Transformation of Left Terminating Programs

Annalisa Bossi\textsuperscript{1}, Nicoletta Cocco\textsuperscript{1}, and Sandro Etalle\textsuperscript{2}

\textsuperscript{1} Dip. di Informatica, Università di Venezia-Ca' Foscari - Italy
\{bossi,cocco\}@dsi.unive.it
\textsuperscript{2} Dept. of Computer Science, University of Maastricht - The Netherlands
etalle@cs.unimaas.nl

Abstract. We propose an unfold-fold transformation system which preserves left termination for definite programs besides its declarative semantics. The system extends our previous proposal in [BCE95] by allowing to switch the atoms in the clause bodies when a specific applicability condition is satisfied. The applicability condition is very simple to verify, yet very common in practice. We also discuss how to verify such condition by exploiting mode information.

1 Introduction

Tamaki and Sato [TS84] proposed an elegant framework for the transformation of logic programs based on unfold/fold rules which is the main reference point in the literature of transformations of logic programs.

However Tamaki-Sato's method often cannot be applied "as it is" to pure Prolog programs, as it does not preserve left-termination, i.e., termination wrt the leftmost selection rule. In other words, it can happen that a left-terminating program (a program whose derivations starting in a ground query are all finite under the leftmost selection rule) be transformed into a non-terminating one.

In this paper we present a new transformation system for definite logic programs which preserves left termination.

As we already remarked in [BCE95], in order to maintain transformation properties, a transformation system has to take into account the order of the atoms in the clause bodies. Moreover, in order to achieve a reasonable degree of flexibility, the system should provide special operations for reordering them. These special operations are critical as they can easily spoil termination. For this, the system we propose is provided with a switch operation, together with suitable applicability conditions, which are very simple to verify, yet rather common in practice. These conditions are particularly powerful in the case that the transformed programs are well-moded.

The present system extends the one we proposed in [BCE95], we also show how the two methods can be profitably combined together.

Structure of the paper Section 2 contains the notation and the preliminaries on left terminating programs. In Section 3 we define the basic unfold/fold
transformation system. In Section 4 we introduce the reordering problem and the applicability condition for switching atoms in the bodies. The transformation system extended with such “allowed switch” can be proved to preserve left termination. In Section 5 we consider well-moded programs and extend the condition for allowed switching to such programs. Such extended transformation system for well-moded programs preserves left termination and it is applicable to most practical cases, as discussed in Section 6. Conclusions and some discussion on related papers follow in Section 7.

2 Preliminaries

In what follows we study definite logic programs executed by means of LD-resolution, namely SLD-resolution with the leftmost selection rule (the Prolog selection rule).

We use bold characters (es. B) to indicate sequences of objects, typically B indicates a sequence of atoms, B_1, ..., B_n, t indicates a sequence of terms, t_1, ..., t_n, and x denotes a sequence of variables, x_1, ..., x_n.

We work with queries that is sequences of atoms, B_1, ..., B_n, instead of goals. Apart from this, we use the standard notation of Lloyd [Llo87] and Apt [Apt97]. In particular, given a syntactic construct E (so for example, a term, an atom or a set of equations) we denote by Var(E) the set of the variables appearing in E. Given a substitution \( \theta = \{x_1/t_1, ..., x_n/t_n\} \) we denote by Dom(\( \theta \)) the set of variables \( \{x_1, ..., x_n\} \), and by Ran(\( \theta \)) the set of variables appearing in \( \{t_1, ..., t_n\} \). Finally, we define Var(\( \theta \)) = Dom(\( \theta \)) \cup Ran(\( \theta \)).

A substitution \( \theta \) is called grounding if Ran(\( \theta \)) is empty, and it is called a renaming if it is a permutation of the variables in Dom(\( \theta \)). Given a substitution \( \theta \) and a set (sequence) of variables \( v \), we denote by \( \theta_v \) the substitution obtained from \( \theta \) by restricting its domain to \( v \). By Pred(\( E \)) we denote the set of predicate symbols occurring in the expression \( E \).

We use the notation introduced in [AP93,Apt97], and we say that a predicate \( p \) is defined in the program \( P \) iff there is a clause in \( P \) that uses \( p \) in its head. Given two relations \( p, q \) in \( \text{Pred}(P) \), we say that \( p \) refers to \( q \) in \( P \) if there is a clause in \( P \) that uses \( p \) in its head and \( q \) in its body. We say that \( p \) depends on \( q \) in \( P \), if \( (p,q) \) is in the reflexive, transitive closure of the relation refers to.

Let \( P, Q \) be programs which define different predicates, we say that \( P \) extends \( Q, P \triangleright Q \), if there is no \( q \in \text{Pred}(Q) \) which refers (in \( Q \)) to a predicate \( p \) defined in \( P \). Notice that \( \triangleright \) is a well-founded ordering. We harmlessly extend this notation to atoms and programs. Let \( B \) be an atom, by \( P|_B \) we denote the set of clauses of \( P \) that define the predicates which the predicate of \( B \) depends on. Similarly by \( P|_p \) we denote the set of clauses of \( P \) that define a predicate \( p \) and all the predicates which it depends on.
2.1 Acceptable and Left Terminating Programs

The class of left terminating definite logic program has been characterized by Apt and Pedreschi in [AP90]. This is done by using the definition of acceptable programs. We briefly restate the results of [AP90].

Definition 1 (Left Terminating Program). A program $P$ is called left terminating if all LD-derivations of $P$ starting in a ground query are finite.

The basic tool used to prove (and characterize) that a program is left terminating is the concept of level mapping originally due to Bezem [Bez88] and Cavedon [Cav89]. A level mapping for a program $P$ is a function $| | : B_P \to \mathbb{N}$, from the ground atoms of the Herbrand base of $P$ to natural numbers. We call $|B|$ the norm of $B$ wrt the level mapping $| |$. We omit the reference to $P$ when the choice of the language does not play a significant role.

We can now provide the central definition of [AP90].

Definition 2 (Acceptable Program). Let $P$ be a program, $| |$ a level mapping for $P$ and $I$ a (not necessarily Herbrand) interpretation of $P$.

- A clause of $P$ is acceptable with respect to $| |$ and $I$ if $I$ is a model of $P$ and for every ground instance $H \leftarrow B_1, \ldots, B_m$ of the clause, the following implication holds:
  $$\text{if } I \models B_1, \ldots, B_{i-1} \text{ then } |H| > |B_i|.$$  
- $P$ is acceptable with respect to $| |$ and $I$ if all its clauses are.
- $P$ is called acceptable if it is acceptable with respect to some level mapping and interpretation of $P$.

The use of acceptable programs is motivated by the following result of Apt and Pedreschi [AP93]

Theorem 3. A definite program is left terminating iff it is acceptable.

Here, we also need a new definition for a stronger termination property.

Definition 4 (Always-Left-Terminating Program). A program $P$ is called always-left-terminating if all LD-derivations of $P$ starting in any query are finite.

In an always-left-terminating program, no computation can diverge. They are generally defined by clauses which are not recursive or by built-ins and used to perform some checks. Clearly an always-left-terminating program is also left terminating.
2.2 Modes, Well-Moded Programs and Termination

The transformation system we propose is provided with special applicability conditions for the case in which the considered programs are well-moded. We now give the basic definitions we refer to in the sequel.

Modes are extensively used in the literature on logic programs, usually they indicate how the arguments of a relation should be used.

Definition 5 (Mode). Consider an $n$-ary predicate symbol $p$. By a mode for $p$ we mean a function $m_p$ from $\{1, \ldots, n\}$ to the set $\{+, -\}$. If $m_p(i) = '+'$, we call $i$ an input position of $p$ and if $m_p(i) = '-'$, we call $i$ an output position of $p$ (both w.r.t. $m_p$).

Most predicates have a natural moding, which reflects their intended use. For example, the natural moding for the usual program append, when used for concatenating two lists, is $\text{app}(+, +, -)$; the natural moding for the predicates in Example 10 are: $\text{path}(+,-)$, $\text{arc}(+,-)$, $\text{goodlist}(+)$, $\text{goodpath}(+,-)$.

We say that a program is moded if each predicate symbol has a mode associated to it. Multiple modes may be obtained by simply renaming the predicates. When every considered predicate has a mode associated with it, we can talk about input positions and output positions of an atom.

The concept of well-moded program is essentially due to Dembinski and Maluszynski [DM85]. To simplify the notation, when writing an atom as $p(u, v)$, we are indicating with $u$ the sequence of terms filling in the input positions of $p$ and with $v$ the sequence of terms filling in the output positions of $p$.

Definition 6 (Well-Moded Query, Program). A clause $p_0(t_0, s_{n+1}) \leftarrow p_1(s_1, t_1), \ldots, p_n(s_n, t_n)$ is called well-moded if for $i \in [1, n + 1]$

$$ \text{Var}(s_i) \subseteq \bigcup_{j=0}^{i-1} \text{Var}(t_j) $$

A query $A$ is called well-moded if the clause $q \leftarrow A$ is well-moded, where $q$ is any (dummy) atom of zero arity.

A program is called well-moded if every clause of it is well-moded.

Note that the first atom of a well-moded query is ground in its input positions and a variant of a well-moded clause is well-moded. The following lemma, due to [AM94], shows the “persistence” of the notion of well-modedness.

Lemma 7. An LD-resolvent of a well-moded query and a well-moded clause with no variable in common, is well-moded.

The relevance of both modes and well-moding for termination is studied in [EBC99]. Where the following concept of well-terminating program is defined as follows:
Definition 8 (Well-Terminating Program). A program is called well-terminating iff all its LD-derivations starting in a well-moded goal are finite. □

Clearly a well-terminating program is also left terminating. In [EBC99] it is also shown that if a well-moded program is acceptable wrt a moded level mapping, namely a level mapping which depends only on the terms filling in input positions, then it is well-terminating (Actually, in [EBC99] it is shown that this holds also when employing the much weaker definition of weakly acceptable program).

3 An Unfold/Fold Transformation System

In this section we introduce a new unfold/fold transformation system. We start from the requirements on the initial program which we assume to be divisible into a hierarchy of modules. Here and in the sequel, standardization apart is always assumed.

Definition 9 (Initial Program). We call a definite program $P_0$ an initial program if it can be partitioned into three programs $P_{new}$, $P_{old}$ and $P_{base}$, such that the following conditions are satisfied:

(I1) $P_{new} \supset (P_{old} \cup P_{base})$ and $P_{old} \supset P_{base}$.

(I2) $P_{new}$ is not recursive.

(I3) all the atoms in the bodies of the clauses of $P_{old}$ are labelled “f”, with the exception of atoms defined in $P_{base}$; no other atom of the initial program is labelled. □

Predicate (atoms) defined in $P_{new}$ are called new predicates (atoms), those defined in $P_{old}$ are called the old ones, while those defined in $P_{base}$ are called the base predicates. The reason of this partition into modules will be clear in the sequel of the paper. Notice that the combination of I1 and I2 guarantees that all the predicates which are defined in $P_{new}$ occur neither in $P_{old} \cup P_{base}$ nor in the bodies of the clauses in $P_{new}$, which is similar to the condition on the initial program provided in [TS84]. We label with “f” (for fold-allowing) only old atoms in the bodies of $P_{old}$.

The new element of this definition w.r.t. its predecessors ([TS84, Sek93, EG96]) lies in the use of the labelling and in the fact that we distinguish $P_{base}$, which is the part of the initial program which is not modified by the transformation. These elements will be needed to guarantee the preservation of left termination.

The following example is inspired by the one in [Sek93].

Example 10. Let $P_0$ contain the following clauses

$c1$: path($X, X, [X]$).
$c2$: path($X, Z, [X | Xs]$) $\leftarrow$ arc($X, Y$), path($Y, Z, Xs)$.
$c3$: goodlist($[]$).
$c4$: goodlist($[X | Xs]$) $\leftarrow$ good($X$), goodlist($Xs$).
$c5$: goodpath($X, Z, Xs$) $\leftarrow$ path($X, Z, Xs$), goodlist($Xs$).
where \( P_{\text{old}} = \{c_1, \ldots, c_4\} \) and \( P_{\text{new}} = \{c_5\} \), thus \texttt{goodpath} is the only new predicate, while the predicates \texttt{arc} and \texttt{good} are defined in \( P_{\text{base}} \) by facts.

The query \texttt{goodpath(X, Z, Xs)} can be employed for finding a path \( Xs \) starting in the node \( X \) and ending in the node \( Z \) which contains exclusively “good” nodes. Under the assumption that the graph described by the predicate \texttt{arc} is acyclic, this program is left terminating. \texttt{goodpath} works on a “generate and test” basis. We can obtain an efficiency improvement via an unfold/fold transformation. \( \Box \)

According to a well-established transformation strategy, the first operation we apply is unfold. Unfold is the fundamental operation for partial evaluation [LS91] and consists in applying a resolution step to a selected atom using all possible resolving clauses. Here—as opposed to the definition adopted in [TS84] and in most of the literature—the \textit{order of the atoms in the queries and in the bodies of the clauses is relevant}. This is natural, since we are dealing with\textit{LD-resolution}.

\textbf{Definition 11 (Unfold).} Let \( cl : H \leftarrow J, A, K \) be a clause of a program \( P \), and \( \{A_1 \leftarrow B_1, \ldots, A_n \leftarrow B_n\} \) be the set of clauses of \( P \) whose heads unify with \( A \), by \texttt{mgu}'s \( \{\theta_1, \ldots, \theta_n\} \).

\begin{itemize}
\item Unfolding \( A \) in \( cl \) consists of substituting \( cl \) by \( \{cl'_1, \ldots, cl'_n\} \),
\item where, for each \( i \), \( cl'_i = (H \leftarrow J, B_i, K)\theta_i \).
\end{itemize}

\( \Box \)

The unfold operation doesn’t modify the labels of the atoms, no matter if the unfolded atom itself is labelled or not. Notice that unfold propagates the labels inside the clauses in the obvious way, as shown by the following example.

\textbf{Example 10 (part 2).} By unfolding the atom \texttt{path(X, Z, Xs)} in the body of \texttt{c5}, we obtain

\begin{verbatim}
c6: goodpath(X, X, [X]) ← goodlist([X]).
c7: goodpath(X, Z, [X|Xs]) ← arc(X, Y),
    path(Y, Z, Xs),
    goodlist([X|Xs]).
\end{verbatim}

In the above clauses we can unfold \texttt{goodlist([X])} and \texttt{goodlist([X|Xs])}. The resulting clauses, after a further unfolding of \texttt{goodlist(□)} in the clause obtained from \texttt{c6}, are

\begin{verbatim}
c8: goodpath(X, X, [X]) ← good(X).
c9: goodpath(X, Z, [X|Xs]) ← arc(X, Y),
    path(Y, Z, Xs),
    good(X),
    goodlist(Xs).
\end{verbatim}

\( \Box \)

Thanks to its correspondence to a resolution step, the unfold operation preserves basically all the declarative semantics available for logic programs. It has also already been proven in [BC94] that it preserves universal termination of
a query, namely any query with a finite LD-tree in a program \( P \) has a finite LD-tree also in a program obtained by unfolding \( P \). Hence unfold preserves also the property of being left terminating.

Now we have reached a crucial step in the transformation: in order to be able to perform the fold operation, we need to permute the atoms \( \text{path}(Y, Xs)^f \) and \( \text{good}(X) \) in c9. For this, we introduce the switch operation.

**Definition 12 (Switch).** Let \( cl : H \leftarrow J, A, B, K \). be a clause of a program \( P \). Switching \( A \) with \( B \) in \( cl \) consists of replacing \( cl \) by \( cl' : H \leftarrow J, B, A, K \). \( \square \)

**Example 10 (part 3).** By permuting \( \text{path}(Y, Z, Xs)^f \) with \( \text{good}(X) \) in c9 we obtain the following clause:

\[
c10: \text{goodpath}(X, Z, [X|Xs]) \leftarrow \text{arc}(X, Y),
\text{good}(X),
\text{path}(Y, Z, Xs)^f,
\text{goodlist}(Xs)^f.
\]

\( \square \)

Fold is the inverse of unfold (when one single clause unifies with the atom to be unfolded). This operation is used in all the transformation systems in order to fold back unfolded clauses and to introduce direct recursion on the definitions. As in Tamaki and Sato [TS84], the transformation sequence and the fold operation are defined in terms of each other.

**Definition 13 (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 applying an unfold, a switch, or a fold operation to a clause of \( P_i \setminus P_{\text{base}} \).

\( \square \)

**Definition 14 (Fold).** Let \( P_0, \ldots, P_i, i \geq 0 \), be a transformation sequence, \( cl : H \leftarrow J, B, K \). be a clause in \( P_i \setminus P_{\text{base}} \), and \( d : D \leftarrow B' \). be a clause in \( P_{\text{new}} \). Folding \( B \) in \( cl \) via \( \tau \) consists of replacing \( cl \) by \( cl' : H \leftarrow J, Dr, K \), provided that \( \tau \) is a substitution such that \( \text{Dom}(\tau) = \text{Var}(d) \) and such that the following conditions hold:

- **(F1)** \( d \) is the only clause in \( P_{\text{new}} \) whose head is unifiable with \( Dr \);
- **(F2)** if we unfold \( Dr \) in \( cl' \), then the result of the operation is a variant (obtained by renaming) of \( cl \);
- **(F3)** one of the atoms in \( J \), or the leftmost of the atoms in \( B \) is labelled “f”. \( \square \)

The conjunction of conditions **F1** and **F2** corresponds to the conjunction of the conditions 1...3 of [TS84]. Therefore, apart from the fact that we take into consideration the order of the atoms in the bodies of the clauses, what distinguishes this fold definition from the one in [TS84] is the further condition **F3**. Note that condition **F3** is less restrictive than the similar one introduced by Seki [Sek91] to preserve finite failures\(^1\). The label “f” indicates that a previous

---

\(^1\) Seki's system requires that all folded atoms be the result of a previous unfolding, moreover it partitions the initial program in \( P_{\text{new}} \) and \( P_{\text{old}} \) only.
unfolding step of an \textit{old} atom has occurred in the transformation sequence and this is used to ensure that a fold operation cannot introduce loops. Notice that fold eliminates the labels in the folded part of the body.

\textbf{Example 10 (part 4)} We can now fold $\text{path}(Y,Z,X_0)^f, \text{goodlist}(X_0)^f$ in 10. The resulting new definition of $\text{goodpath}$ is

\begin{verbatim}
c8:  goodpath(X,X,[X]) ← good(X).
c11: goodpath(X,Z,[X|X]) ← arc(X,Y),
     good(X),
     goodpath(Y,Z,X).
\end{verbatim}

Notice that the definition of \textit{goodpath} is now recursive and it checks the “goodness” of the path while generating the path itself. \hfill \Box

\textbf{Correctness result, declarative viewpoint} Of course, it is of primary importance ensuring the correctness of the system from a declarative point of view. In the case of our system, the applicability conditions are more restrictive than those used by Tamaki-Sato in [TS84] (which, on the other hand, does not guarantee the preservation of left termination). For this reason, all the correctness results for the declarative semantics that hold for the system of [TS84] are valid for our system as well and we have the following.

\textit{Remark 15.} Let $P_0, \ldots, P_n$ be a transformation sequence.

- [TS84] The least Herbrand models of the initial and final programs coincide.
- [KK90] The computed answers substitution semantics of the initial and final programs coincide. \hfill \Box

\section{Preservation of Left Termination}

The system presented in the previous section does not preserve left-termination yet. This is because the switching operation is \textit{unrestricted}: It has no special applicability conditions.

In this section we first analyze the reasons why in general an unfold/fold transformation system does not preserve termination, then we provide a simple condition on the switch operation that will ensure the preservation of the left termination property.

\textbf{The reordering problem} As already explained in the introduction, the rearrangement of the atoms in the body of a clause is a typical operation which does not preserve left termination. For instance if we take the program

\begin{verbatim}
p ← q, p.
\end{verbatim}
We have that at the moment the program is terminating (q fails), however, if we swap the two atoms in the body of the clause, we get a program which is not terminating. Another typical situation is the one in which we have in the body of a clause a combination such as \( \ldots p(X, Y), q(Y, Z) \ldots \), where the rightmost atom uses \( Y \) as input variable; in this case, bringing \( q(Y, Z) \) to the left of \( p(X, Y) \) can easily introduce non-termination, as \( q(Y, Z) \) might be called with its arguments not sufficiently instantiated.

In the context of an unfold/fold transformation system, this situation is further complicated by the presence of the other operations, in particular of fold which may introduce recursion and hence non-termination. Consider the following example taken from [BCE96].

**Example 16.** Let \( P_6 \) be the program

\[
\begin{align*}
c1: & \quad z \leftarrow p, \ x. \\
c2: & \quad p \leftarrow q^f, \ x. \\
c3: & \quad q \leftarrow r, \ p^f.
\end{align*}
\]

Where \( P_{new} = \{c1\}, P_{old} = \{c2, c3\} \) and \( r \) is a base predicate which fails. Notice that this program is left terminating. By unfolding \( p \) in \( c1 \) we obtain the following clause:

\[
c4: \quad z \leftarrow q^f, \ r, r.
\]

By further unfolding \( q \) in \( c4 \) we obtain:

\[
c5: \quad z \leftarrow r, p^f, r, r.
\]

Now if we permute the first two atoms, we get:

\[
c6: \quad z \leftarrow p^f, r, r, r.
\]

Notice that this switch operation does preserve left termination. However, if we now fold the first two atoms, using clause \( c1 \) for folding, we obtain the following:

\[
c7: \quad z \leftarrow z, r, r.
\]

which is obviously not left terminating. \( \square \)

In this example the fold operation satisfies our applicability conditions and the applicability conditions of both Tamaki-Sato’s [TS84] and Seki’s [Sek91] systems. This shows the need of introducing extra applicability conditions for the switch operation. In the above example the switch operation does preserve left termination, while left termination is subsequently spoiled by the application of the fold operation.

Now, we present a new applicability condition for switching. This condition is extremely simple, yet rather common in practice. We call this operation an **allowed switch** since, when \( P_{base} \) has an appropriate termination property, it guarantees the preservation of left termination in a transformation sequence, namely also after folding.
Definition 17 (Allowed Switching, (Condition SW1)). Let \( P_0, \ldots, P_n \) be a transformation sequence. Suppose that, for some \( i \), \( P_{i+1} \) is obtained from \( P_i \) by switching A with B in the clause \( cl : H \leftarrow J, A, B, K \). This operation is allowed if B is a base atom. \( \square \)

Condition SW1 allows us to prove the following.

Theorem 18 (Main). Let \( P_0, \ldots, P_n \) be a transformation sequence. If \( P_{\text{base}} \) is always-left-terminating, \( P_0 \) is left terminating and every switch operation performed in \( P_0, \ldots, P_n \) satisfies condition SW1, then \( P_n \) is left terminating. \( \square \)

The intuition behind the above result is that if the base atoms cannot diverge (\( P_{\text{base}} \) always-left-terminating), then one is allowed to move them to the left (condition SW1) in the body of a clause. The fact that \( P_{\text{base}} \) cannot be modified by the transformation sequence further guarantees that the transformation does not spoil its own applicability condition by making base atoms non always-left-terminating.

The situation described by Theorem 18 is quite common when we transform programs via an unfold/fold transformation: The atoms we need to move leftward often perform some checking. Therefore they are usually defined by built-ins or by sets of unit clauses. Notice that it applies for instance to the transformation of Example 10: In it, the base predicates arc and good are defined by facts, and therefore \( P_{\text{base}} \) is always-terminating; moreover, the switch operation consists in shifting a base atom to the left.

The proof of Theorem 18 can be given by cases, since we have to consider each possible transformation operation. By induction on \( i, i \in [0, n] \), we can prove that each \( P_i \) is acceptable wrt a level mapping derived from the one of \( P_0 \). Moreover, we need to apply the following modularity result in order to guarantee that \( P_i \) is left terminating for \( i \in [0, n] \).

Proposition 19. Let \( P \) and \( Q \) be two programs such that \( P \) extends \( Q \). Let \( M \) be a model of \( P \cup Q \) such that \( M \) is a model of \( \text{Comp}(P \cup Q) \). Suppose that

(i) \( Q \) is always-left-terminating,
(ii) \( P \) is acceptable wrt \( M \) (and a level mapping \( | \nu \)).

Then \( P \cup Q \) is left terminating. \( \square \)

This result is based on the intuition that, given a ground query \( G \) in \( P \cup Q \), we can “ignore” the atoms defined in \( Q \) in the LD-derivations of \( G \), since they always terminate. Namely we can substitute the atoms in \( Q \) by the results of their finite LD-trees.

Finally, since for left terminating programs the Finite Failure set coincides with the complement of the least Herbrand model, we also have the following.

Corollary 20 (Preservation of finite failure). If the conditions of Theorem 18 are satisfied then the finite failure set of \( P_n \) coincides with the one of \( P_0 \). \( \square \)
The following is a simple example of transformation making use of such an allowed switch.

**Example 21.** Let \( P\text{\textsubscript{old}} \) be the left terminating program

\[
c1: \text{prefix}(Xs, []).
c2: \text{prefix}([X|Xs], [X|Ys]) \leftarrow \text{prefix}(Xs, Ys)^f.
c3: \text{eqTo}(\[], N).
c4: \text{eqTo}(\[N|Xs\], N) \leftarrow \text{eqTo}(Xs, N)^f.
c5: \text{pos}(\[]).
c6: \text{pos}([X|Xs]) \leftarrow X>0, \text{pos}(Xs)^f.
c7: \text{pre}(Ls, N, Ss) \leftarrow \text{prefix}(Ls, Ss)^f, \text{eqTo}(Ss, N)^f, \text{pos}(Ss)^f.
\]

where the predicate \( > \) is a built-in and then in \( P\text{\textsubscript{base}} \).

\( \text{pre}(Ls, N, Ss) \) states the property of \( Ss \) of being a prefix of \( Ls \) composed of elements which are all of value \( N \) and positive. In order to avoid scanning the list \( Ls \) many times, we can apply the following transformation sequence.

We introduce a new predicate

\[
d: \text{scan}(N, Ss) \leftarrow \text{eqTo}(Ss, N), \text{pos}(Ss).\
\]

Hence \( P_0 = \{d, c1, \ldots, c7\} \). We unfold both atoms in the body of \( d \)

\[
d1: \text{scan}(N, []).
d2: \text{scan}(N, [N|Xs]) \leftarrow \text{eqTo}(Xs, N)^f, N>0, \text{pos}(Xs)^f.
\]

In order to fold back in \( d2 \) with the definition \( d \), we need to switch \( \text{eqTo}(Xs, N)^f \) and \( N>0 \). Condition SW1 holds, hence we may safely perform the switch

\[
d3: \text{scan}(N, [N|Xs]) \leftarrow N>0, \text{eqTo}(Xs, N)^f, \text{pos}(Xs)^f.
\]

and then fold with definition \( d \) both in \( d3 \) and in \( c7 \). The resulting program \( P \) contains \( \{c3, \ldots, c6\} \) and

\[
c1: \text{prefix}(Xs, []).
c2: \text{prefix}([X|Xs], [X|Ys]) \leftarrow \text{prefix}(Xs, Ys)^f.
c6: \text{pos}([X|Xs]) \leftarrow X>0, \text{scan}(N, Xs).
c8: \text{pre}(Ls, N, Ss) \leftarrow \text{prefix}(Ls, Ss)^f, \text{scan}(N, Ss).
d1: \text{scan}(N, [])
d4: \text{scan}(N, [N|Xs]) \leftarrow N>0, \text{scan}(N, Xs).
\]

where \( \{c3, \ldots, c6\} \) are no more in \( P\text{\textsubscript{pre}} \).

We can continue the manipulation of the program through another transformation sequence. Now \( P_0 \) is the final program \( P\text{\textsubscript{pre}} \) and we want to further simplify it. The only base atom is \( N > 0 \) as before, so \( P\text{\textsubscript{old}} = \{c1, c2, d1, d4\} \), while the predicate \( \text{pre} \) is now in \( P\text{\textsubscript{new}} \). In accordance with this choice, we have to modify the labelling in \( d4 \) and \( c8 \). This means that we add the label "f" to \( \text{scan}(N, Ss) \) in the body of \( d4 \) while we cancel it to \( \text{pre}(Ls, N, Ss) \) in the body of \( c8 \).

Let us unfold both the atoms in the body of \( c8 \).
\[e_1: \text{pre}(L_s, N, \square)\].
\[e_2: \text{pre}(\text{prefix}(L_s', S_s')^f, N > 0, \text{scan}(N, S_s')^f).\]

In order to fold back in \(P_{\text{new}}\) with the definition in \(c_8\), we need to switch \(\text{prefix}(L_s', S_s')^f\) and \(N > 0\). \(\text{SW1}\) is trivially verified. Hence we can perform the switch

\[e_3: \text{pre}(\text{prefix}(L_s', S_s')^f, N > 0, \text{scan}(N, S_s')^f).\]

and then fold with \(c_8\).

\[e_1: \text{pre}(L_s, N, \square).
\[e_4: \text{pre}(\text{prefix}(L_s', S_s')^f, N > 0, \text{pre}(L_s', N, S_s')).\]

Let the final program be \(P'\), no other clause is in \(P|_{\text{pre}}\).

In this case, Theorem 18 guarantees that the final program is left-terminating. Notice that checking this is trivial: One has to check that the atom shifted to the left is a \textit{base} atom with \(P_{\text{base}}\) always-left-terminating. This can be trivially guaranteed by the fact that \(P_{\text{base}}\) is not recursively defined or that it contains only built-ins.

One might wonder why in the definition of initial program we do not allow \textit{base} atoms to be labeled \textit{"f"}. Our condition \textit{SW1} for allowed switch always admits to move \textit{base} atoms to the left. As a consequence, if \textit{base} atoms would be labelled, condition \textit{F3} in fold definition would often be trivially satisfied, yielding to unsound applicability conditions. The following example shows it.

**Example 22.** Let \(P_0\) be the program

\[c_1: p \leftarrow q, r(X).
\[c_2: r(g(X)) \leftarrow r(X)^f, s^f.
\[c_3: s.
\]

where \(P_{\text{new}} = \{c_1\}, P_{\text{old}} = \{c_2\}, P_{\text{base}} = \{c_3\}\). Notice that \(P_0\) is left terminating and \(P_{\text{base}}\) is always terminating. Here we have labeled \textit{"f"} the \textit{base} atom \(s\) in the second clause. By unfolding \(r(X)\) in \(c_1\) we obtain the following clause

\[c_4: p \leftarrow q, r(Y)^f, s^f.\]

Now, since \(s\) is a \textit{base} atom, by condition \textit{SW1}, we are allowed to move it leftward. By doing this twice, we obtain the clause

\[c_5: p \leftarrow s^f, q, r(Y)^f.\]

Now, the presence of the labeled atom \(s^f\) allows us - by condition \textit{F3} - to fold \(q, r(Y)^f\) with \(c_1\) in \(c_5\), the resulting program is \(\{c_2, c_3\}\) plus the clause

\[c_6: p \leftarrow s^f, p.\]

which is obviously not left terminating. \(\square\)
5 Transformation of Well-Moded Programs

In this section we introduce an extension of Theorem 18. By referring to moded programs, we can partially lift the requirement that \( P_{\text{base}} \) has to be always-terminating. In this setting, it is sufficient that it is well-terminating, i.e. that it terminates for every well-moded query. This enlarges considerably the application of our techniques: Most well-moded programs are in fact well-terminating; in particular a well-terminating programs might well be recursive, which is usually not the case for always-terminating ones.

**Theorem 23 (Main for Well-Moded Programs).** Let \( P_0 \) be a well-moded definite program and \( P_0, \ldots, P_n \) a transformation sequence of well-moded programs. If \( P_{\text{base}} \) is well-terminating, \( P_0 \) is left terminating and every switch operation performed in \( P_0, \ldots, P_n \) satisfies condition \( SW1 \), then \( P_n \) is left terminating.

The proof is similar to the one of Theorem 18, but we need to apply the following modularity result.

**Proposition 24.** Let \( P \) and \( Q \) be two well-moded programs such that \( P \) extends \( Q \). Let \( M' \) be a model of \( P \cup Q \). Suppose that

(i) \( Q \) is well-terminating,
(ii) \( P \) is acceptable wrt \( M \) and a level mapping \( ||P|| \).

Then \( P \cup Q \) is left terminating.

Due to the lack of space the proof is omitted. The intuition is the following. Since a ground query is well-moded and LD-resolution preserves well-moding, see Lemma 7, then any LD-resolvent of a ground query is also well-moded. As a consequence, when the leftmost atom in an LD-resolvent is in \( Q \), it is well-moded as well and then terminating.

**Example 25.** Let us consider the following simple program which counts the elements in a list of natural numbers and computes their sum.

\[
d: \quad p(L_s, S, N) \leftarrow \text{sumList}(L_s, S), \text{count}(L_s, N). \\
c1: \quad \text{sumList}([], 0). \\
c2: \quad \text{sumList}(H \cdot T_s, S) \leftarrow \text{sumList}(T_s, S'), \text{sum}(S', H, S). \\
c3: \quad \text{sum}(0, Y, Y). \\
c4: \quad \text{sum}(s(X), Y, s(Z)) \leftarrow \text{sum}(X, Y, Z). \\
c5: \quad \text{count}([], 0). \\
c6: \quad \text{count}(H \cdot T_s, s(N)) \leftarrow \text{count}(T_s, N).
\]

Let us consider the following modes: \( p(+, -, -), \text{sumList}(+, -), \text{sum}(+, +, -) \) and \( \text{count}(+,-) \). \( p \) is the only new predicate and \( \text{count} \) and \( \text{sum} \) are base predicates. Note that \( P_0 \) is well-moded and \( P_{\text{base}} \) is well-terminating for the given modes. 

\( p \) scans the list three times; this can be fixed via an unfold/fold transformation sequence. As usual, our first step consists in some unfold operations. Let us unfold all the atoms in the body of \( d \)
d1: \( p(\emptyset, 0, 0) \).

d2: \( p([H|Ts], S, s(N')) \leftarrow \text{sumList}(Ts, S)^f, \)
\( \text{sum}(S', H, S), \text{count}(Ts, N') \).

Now we need to move count \((Ts, N')\) to the left of \(\text{sum}(S', H, S)\). This is allowed by SW1 since count \((Ts, N')\) is in \(P_{\text{base}}\). After the switch operation we obtain the following clause:

\[
d3: \ p([H|Ts], S, s(N')) \leftarrow \text{sumList}(Ts, S)^f, \\
\text{count}(Ts, N'), \text{sum}(S', H, S).
\]

We can now apply the fold operation, and fold sumList \((Ts, S)^f\), count \((Ts, N')\) with definition d in the body of d3. After this operation the final program is given by \(\{c1, \ldots, c6\}\) plus

\[
d1: \ p(\emptyset, 0, 0).

d4: \ p([H|Ts], S, s(N')) \leftarrow p(Ts, S', N'), \text{sum}(S', H, S).
\]

Now \(p\) has a recursive definition which scans the list only once. \(\square\)

### 6 On the Applicability of condition SW1

In this Section we intend to discuss the applicability of condition \(\text{SW1}\) for switching and to relate it to our previous proposal in [BCE93].

In our experience, in presence of modes, Theorem 23 allows one to perform almost any switch. The requirement that \(P_{\text{base}}\) be well-terminating is not a real restriction: As we already mentioned, in presence of modes most terminating programs are well-terminating as well. Thus in practice there remain two main situations in which Theorem 23 is not applicable: The first one is when the switching spoils the well-modedness of the clause. This is a very reasonable limitation: Since we apply our transformation sequence to well-modeled programs, we want to preserve the well-modedness property and this clearly restricts the applicability of transformation operations. The second case is when the atom we want to shift leftward cannot be a \(\text{base}\) atom because it is either defined in that part of the program which is being modified by the transformation, or because we need to label it in order to apply the folding operation (condition (F3)). In this case we can usually apply another condition for switching, namely the one we already defined in [BCE93]. We briefly recall here such condition.

Given a moded atom \(A\), we denote by \(\text{In}(A)\) and \(\text{Out}(A)\) the sequence of terms filling in, respectively, the input and the output positions of \(A\). Moreover we denote by \(\text{VarIn}(A)\) (resp. \(\text{VarOut}(A)\)) the set of variables occurring in the input (resp. output) positions of \(A\). A similar notation is used for sequences of atoms.

**Definition 26 (Non-Failing Atom).** Let \(P\) be a moded definite program, \(M_P\) its least Herbrand model and \(cl : H \leftarrow J, A, K\) be a clause in \(P\). We say that \(A\) is non-failing in \(cl\) if for each grounding \(\theta\), such that \(\text{Dom}(\theta) = \text{Var}(\text{In}(H), J, \text{In}(A))\) and \(M_P \models J\theta\), there exists \(\gamma\) such that \(M_P \models A\theta\gamma\). \(\square\)
The reason why we called such an atom “non-failing” is the following: Suppose that cl is used in the resolution process, and that the unification only binds the variables in the input positions of H, then, if A will eventually be selected by the leftmost selection rule, the computation of the subgoal A will eventually succeed.

**Definition 27 (Allowed Switching, (Condition SW2)).** Let cl : H ← J, A, B, K. be a clause of a moded program P, and let P’ be the result of switching A with B in cl. The switch is allowed if the following three conditions are satisfied

- A is an old atom,
- \( \text{VarOut}(A) \cap \text{VarIn}(B) = \emptyset \), and
- A is non-failing in cl.

Requiring that \( \text{VarOut}(A) \cap \text{VarIn}(B) = \emptyset \) ensures that the input of B does not depend on the output of A, and this is a natural requirement when transforming moded programs. On the other hand, the requirement that A is non-failing in cl intuitively forbids the possibility that left termination holds by failure of A, even if B is non-terminating; in such a case, moving A rightward would result in the introduction of a loop.

We give now an example where it is necessary to apply condition SW2 for switching two atoms.

**Example 28.** Let \( P_0 \) be the program

\begin{align*}
\text{c1: } & \text{diff}(\text{Ls}, \text{I1}, \text{I2}, \text{N}) \leftarrow \\
& \text{count}(\text{Ls}, \text{I1}, \text{N}^1), \text{count}(\text{Ls}, \text{I2}, \text{N}^2)^1, (\text{N is N}^1-N^2). \\
\text{c2: } & \text{count}([], \text{I}, 0). \\
\text{c3: } & \text{count}([\text{H} | \text{Ts}], \text{I}, \text{M}) \leftarrow \\
& \text{member}(	ext{H}, \text{I}, \text{yes}), \text{count}([\text{Ts}, \text{I}, \text{M}]^1), (\text{M is M}^1+1). \\
\text{c4: } & \text{count}([\text{H} | \text{Ts}], \text{I}, \text{M}) \leftarrow \text{member}(	ext{H}, \text{I}, \text{no}), \text{count}([\text{Ts}, \text{I}, \text{M}]^1). \\
\text{c5: } & \text{member}(	ext{X}, [], \text{no}). \\
\text{c6: } & \text{member}(	ext{X}, [\text{Xi} | \text{Xs}], \text{yes}). \\
\text{c7: } & \text{member}(	ext{X}, [\text{Y} | \text{Xs}], \text{R}) \leftarrow X \neq Y, \text{member}(	ext{X}, \text{Xs}, \text{R}). \\
\text{d: } & \text{p}(\text{Ls}, \text{I1}, \text{I2}, \text{N}, \text{N}^2) \leftarrow \\
& \text{count}(\text{Ls}, \text{I1}, \text{N}^1), \text{count}(\text{Ls}, \text{I2}, \text{N}^2). \\
\end{align*}

Together with the following modes: diff(+,+,-), count(+,+,-), member(+,+,-), is(-,+), \( \neq (+,+,-) \) and \( p(+,+,-,,-) \).

p is the only new predicate. member, \( \neq \) and is are base predicates. member is well-terminating, \( \neq \) and is are built-ins, hence trivially well-terminating. \( P_0 \) is well-moded.

The predicate p is meant to improve the efficiency of predicate diff. Given a list \( \text{Ls} \) and two check-lists \( \text{I1} \) and \( \text{I2} \), p determines how many elements of \( \text{Ls} \) are in each check-list. In order to scan \( \text{Ls} \) only once, we apply the transformation sequence of well-moded programs.

Let us unfold both the atoms in the body of d...
\[ d_1: \ p([\emptyset, I_1, I_2, 0, 0]). \]
\[ d_2: \ p([H|Ts], I_1, I_2, N_1, N_2) \leftarrow \\
\quad \text{member}(H, I_1, \text{yes}), \ \text{count}(Ts, I_1, N_1'), \ (N_1 \text{ is } N_1'+1), \\
\quad \text{member}(H, I_2, \text{yes}), \ \text{count}(Ts, I_2, N_2'), \ (N_2 \text{ is } N_2'+1). \]
\[ d_3: \ p([H|Ts], I_1, I_2, N_1, N_2) \leftarrow \\
\quad \text{member}(H, I_1, \text{no}), \ \text{count}(Ts, I_1, N_1'), \\
\quad \text{member}(H, I_2, \text{yes}), \ \text{count}(Ts, I_2, N_2'), \ (N_1 \text{ is } N_1'+1). \]
\[ d_4: \ p([H|Ts], I_1, I_2, N_1, N_2) \leftarrow \\
\quad \text{member}(H, I_1, \text{yes}), \ \text{count}(Ts, I_1, N_1'), \\
\quad \text{member}(H, I_2, \text{no}), \ \text{count}(Ts, I_2, N_2'). \]
\[ d_5: \ p([H|Ts], I_1, I_2, N_1, N_2) \leftarrow \\
\quad \text{member}(H, I_1, \text{no}), \ \text{count}(Ts, I_1, N_1'), \\
\quad \text{member}(H, I_2, \text{no}), \ \text{count}(Ts, I_2, N_2'). \]

In order to fold back with definition \( d \) we need to reorder the atoms in all the clauses \{\( d_2, \ldots, d_5 \)\}. First we switch the third and the fourth atoms and then the fourth and the fifth ones in clauses \( d_2, d_4 \). The first switch is allowed for the condition \( \text{SW1} \) since the fourth atom, \text{member}, is a \textit{base} one. The second switch can be allowed only by condition \( \text{SW2} \). In fact the fifth atom, \text{count}, is not a \textit{base} atom. Such atom cannot be in \( P_{\text{base}} \), otherwise the final folding with \( d \) would be forbidden because condition \( \text{(F3)} \) in fold definition cannot be satisfied.

Hence we have to verify condition \( \text{SW2} \), namely to prove that \( (N_1 \text{ is } N_1'+1) \) is non-failing in \( d_2 \).

These switches move the \( is \) atom two positions to the right
\[ d_1: \ p([\emptyset, I_1, I_2, 0, 0]). \]
\[ d_6: \ p([H|Ts], I_1, I_2, N_1, N_2) \leftarrow \\
\quad \text{member}(H, I_1, \text{yes}), \ \text{count}(Ts, I_1, N_1'), \\
\quad \text{member}(H, I_2, \text{yes}), \ \text{count}(Ts, I_2, N_2'), \ (N_1 \text{ is } N_1'+1), \ (N_2 \text{ is } N_2'+1). \]
\[ d_3: \ p([H|Ts], I_1, I_2, N_1, N_2) \leftarrow \\
\quad \text{member}(H, I_1, \text{no}), \ \text{count}(Ts, I_1, N_1'), \\
\quad \text{member}(H, I_2, \text{yes}), \ \text{count}(Ts, I_2, N_2'), \ (N_2 \text{ is } N_2'+1). \]
\[ d_7: \ p([H|Ts], I_1, I_2, N_1, N_2) \leftarrow \\
\quad \text{member}(H, I_1, \text{yes}), \ \text{count}(Ts, I_1, N_1'), \\
\quad \text{member}(H, I_2, \text{no}), \ \text{count}(Ts, I_2, N_2'), \ (N_1 \text{ is } N_1'+1). \]
\[ d_5: \ p([H|Ts], I_1, I_2, N_1, N_2) \leftarrow \\
\quad \text{member}(H, I_1, \text{no}), \ \text{count}(Ts, I_1, N_1'), \\
\quad \text{member}(H, I_2, \text{no}), \ \text{count}(Ts, I_2, N_2'). \]

Now we switch the second and the third atom in all the clauses \{\( d_6, d_3, d_7, d_5 \)\}. This is allowed by condition \( \text{SW1} \) since \text{member} is a \textit{base} atom.
\[ d_1: \ p([\emptyset, I_1, I_2, 0, 0]). \]
\[ d_2: \ p([H|Ts], I_1, I_2, N_1, N_2) \leftarrow \\
\quad \text{member}(H, I_1, \text{yes}), \ \text{member}(H, I_2, \text{yes}), \\
\quad \text{count}(Ts, I_1, N_1'), \ \text{count}(Ts, I_2, N_2'). \]
\[(N1 \text{ is } N1' + 1), \ (N2 \text{ is } N2' + 1).\]

e3: \[p([H|Ts], I1, I2, N1, N2) \leftarrow\]
   member(H, I1, no), member(H, I2, yes),
   count(Ts, I1, N1)\text{f}, count(Ts, I2, N2')\text{f}, \ (N2 \text{ is } N2' + 1).

e4: \[p([H|Ts], I1, I2, N1, N2) \leftarrow\]
   member(H, I1, yes), member(H, I2, no),
   count(Ts, I1, N1')\text{f}, count(Ts, I2, N2)\text{f}, \ (N1 \text{ is } N1' + 1).

e5: \[p([H|Ts], I1, I2, N1, N2) \leftarrow\]
   member(H, I1, no), member(H, I2, no),
   count(Ts, I1, N1)\text{f}, count(Ts, I2, N2)\text{f}.

Now fold is possible and we obtain the final program which is given by \{c5, c6, c7\} plus

\[f1: \ \text{diff}(Ls, I1, I2, N) \leftarrow\]
   \[p(Ls, I1, I2, N1, N2), \ (N \text{ is } N1 - N2).\]

d1: \[p([], I1, I2, 0, 0).\]

f2: \[p([H|Ts], I1, I2, N1, N2) \leftarrow\]
   member(H, I1, yes), member(H, I2, yes),
   \[p(Ts, I1, I2, N1', N2'), \ (N1 \text{ is } N1' + 1), \ (N2 \text{ is } N2' + 1).\]

f3: \[p([H|Ts], I1, I2, N1, N2) \leftarrow\]
   member(H, I1, no), member(H, I2, yes),
   \[p(Ts, I1, I2, N1, N2'), \ (N2 \text{ is } N2' + 1).\]

f4: \[p([H|Ts], I1, I2, N1, N2) \leftarrow\]
   member(H, I1, yes), member(H, I2, no),
   \[p(Ts, I1, I2, N1', N2), \ (N1 \text{ is } N1' + 1).\]

f5: \[p([H|Ts], I1, I2, N1, N2) \leftarrow\]
   member(H, I1, no), member(H, I2, no),
   \[p(Ts, I1, I2, N1, N2).\]

\{c1, \ldots, c4\} are no more in \[P|\text{diff}.\]

The proof of Theorem 23 can be extended to include also switches allowed by condition \textbf{SW2}. In fact this switch is compatible with our transformation system and it allows one to move an atom to the right when it is an \textit{old non-failing} atom, whose output cannot influence the input of the atom to its right This applicability condition, differently from \textbf{SW1}, is not trivial at all to verify. In fact being \textit{non-failing} is a semantic requirement and in general not computable. Nevertheless, if we have more information on the initial program \(P_0\) and on the transformation sequence, such verification becomes feasible. In fact in [BC98] we have defined the class of \textit{noFD programs and queries} which cannot have finitely failing \textit{LD}-derivations and in [BC99] we have defined a larger class of \textit{successful programs} which, for terminating \textit{noFD} queries, are guaranteed to have at least one successful derivation. Often the \textit{noFD} and the successful properties for terminating programs are easy to verify and they are sufficient to guarantee the \textit{non-failing} property.
7 Conclusions and Related Work

This paper extends our previous work in [BCE95] and proposes an unfold-fold transformation system able to preserve left termination besides declarative semantics. The system introduces a new applicability condition for the switching operation. This condition is simple to verify, yet common in practice. We have also considered separately the situation in which the transformed program is well-mode. For this case the applicability condition of the switching operation are particularly mild. This proposal can also integrated with our previous one for switching, given in [BCE95], for solving the few cases in which SWI is not applicable.

Summarizing, in order to preserve left termination, we have modified Tamaki-Sato’s system by

- adopting a definition of unfold and fold which takes into account the order of the atoms in the bodies of the clauses;
- introducing the possibility of reordering – under appropriate applicability conditions – atoms in the bodies by allowed switches;
- adopting a labelling rule and a new applicability condition for the folding operation which is slightly more restrictive than the one of [TS84].

Our system is based on a modularization of the program which distinguishes the part which can be transformed and guides the transformation.

Other approaches to preserving termination properties, while transforming a program, can be found in [PP91,CG94,BE94,BC94,BC97].

The work of Proietti and Pettorossi in [PP91] made an important step forward in the direction of the preservation of left termination. They proposed a transformation system which is more restrictive than the ordered version of [TS84] since only unfolding the leftmost atom or a deterministic atom is allowed. They proved that such a system preserves the “sequence of answer substitution semantics” (a semantics for Prolog programs, defined in [JM84,Bau89]). This guarantees also that if the initial program is left terminating, then the resulting program is left terminating as well. They do not allow any reordering of atoms.

In [BE94] we proved that Tamaki-Sato’s transformation system preserves the property of being acyclic [AB90]. This has to do with the preservation of termination, in fact a definite program $P$ is acyclic if and only if all its derivations starting in a ground goal are finite whichever is the selection rule employed. Moreover, the tools used in [BE94] are quite similar to the ones used here. Unfortunately, as pointed out in [AP93], the class of acyclic programs is quite restrictive, and there are many natural programs which are left terminating but not acyclic.

The preservation of universal termination of a query with LD-resolution was also studied in [BC94], where we defined an appropriate operational semantics and split the equivalence condition to be satisfied into two complementary conditions: a “completeness” condition and the condition of being “non-increasing”. The validity of this second condition, which is very operational, ensures us that
a transformation cannot introduce infinite derivations. Again, however, the allowed transformations are seriously restricted by the impossibility of reordering atoms in the bodies. Hence in [BC97] the transformation system was extended by introducing a replacement transformation operation, a very powerful operation which includes switch as a particular case. The major problem in this proposal is how to verify in practice the applicability conditions necessary for preserving universal termination which are semantic conditions and operational in style.

More difficult is a comparison with [CG94] since they define a transformation system based only on unfold and replacement operations. In [CG94] the preservation of termination is considered but no condition for it is given and the verification is “a posteriori”.

Acknowledgements This work was supported partly by Italian MURST with the National Research Project 9701248444-004 on “Tecniche formali per la specifica, l’analisi, la verifica, la sintesi e la trasformazione di sistemi software”.

References


