \\
 & h(s(X)) & \leftarrow h(X). \qquad \} \\
 & & \\
P_{new} = \{c_2 : & d(X) & \leftarrow h(X), q(X). \qquad \}
\end{array}
$$

It is easy to verify that the program is left-terminating. However, if we fold $q(X), h(X)$ in the body of c_1, then the resulting program will be

$$
\begin{array}{llll}
P_2 = \{ & c_1 : & p & \leftarrow d(X). \\
 & c_2 : & d(X) & \leftarrow h(X), q(X). \\
 & & q(s(0)). & \\
 & & h(s(X)) & \leftarrow h(X). \qquad \}
\end{array}
$$

Now the goal $\leftarrow p$ originates an infinite LDNF-derivation. Hence left termination is not preserved. □

This shows that in general left termination is not preserved along the transformation sequence, and that this applies also when we adopt Seki's (more restrictive) *modified* folding operation.

It can be argued that, since the reason why left-termination is not preserved is because the transformation system is defined modulo reordering, then what we have to do is to restate the definition of unfolding and folding so that the order of the literals in the bodies of the clauses is taken into account. That is indeed a possible approach, however a fold operation so defined would be far more restrictive than the present one; in fact we would have to require that the literals that are going to be folded are all found next to each other in the exact same sequence as in the body of the folding clause. This is often not the case, in particular when the folded clause is the result of some previous unfold operation; notice that this is what happens in Example 3.2.

However, we can relax the requirement of the acyclicity of the initial program, by exploiting the result in a modular way. For this, we have to use the concept of *acceptable* programs, introduced by Apt and Pedreschi in [AP93]. Informally, acceptable are to left terminating programs what acyclic are to terminating ones, in fact in [AP93] is proven that, in cases of non-floundering programs, the classes of acceptable and of left terminating programs coincide. It is easy to prove that if the initial program is acceptable wrt the level mapping $|\ |$ and the model M and if the transformation is performed within a subset of P which is acyclic wrt $|\ |$, then the resulting program is acceptable (hence left-terminating) as well.

5. Semantic Consequences

5.1 Preliminaries: three-valued model semantics

In this section we refer to a fixed but unspecified language $\mathcal{L}$ that we assume contains all the functions symbols and the predicate symbols of the programs that we consider. We also refer to the usual Clark's completion definition, $Comp(P)$, [Cla78] which consists of the completed definition of each predicate together with CET, Clark's Equality Theory, which is needed in order to interpret "=" correctly. When working with 3-valued logic, the same definition

applies, with the only difference that the connective $\leftrightarrow$, used in the completed definitions of the predicates, is replaced with $\Leftrightarrow$, Lucasiewicz's operator of "having the same truth value". In this context, we have that a *three valued (or partial) interpretation*, is a mapping from the ground atoms of $\mathcal{L}$ into the set $\{true, false, undefined\}$.

We can now give the definition of Fitting's operator [Fit85].

Definition 5.1 Let P be a normal program, I a three valued interpretation, A a ground atom; $\Phi_P(I)$ is the three valued interpretation defined as follows:

- A is *true* in $\Phi_P(I)$, iff there exists a clause $c: \ A \leftarrow \tilde{L}.$ in $Ground(P)$ such that $\tilde{L}$ is *true* in I;
- A is *false* in $\Phi_P(I)$, iff for all clauses $c: \ A \leftarrow \tilde{L}.$ in $Ground(P)$, $\tilde{L}$ is *false* in I. □

We adopt the standard notation: $\Phi_P^{\uparrow 0}$ is the interpretation that maps every ground atom into the value *undefined*, $\Phi_P^{\uparrow \alpha+1} = \Phi_P(\Phi_P^{\uparrow \alpha})$, $\Phi_P^{\uparrow \alpha} = \cup_{\delta<\alpha}\Phi_P^{\uparrow \delta}$, when α is a limit ordinal. Φ_P is a monotonic operator, it follows that its Kleene's sequence is monotonically increasing and it converges to the least fixpoint of Φ_P. Hence there always exists an ordinal α such that $lfp(\Phi_P) = \Phi_P^{\uparrow \alpha}$. Since Φ_P is monotone but not continuous, α could be greater than ω.

Φ_P characterizes the three valued semantics of $Comp(P)$, in fact Fitting, in [Fit85] shows that the three-valued models of P are exactly the fixpoints of Φ_P; it follows that any program has a least three-valued Herbrand model. This model is usually referred to as Fitting's model.

5.2 Semantics of Acyclic Programs

From the point of view of declarative semantics, acyclic programs enjoy various relevant properties. Before stating them, we need to introduce some domain closure axioms, often referred to as "weak domain closure axioms".

Definition 5.2 $\mathrm{DCA}_{\mathcal{L}}$ is the axiom $\exists \tilde{y}_1 \ (x = f_1(\tilde{y}_1)) \vee \ldots \vee \exists (\tilde{y}_r \ x = f_r(\tilde{y}_r))$. where $f_1, \ldots, f_r$ are all the function symbols in the language $\mathcal{L}$ and $\tilde{y}_i$ are tuples of variables of the appropriate arity. □

Now we summarize some of the semantic properties of acyclic programs. For the definition and the properties of the Well-Founded model semantics we refer to [GRS88].

Theorem 5.3 Let P be an acyclic program, and let $M = \Phi_P^{\uparrow \omega}$. Then M is total, that is, no atom is undefined in it, moreover

(i) M is the unique fixpoint of Φ_P; hence it is the unique three-valued (and also two-valued) Herbrand model of $Comp(P)$ and coincides with Fitting's model of P.

(ii) M coincides with the Well-Founded model of P;

(iii) M coincides with the set of ground atomic logical consequences of $Comp(P) \cup \mathrm{DCA}_{\mathcal{L}}$ in 2 and 3 valued logic;

(iv) for all ground atoms A such that no SLDNF-derivation of $P \cup \{ \leftarrow A\}$ flounders,

A is *true* in M iff there exists a SLDNF-refutation for $P \cup \{ \leftarrow A\}$;
A is *false* in M iff $P \cup \{ \leftarrow A\}$ has a finitely failed SLDNF tree.

Proof. The fact that M is total and statement (i) are proven in [AP93], where the more general case of *acceptable* programs is considered; (ii) is a consequence if (i) and the fact that the Well-Founded model is also a three-valued model of $Comp(P)$ [GRS88]; (iii) and (iv) are consequences of Theorem 4.4 in [AB91]. □

5.3 Semantics of transformed programs
An immediate consequence of Theorem 5.3 is the following.

Lemma 5.4 Let $P_0, \ldots, P_n$ be a transformation sequence, suppose that P_0 is acyclic, then $\Phi_{P_0}^{\uparrow\omega} = \Phi_{P_n}^{\uparrow\omega}$.

Proof. By Theorem 5.3, for each i, the Well-Founded model of P_i coincides with $\Phi_{P_i}^{\uparrow\omega}$ and by Proposition 4.1 in [Sek93], the Well-Founded models of P_0 and P_n coincide. □

Because of Theorem 5.3, Corollary 4.4 has also some semantic consequences, the most relevant of which are:

Corollary 5.5 Let $P_0, \ldots, P_n$ be a transformation sequence, suppose that P_0 is acyclic, then

(a) the Fitting's models of P_0 and of P_n coincide;

(b) the set of ground logical consequences of $Comp(P_0) \cup \mathrm{DCA}_{\mathcal{L}}$ and of $Comp(P_n) \cup \mathrm{DCA}_{\mathcal{L}}$ coincide;

(c) for all ground atoms A such that no SLDNF-derivation of $P_0 \cup \{ \leftarrow A\}$ and of $P_n \cup \{ \leftarrow A\}$ flounders,

- there exists a SLDNF-refutation for $P_0 \cup \{ \leftarrow A\}$ iff there exists one for $P_n \cup \{ \leftarrow A\}$,
- all SLDNF trees for $P_0 \cup \{ \leftarrow A\}$ are finitely failed iff all SLDNF trees for $P_n \cup \{ \leftarrow A\}$ are;

in particular we have that

(d) If P_0 is definite, then its Finite Failure Set coincides with the one of P_n. □

This shows that if the initial program is acyclic, then the transformation enjoys most of the properties that were proven for Seki's more restrictive modified folding. In some situations this can be useful for relaxing the applicability of the folding operation.

Acknowledgements The authors express their gratitude to Prof. K. R. Apt and to Maurizio Gabbrielli for their useful suggestions.

REFERENCES

[AB91] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 29(3):335–363, 1991.

[AD93] C. Aravidan and P. M. Dung. On the correctness of Unfold/Fold transformation of normal and extended loginc programs. Technical report, Division of Computer Science, Asian Institute of Technology, Bangkok, Thailand, April 1993.

[AP93] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[Apt90] K. R. Apt. Introduction to 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.

[BC93] A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87, 1993.

[Bez93] M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, (15):79–97, 1993.

[Cav91] L. Cavedon. Acyclic programs and the completeness of SLDNF-resolution. *Journal of Theoretical Computer Science*, 86:81–92, 1991.

[Cla78] K. L. Clark. Negation as failure rule. In H. Gallaire and G. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[Fit85] M. Fitting. A Kripke-Kleene semantics for Logic Programs. *Journal of Logic Programming*, (4), 1985.

[GRS88] A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and the Well-Founded Semantics for General Logic Programs. In *Proc. Seventh ACM symposium on Principles of Database System*, pages 211–230, 1988.

[KK88] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 413–422. Institute for New Generation Computer Technology, Tokyo, 1988.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

[Mah87] M.J. Maher. Correctness of a logic program transformation system. IBM Research Report RC13496, T.J. Watson Research Center, 1987.

[Prz89] T. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the Eighth Symposium on Principles of Database Systems*, pages 11–21. ACM SIGACT-SIGMOD, 1989.

[Sat90] T. Sato. An equivalence preserving first order unfold/fold transformation system. In *Second Int. Conference on Algebraic and Logic Programming, Nancy, France, October 1990, (Lecture Notes in Computer Science, Vol. 463)*, pages 175–188. Springer-Verlag, 1990.

[Sek89] H. Seki. Unfold/fold transformation of stratified programs. In G. Levi and M. Martelli, editors, *6th International Conference on Logic Programming*, pages 554–568. The MIT Press, 1989.

[Sek90] H. Seki. A comparative study of the Well-Founded and Stable model semantics: Transformation's viewpoint. In D. Pedreschi W. Marek, A. Nerode and V.S. Subrahmanian, editors, *Workshop on Logic Programming and Non-Monotonic Logic, Austin, Texas, October 1990*, pages 115–123, 1990.

[Sek91] H. Seki. Unfold/fold transformation of stratified programs. *Journal of Theoretical Computer Science*, 86:107–139, 1991.

[Sek93] H. Seki. Unfold/fold transformation of general logic programs for the Well-Founded semantics. *Journal of Logic Programming*, 16:5–23, 1993.

[TS84] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.