Transformation of Left Terminating Programs: the Reordering Problem

Annalisa Bossi[1], Nicoletta Cocco[2] and Sandro Etalle[3]

[1] Dipartimento di Matematica, Università della Calabria,
Arcavacata di Rende (Cosenza) Italy.
[2] Dipartimento di Matematica Applicata e Informatica,
Universita' di Venezia - Ca' Foscari, via Torino, 155, 30173 - Mestre-Venezia
[3] D.I.S.I - Università di Genova,
Via Dodecanneso, 35, Genova

bossi@ccusc1.unical.it, cocco@moo.dsi.unive.it, sandro@disi.unige.it

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. When dealing with logic programs such as PROLOG programs, one is particularly interested in preserving *left termination* i.e. termination wrt the leftmost selection rule, which is by far the most widely employed of the search rules. Unfortunately, the most popular Unfold/Fold transformation systems ([TS84, Sek91]) do not preserve the above termination property. In this paper we study the reasons why left termination may be spoiled by the application of a transformation operation and we present a transformation system based on the operations of Unfold, Fold and Switch which – if applied to a left terminating programs – yields a program which is left terminating as well.

1 Introduction

As shown by a number of applications, program transformation is a valuable methodology for the development and optimization of large programs. In this field, the unfold/fold transformation rules were first introduced by Burstall and Darlington [BD77] for transforming clear, simple functional programs into equivalent, more efficient ones. Then, such rules were adapted to logic programs both for program synthesis [CS77, Hog81], and for program specialization and optimization [Kom82].

Soon later, Tamaki and Sato [TS84] proposed an elegant framework for the transformation of logic programs based on unfold/fold rules. Their system was proven to be correct w.r.t. the least Herbrand model semantics [TS84] and the computed answer substitution semantics [KK88].

Tamaki-Sato's system became quite soon the main reference point in the literature of unfold/fold transformations of logic programs.

However, Tamaki-Sato's method cannot be applied "as it is" to most of the actual logic programs (like pure PROLOG programs) because it does not preserve left termination (here we say that a program is left terminating if all its derivations starting in a ground goal and using PROLOG's fixed "leftmost" selection rule are

finite). So it can happen that a left terminating program is transformed into a non left terminating one. This is of course a situation that we need to avoid.

This problem has already been tackled in [PP91, BC94] [4], but none of these proposals includes any operation that could be employed in order to *rearrange* the atoms in the bodies of a clause. This is actually a serious limitation, in fact such an operation is often needed in order to be able to perform a subsequent fold operation. Indeed, in the majority of the examples we find in the literature, the fold operation is only possible after a rearrangement one.

In this paper we propose a new transformation system, in which we explicitly consider a *switch* operation, for which we provide specific (and needed) applicability conditions. We also provide new applicability conditions for the fold operation, and we prove that the system, when applied to a left terminating program always returns a programs which is left terminating as well.

We obtain our results by exploiting the properties of *acceptable programs*, as defined by Apt and Pedreschi in [AP90].

Section 2 contains the notation and the preliminaries on left terminating and acceptable programs. In section 3 we define the unfold/fold transformation system, and we state the main correctness results. In section 4 we will discuss and motivate the approach we have followed by showing further examples and by relating our transformation system with the ones of Tamaki and Sato [TS84] and of Seki [Sek91], as well as with other approaches to the problem of preserving termination.

The main proofs are reported in [BCE96].

2 Preliminaries

In what follows we study definite (i.e. negation-free) logic programs executed by means of the *LD-resolution* (which corresponds to the SLD resolution combined with the fixed PROLOG selection rule).

We adopt the "bold" notation (es. $\mathbf{B}$) to indicate (ordered) sequences of objects, typically $\mathbf{B}$ indicates a sequence of atoms, $B_1, \ldots, B_n$, $\mathbf{t}$ is a sequence of terms, $t_1, \ldots, t_n$, and $\mathbf{x}$ is a sequence of variables, $x_1, \ldots, x_n$.

We work with *queries* $\mathbf{Q}$, that is sequences of atoms, $B_1, \ldots, B_n$, instead of *goals*. Apart from this, we use the standard notation of Lloyd [Llo87] and Apt [Apt90]. 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 $\{x_1, \ldots, x_n\}$, by $Range(\theta)$ the set of terms $\{t_1, \ldots, t_n\}$, and by $Ran(\theta)$ the set of variables appearing in $\{t_1, \ldots, t_n\}$. Finally, we define $Var(\theta) = Dom(\theta) \cup Ran(\theta)$.

Recall that a substitution θ 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 θ and a set (sequence) of variables $\mathbf{v}$, we denote by $\theta_{|\mathbf{v}}$ the substitution obtained from θ by restricting its domain to $\mathbf{v}$.

[4] Another related paper is [BE94] where termination with respect to any selection rule is considered.

2.1 Left Termination and Acceptable Programs

We begin with the key definition.

Definition 1. A program P is called *left terminating* if all LD-derivations of P starting in a ground query are finite. □

Acceptable programs were introduced by Apt and Pedreschi in [AP90] in order to characterize the class of left terminating definite logic programs. Their results were successively extended to cover also general logic programs [AP93].

Given an interpretation I of a program P, and given a sequence of ground atoms $\mathbf{B} = B_1, \ldots, B_n$, we say that B_j is *reachable* (under I) if $I \models B_1, \ldots, B_{j-1}$. In fact, if I is the least model and we look at $B_1, \ldots, B_n$ as a query to be evaluated with the leftmost selection rule, the fact that $I \models B_1, \ldots, B_{j-1}$ implies that $B_1, \ldots, B_{j-1}$ will eventually be resolved by the LD-resolution process, and hence that, the (leftmost) selection rule will eventually reach (i.e. select) B_j.

Definition 2. Let $\mathbf{B} = B_1, \ldots, B_n$ be a sequence of ground atoms. Let $|\ |$ be a level mapping (i.e. a function mapping the Herbrand base into natural numbers), and I be an interpretation. Moreover, let $B_1, \ldots, B_k$ be the set of reachable atoms (under I) of $\mathbf{B}$. Then we define

$$|\mathbf{B}|_I = \sup(|B_1|, \ldots, |B_k|),$$

In other words, $|\mathbf{B}|_I$ is equal to the greatest of the level mappings of the reachable atoms of $\mathbf{B}$. □

Notice that, for any single atom B, $|B|_I = |B|$, whatever the interpretation I and the level mapping are.

Definition 3 (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 $H \leftarrow \mathbf{B}$ of it,

$$|H| > |\mathbf{B}|_I$$

  In other words, for every ground instance $H \leftarrow B_1, \ldots, B_m$, and for every reachable B_i, $|H| > |B_i|$,
- P is *acceptable with respect to* $|\ |$ and I iff all its clauses are. P is called *acceptable* if it is acceptable with respect to some level mapping and interpretation of P. □

We can now fully motivate the use of acceptable programs.

Theorem 4. [AP93] A program is left terminating iff it is acceptable. □

2.2 Input positions

Modes are extensively used in the literature on Logic Programs (see for instance [AM94], usually they indicate how the arguments of a relation should be used. When dealing with termination properties mode information are particularly interesting. In fact we may relate the length of LD-derivations to some measure on input terms. This has been the basis of some approaches to proving universal termination of definite programs [Plü90, BCF94, DSF93]. In our transformation system we use just simpler information since we assume that each n-ary relation symbol p has a set of input positions $In(p) \subseteq \{1, \ldots, n\}$ associated to. For an atom A we denote by $In(A)$ the family of terms filling in the input positions of A and by $VarIn(A)$ the set of variables occurring in the input positions of A. Similar notation is used for sequences of atoms. In this paper input positions are going to be used only to broaden the range of transformations we can prove to maintain the left termination of the initial program. Throughout the paper we make the following assumption:

Assumption 1 The level mapping of an atom is uniquely determined by the terms that are found in its input positions. □

Of course this assumption imposes no syntactic restriction of the program we are going to manipulate, as we can always assume that all the positions of every relation symbol are input positions.

3 A termination preserving unfold/fold transformation system

We can now introduce the transformation system. Here we use the concept of *labelled atom*, in particular we need to label with "f" (for fold-allowing) some atoms in the bodies of the clauses of the program.

We start from the requirements on the *initial* program, which are similar to the ones proposed in [TS84]. Standardization apart is always assumed.

Definition 5 (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) P_{new} is a non-recursive extension of P_{old}, namely all the predicates which are defined in P_{new} occur neither in P_{old} nor in the bodies of the clauses in P_{new};
(I3) all the atoms in the bodies of the clauses of P_{old} are labelled "f" and they are the only ones with such a label. □

The predicates defined in P_{new} are called *new* predicates, while those defined in P_{old} are the *old* predicates. The only difference between these conditions and the ones stated in [TS84] is the presence of **I3** which is due to the particular conditions we use for the fold operation.

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

Example 1. Let P_0 be the following program

```
c1:  path(X,[X]).
c2:  path(X,[X|Xs]) ← arc(X,Y)^f, path(Y,Xs)^f.

c3:  goodlist([]).
c4:  goodlist([X|Xs]) ← good(X)^f, goodlist(Xs)^f.

c5:  goodpath(X,Xs) ← path(X,Xs),goodlist(Xs).
```

Together with a database DB where the predicates arc and good are defined. The query goodpath(X,Xs) can be employed for finding a path Xs starting from the node X which contains exclusively "good" nodes. We consider the first position in all the relations as the input one. Notice that, under the assumption that the directed graph described by the relation arc is acyclic, the program is left terminating. Indeed, the ordering on the graph nodes induces a level mapping for the relation *arc* which can be used to prove left termination of the program. Clearly, we can choose all level mappings satisfying assumption 1.

As it is now, goodpath works on a "generate and test" basis: first it produces a whole path, and then it checks whether it contains only "good" nodes or not. Of course this strategy is quite naive: checking if the node is "good" or not *while* generating the path would noticeably increase the performances of the program. We can obtain such an improvement via an unfold/fold transformation. For this we split the program into $\mathtt{P_{old}} = \{\mathtt{c1}, \ldots, \mathtt{c4}\} \cup \mathtt{DB}$ and $\mathtt{P_{new}} = \{\mathtt{c5}\}$, thus goodpath is the only new predicate. This also explains the labelling used above. □

According to the usual transformation strategy, the first operation we apply is the unfold one. Unfold is the fundamental operation for program transformations and consists in applying a resolution step to the considered atom in all possible ways. Recall that the order of *the atoms in the queries and in the bodies of the clauses is relevant* since we are dealing with LD-resolution.

Definition 6 (unfold). Let $cl : H \leftarrow \mathbf{J}, A, \mathbf{K}.$ be a clause.
Let $\{A_1 \leftarrow \mathbf{B}_1., \ldots, A_n \leftarrow \mathbf{B}_n.\}$ be the set of clauses of P whose heads unify with A, by mgu's $\{\theta_1, \ldots, \theta_n\}$.

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

The unfold operation doesn't modify the labels of the atoms, no matter if the unfolded atom itself is labelled or not. Thus unfold allows to propagate the labels inside the clauses in the obvious way. This is best shown by the following example.

Example 1 (part 2) By unfolding the atom path(X,Xs) wrt P_0 in the body of c5, we obtain

```
c6:  goodpath(X,[X]) ← goodlist([X]).
c7:  goodpath(X,[X|Xs]) ← arc(X,Y)^f,
        path(Y,Xs)^f, goodlist([X|Xs]).
```

In the above clauses we can unfold goodlist([X]) and goodlist([X|Xs]) The resulting clauses, after a further unfolding of goodlist([]) wrt P_0 in the clause obtained by c6, are

```
c8:  goodpath(X,[X]) ← good(X)^f.
c9:  goodpath(X,[X|Xs]) ← arc(X,Y)^f,
        path(Y,Xs)^f, good(X)^f, goodlist(Xs)^f.
```

Let $P_1 = \{c1, \ldots, c4, c8, c9\} \cup DB$. □

Thanks to its correspondence to a resolution step, the unfold operation is safe wrt 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 and hence 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 `path(Y,Xs)` and `good(X)`. But, the rearrangement of the atoms in the body of a clause is a typical operation which does not preserve left termination. Moreover, as we'll discuss in section 4 in the context of an unfold/fold transformation system things are further complicated by the presence of the other operations. The approach we propose for guaranteeing left termination is based on the following definition.

Definition 7 (non-failing atom). Let P be a program, M_P its least Herbrand model and $cl : H \leftarrow \mathbf{J}, A, \mathbf{K}.$ be a clause of P. We say that

$$A \text{ is } \textit{non-failing} \text{ in } cl$$

iff for each grounding θ, such that $Dom(\theta) = Var(In(H), \mathbf{J}, In(A))$ and $M_P \models \mathbf{J}\theta$, there exists γ such that $M_P \models A\theta\gamma$. □

The reason why we call such an atom "non-failing" is the following: suppose that cl is used in the resolution process, and that the unification can bind only 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.

Note that, without restricting the domain of θ, the non-failing condition would be seldom satisfied. By restricting the set of substitutions, it becames feasible to verify it in many cases. However the condition is clearly not decidable in general.

We are now ready to introduce the switch operation.

Definition 8 (switch). Let $cl : H \leftarrow \mathbf{J}, A, B, \mathbf{K}.$ be a clause of a program P. *Switching* A with B in cl consists of replacing cl with $cl' : H \leftarrow \mathbf{J}, B, A, \mathbf{K}.$

We say that the switch is *allowed* if the following three conditions hold:

- A is an *old* atom,
- $VarIn(B) \subseteq VarIn(H) \cup Var(\mathbf{J})$,
- A is non-failing in cl. □

Requiring that $VarIn(B) \subseteq VarIn(H) \cup Var(\mathbf{J})$ ensures that the input of B does not depend (solely) 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 B leftward would result in the introduction of a loop.

Example 1 (part 3) By permuting `path(Y,Xs)` with `good(X)` we obtain the following clause:

```
c10: goodpath(X,[X|Xs]) ← arc(X,Y)^f,
        good(X)^f, path(Y,Xs)^f, goodlist(Xs)^f.
```

Let $P_2 = \{c1, \ldots, c4, c8, c10\} \cup DB$. Notice that this operation is *allowed*, whatever the model of the program we are referring to: in fact if we take N to be the least Herbrand model of P_2 then we have that for any substitution $\theta = \{Y/t\}$ there exists a substitution γ (namely, $\gamma = \{Xs/[t]\}$) such that $N \models path(Y, Xs)\theta\gamma$. So for any θ, $Dom(\theta) = \{X, Y\}$ there exists a substitution γ such that $N \models path(Y, Xs)\theta\gamma$. By Herbrand's theorem, this holds for any other model M, and therefore `path(Y,Xs)` is non-failing (wrt any model M) in `c9`. □

The switch is the simplest form of reordering operation. Clearly, any permutation of the atoms in a clause body can be obtained by a finite composition of switches of adjacent atoms.

When we apply a switch operation, we *are allowed to exchange the labels of A and B* (we don't have to, though, but such an exchange may come in handy for the application of a subsequent fold operation).

Fold is the inverse of unfold when one single unfold is possible. It consists in substituting an atom A for an equivalent conjunction of literals $\mathbf{K}$ 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. As in Tamaki and Sato [TS84], the transformation sequence and the fold operation are defined in terms of each other.

Definition 9 (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 wrt P_i, a switch, or a fold operation to a clause of P_i. □

Definition 10 (fold). Let $P_0, \ldots, P_i$, $i \geq 0$, be a transformation sequence, $cl : H \leftarrow \mathbf{J}, \mathbf{B}, \mathbf{K}.$ be a clause in P_i, and $d : D \leftarrow \mathbf{B}'.$ be a clause in P_{new}. *Folding* $\mathbf{B}$ *in* cl *via* τ consists of replacing cl by $cl' : H \leftarrow \mathbf{J}, D\tau, \mathbf{K}$, provided that τ is a substitution such that $Dom(\tau) = Var(d)$ and such that the following conditions hold:

(F1) d is the only clause in P_{new} whose head is unifiable with $D\tau$;
(F2) If we unfold $D\tau$ in cl' wrt P_{new}, then the result of the operation is a variant of cl;
(F3) one of the atoms in $\mathbf{J}$, or the leftmost of the atoms in $\mathbf{B}$ is labelled "f". □

Notice that the fold clearly eliminates the labels in the folded part of the body. The conditions **F1** and **F2** may appear new, but it is not difficult to prove that their combination corresponds to the combination of the conditions **F1**...**F3** of [TS84]. This is shown by the following remark 11. 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 condition **F3**. The comparison of our definition with the one given by Seki [Sek91] for preserving Finite

Failures is not straightforward. In fact, in [Sek91] $SLDNF-resolution$ is considered, hence, no reordering is needed. But, all the atoms in $\mathbf{B}$ have to be labelled. On the other hand, we require a much weaker condition on labelling but, since we consider $LD-resolution$, we have to take into account the order of atoms.

Remark 11. The following observations are in order:

(a) Condition **F1** can be restated as follows: "d is the only clause of P_{new} that can be used to unfold $D\tau$ in cl'".

(b) If we let $\mathbf{v} = Var(\mathbf{B}') \setminus Var(D)$ be the set of local variables of d, then condition **F2** can also be expressed as follows:

(F2a) $\mathbf{B}'\tau = \mathbf{B}$;
(F2b) For any $x, y \in \mathbf{v}$
- $x\tau$ is a variable;
- $x\tau$ does not appear in cl';
- if $x \not\equiv y$ then $x\tau \not\equiv y\tau$;

The equivalence between **F2** and the combination of **F2a** and **F2b** follows (indirectly, though), from Theorems 6.3 and the discussion after definition 4.9 in [EG94]. These latter conditions are the "standard" ones for folding (they are present, for instance, in Seki's [Sek91, Sek93]). □

Example 1 (part 4) We can now fold `path(Y,Xs)`, `goodlist(Xs)` in `c10`. The resulting clause is

`c11: goodpath(X,[X|Xs])` $\leftarrow$ `arc(X,Y)`^{f}, `good(X)`^{f}, `goodpath(Y,Xs)`.

Let $\mathtt{P}_2 = \{\mathtt{c1}, \ldots, \mathtt{c4}, \mathtt{c8}, \mathtt{c11}\} \cup \mathtt{DB}$. Notice that because of this operation the definition of `goodpath` is now recursive and it checks the "goodness" of the path while generating the path itself. □

3.1 Correctness result

We can now state the main properties of the transformation system.

Theorem 12 (main). Let $P_0, \ldots, P_n$ be a transformation sequence. Suppose that P_0 is acceptable and every switch operation performed in $P_0, \ldots, P_n$ is allowed, then

- P_n is acceptable,

In particular,

- P_n is left terminating.

Proof. See [BCE96]. □

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

Remark 13. 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. □

Finally, since our system guarantees the preservation of left termination, and since for left termination programs the Finite Failure set coincides with the complement of the least Herbrand model, we also have the following.

Corollary 14 (preservation of finite failure). Let $P_0, \ldots, P_n$ be a transformation sequence. If P_0 is acceptable and every switch operation performed in $P_0, \ldots, P_n$ is allowed, then the finite failure set of P_n coincides with the one of P_0. □

4 Discussion and related work

Tamaki-Sato's system [TS84] was devised for definite logic program, and it is correct wrt (among others) the least Herbrand model and the computed answer substitution semantics. Later Seki proposed a new version with more restrictive applicability conditions (with the so-called *modified* fold operation) and proved that it maintains the Finite Failure set of the initial program [Sek91].

Concerning the termination issue, neither Tamaki-Sato's nor Seki's method are devised to preserve left termination of the initial program. Indeed they don't. Moreover, their definition of the (fold and unfold) operations are given *modulo reordering of the atoms in the bodies of the clauses*, which clearly does not preserve left termination. On the other hand, even the *ordered* version [5] of Tamaki-Sato's method does not preserve left termination. This is shown by the following example.

Example 2. Let $\mathtt{P_0}$ be the following program

```
c1:  p ← q, h(X).
c2:  h(s(X)) ← h(X)^f.
```

Where $\mathtt{P_{new}} = \{\mathtt{c_1}\}$ and $\mathtt{P_{old}} = \{\mathtt{c_2}\}$. Notice that there is no definition for predicate `q`, so everything fails (finitely). Notice also that the program is left terminating. By unfolding `h(X)` in `c1` we obtain a variant of `c1`, but with a different labelling :

```
c3:  p ← q, h(Y)^f.
```

Now, following the Tamaki-Sato approach, we can fold `q, h(Y)` in `c3`, using clause `c1` for folding. The result is

```
c4:  p ← p
```

[5] Here by *ordered version of Tamaki-Sato's method* we mean a transformation system just like Tamaki-Sato's but in which (a) no reordering is allowed in the bodies of the clauses, and (b) the definition of unfold and fold are restated (in the obvious way) so to take into consideration the order of the atoms in the bodies of the clauses. Consequently the *ordered* version of Tamaki-Sato's method has applicability conditions which are quite more restrictive that the original ones.

Now the program is clearly not left terminating any longer. □

This shows that for preserving left termination we need in any case a system more restrictive than the one of [TS84].

By adopting a definition of unfold and fold which takes into consideration the order of the atoms in the bodies of the clauses and by appropriately restricting the fold operation we can preserve left termination. This has been shown also in [PP91, BC94]. Unfortunately, the systems thus obtained have a serious limitation for practical applicability. Indeed, most of the examples found in the literature on unfold/fold transformations require at some point a reordering operation (this obviously applies to example 2 which is reported in the previous section); it is usually needed in order to be able to perform a subsequent fold operation.

In order to partially recover the power of the Tamaki and Sato's system while preserving termination, we introduced a switch operation. Such an operation presents subtle aspects which have to be taken into consideration. This is best shown by the following example.

Example 3. Let P_0 be the following program.

c1: z ← p, r.
c2: p ← q^f, r^f.
c3: q ← r^f, p^f.

Where $P_{new} = \{c1\}$ and $P_{old} = \{c2, c3\}$. Notice that r is not defined anywhere, so everything fails; notice also that this program is left terminating. By unfolding p in c1 we obtain the following clause:

c4: z ← q^f, r^f, r.

By further unfolding q in c4 we obtain:

c5: z ← r^f, p^f, r^f, r.

Now we permute the first two atoms, which are both labelled, obtaining:

c6: z ← p^f, r^f, r^f, 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: z ← z, r^f, r.

which is obviously non left terminating. □

In the above example we have employed the ordered version of Seki's definition of fold, i.e. the most restrictive definition of fold we could consider, surely more restrictive than the one we propose here. Nevertheless we have a situation in which the switch operation does preserve left termination in a local way while left termination will subsequently be destroyed by the application of the fold operation. This shows that the switch operation requires applicability conditions which do not just guarantee the termination properties of the actual program.

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

- adopting a definition of unfold and fold which takes into consideration the order of the atoms in the bodies of the clauses;
- adopting a labelling rule and a restriction on folding which guarantees to preserve left termination;
- introducing the possibility to reorder atoms in the bodies by allowed switches.

Other approaches to preserving termination properties can be found in [PP91, CG94, BE94, BC94].

The work of Proietti and Pettorossi [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 left most 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 [N.J84, 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 the *LD-resolution* was also studied in [BC94]. In order to capture c.a.s. and universal termination, in that paper we defined an appropriate operational semantics and splitted the equivalence condition to be satisfied into two complementary conditions: the completeness condition and the condition of being non-increasing. The validity of this second condition, which is very operational, ensures us that the transformation cannot introduce infinite derivations. We proved that, by restricting the Tamaki-Sato's original system, the whole transformation sequence is non-increasing and then it preserves also universal termination. As a consequence, acceptability of programs is also preserved by such a restricted transformation sequence. Again, however, the allowed transformations are seriously restricted by the impossibility of reordering atoms in the bodies.

More difficult is a comparison with [CG94] since they define a transformation system based only on unfold and replacement operations. The replacement operation is very powerful and it includes both fold and switch as particular cases. In [CG94]the preservation of termination is considered but the verification is "a posteriori".

References

[AB90] K. R. Apt and M. Bezem. Acyclic programs. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 617–633. The MIT Press, 1990.

[AM94] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 1994.

In print. Also Technical report CS-R9358, CWI, Amsterdam, The Netherlands. Available via anonymous ftp at ftp.cwi.nl, or via xmosaic at http://www.cwi.nl/cwi/publications/index.html.

[AP90] K. R. Apt and D. Pedreschi. Studies in Pure Prolog: termination. In J. W. Lloyd, editor, *Symposium on Computational Logic*, pages 150–176. Springer-Verlag, 1990.

[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, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[Bau89] M. Baudinet. *Logic Programming Semantics: Techniques and Applications.* PhD thesis, Stanford University, Stanford, California, 1989.

[BC94] A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In G. Levi and M. Rodríguez-Artalejo, editors, *Proc. Fourth Int'l Conf. on Algebraic and Logic Programming*, volume 850 of *Lecture Notes in Computer Science*, pages 269–286. Springer-Verlag, Berlin, 1994.

[BCE96] A. Bossi, N. Cocco, and S. Etalle. Transformation of Left Terminating Programs: The Reordering Problem. Technical Report CS96-1, Dip. Matematica Applicata e Informatica, Università Ca' Foscari di Venezia, Italy", 1996.

[BCF94] A. Bossi, N. Cocco, and M. Fabris. Norms on Terms and their use in Proving Universal Termination of a Logic Program. *Theoretical Computer Science*, 124:297–328, 1994.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BE94] A. Bossi and S. Etalle. Transforming Acyclic Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1081–1096, July 1994.

[CG94] J. Cook and J.P. Gallagher. A transformation system for definite programs based on termination analysis. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation.* Springer-Verlag, 1994.

[CS77] K.L. Clark and S. Sickel. Predicate logic: a calculus for deriving programs. In *Proceedings of IJCAI'77*, pages 419–120, 1977.

[DSF93] S. Decorte, D. De Schreye, and M. Fabris. Automatic Inference of Norms: a Missing Link in Automatic Termination Analysis. In D. Miller, editor, *Proc. 1993 Int'l Symposium on Logic Programming.* The MIT Press, 1993.

[EG94] S. Etalle and M. Gabbrielli. Transformations of CLP Modules. Technical Report CS-R9515, CWI, Amsterdam, 1994.

[Hog81] C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, April 1981.

[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.

[KK90] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. *Theoretical Computer Science*, 75(1&2):139–156, 1990.

[Kom82] H. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, pages 255–267. ACM, 1982.

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

[N.J84] Mycroft A. N.Jones. Stepwise development of operational and denotational semantics for Prolog. In *International Symposium on Logic Programming, Atlantic City, NJ, (U.S.A.)*, pages 289–298, 1984.

[Plü90] L. Plümer. *Termination Proofs for Logic Programs.* Lecture Notes in Artificial Intelligence 446. Springer-Verlag, 1990.

[PP91] M. Proietti and A. Pettorossi. Semantics preserving transformation rules for prolog. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91).* ACM press, 1991.

[Sek91] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86(1):107–139, 1991.

[Sek93] H. Seki. Unfold/fold transformation of general logic programs for the Well-Founded semantics. *Journal of Logic Programming*, 16(1&2):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.