Logic Programs as Term Rewriting Systems

Massimo Marchiori

Department of Pure and Applied Mathematics
University of Padova
Via Belzoni 7, 35131 Padova, Italy
`max@hilbert.math.unipd.it`

Abstract

This paper studies the relationship between logic programs and term rewriting systems (TRSs). A compositional transform is defined which given a logic program computes a TRS. For a relevant class of logic programs, called Simply Well Moded (SWM), there is a one-to-one correspondence between computed answer substitutions of the logic program and normal forms of the corresponding TRS. Moreover the transform preserves termination, i.e., a logic program terminates iff the corresponding TRS terminates. This transform is refined in such a way that the above results hold for a relevant class of unification free programs containing SWM, the class of Flatly Well Moded (FWM) programs.

Note: This work was done during an author's stay at CWI, Amsterdam.

1 Introduction

The study of transforms from logic programs into term rewriting systems (TRSs for short) is an important topic. Theoretically, it is interesting to analyze which classes of logic programs can be regarded as TRSs in disguise. Practically, using an appropriate transform allows to infer properties of a logic program from properties of the corresponding TRS. For instance, this method has been successfully applied in a number of works, to study termination of logic programs ([SRK90, RKS92, GW92, AM93]), using transforms that preserve non-termination.

Transforms that compute TRSs which not only preserve non-termination, but are equivalent to the corresponding logic programs, have been investigated only very recently ([Mar93, AM93]). Equivalence means that the logic program terminates iff the corresponding TRS terminates, and that computed answer substitutions of the logic program and normal forms of the TRS are in one-to-one correspondence.

The aim of this paper is to provide a *compositional* transform which maps a logic program into an equivalent TRS. Compositionality is fundamental because, for instance, a compositional transform allows to transform a logic program into a TRS in a modular way, i.e., clause by clause. The classes of logic programs we consider are contained in the so-called well moded programs ([DM85]). Moreover the transform can deal with a broader class of programs, where types instead of modes are considered.

The contribution of this paper can be summarized as follows.

First, a compositional transform Tswm is introduced which allows to translate into a class of equivalent TRSs both the class of Simply Well Moded programs of [AE93] and a wider class of programs which are *not well moded*, namely the Simply and Well Typed programs (SWT) of [AE93].

It is also shown how with little modifications Tswm can be extended to the whole class of Well Moded programs, losing equivalence, but still providing a relevant correspondence, in the sense that if the TRS terminates then the corresponding logic program terminates as well.

Finally, it is introduced a second transform called Tfwm that extends Tswm by translating the class of *flatly well moded* programs (FWM), introduced in [Mar94], into equivalent TRSs. The FWM class has been proven to be locally *maximal* among the unification free ones. Moreover, it is the locally greatest class of unification free programs containing SWM. Informally, a class of logic programs is locally maximal (resp. greatest) among the unification free programs if it is maximal (resp. greatest) among those classes of unification free programs that can be characterized by means of clause by clause syntactical criterions (see [Mar94] for a formal definition). Recall that a logic program is *unification free* if unification in the resolution process can be performed by means of iterated applications of the simpler pattern matching ([MK85]). These results can be roughly summarized in the following slogan:

$$\text{Logic Programs} - \text{Unification} \subseteq \text{Term Rewriting Systems}$$

1.1 Relations with Previous Work

The idea to transform logic programs into TRSs started with [SRK90], and has been further refined in [RKS92, GW92, AM93]. The main stimulus in this field was the study of termination of logic programs: a transform is defined with the property that if a logic program does not terminate, then the corresponding TRS does not terminate as well. Thus, a *sufficient* criterion for proving the termination of a logic program consists of checking whether the transformed TRS terminates.

This approach is attractive because TRSs enjoy powerful techniques to prove termination, as path orderings for instance (see [Der87, DJ90]), whereas for logic programs the situation is far more complicated.

In this paper the above transformational methodology is taken a step further, in the sense that we introduce transforms that fully translate a logic program: this way a *sufficient* but also *necessary* criterion for termination can be given.

A transform treating this kind of correspondence was introduced in [AM93]. However, this transform is not compositional. Also, the transform Tfwm here presented is able to transform classes of logic programs that are completely *out of scope* from the previous transforms, a drawback observed also by the same authors of [GW92, AM93].

The paper is organized as follows: Section 2 gives the necessary preliminaries; Section 3 introduces the concept of transform and a formal setting for its analysis. In Section 4 we specify the languages used both for the logic programs and the corresponding TRSs, and the basic rewrite rules used in the transforms. In Section 5 the transform Tswm is defined, and its completeness is proven w.r.t. the class SWM. Section 6 introduces some techniques to simplify the output of a transform, and illustrates its application to prove the termination of a logic program by means of an example. Section 7 introduces the transform Tfwm, that is proven complete for the class FWM. The relevance of this transform is illustrated by means of a significant example. Finally, Section 8 concludes with some important remarks. Proofs are here omitted or only sketched: they can be found in the full version of this paper.

2 Preliminaries

We assume familiarity with the basic notions regarding logic programs ([Llo87, Apt90]) and term rewriting systems ([Klo92]).

In this paper we will deal only with so-called *LD-derivations* (briefly *derivations*), that is SLD-derivations in which the leftmost selection rule is used.

With $\text{TERM}(F)$ we will denote the set of terms built up from some function set F. Sequences of terms will be written in vectorial notation (e.g. $\bar{t}$), and with $t^{(i)}$ we will denote the i-th term in $\bar{t}$.

Given a family S of objects (terms, atoms, etc.), $\text{Var}(S)$ is the set of all the variables contained in it; moreover, S is said *linear* iff no variable occurs more than once in it; objects $O_1, \ldots, O_n$ will be said *disjoint* iff $1 \leq i < j \leq n \Rightarrow \text{Var}(O_i) \cap \text{Var}(O_j) = \emptyset$. $s\,\mathcal{U}\,t$ will mean that the terms s and t are disjoint and unifiable. With $t[A_1/t_1, \ldots, A_n/t_n]$ we will mean the term obtained by t replacing every occurrence in t of the symbol A_i with t_i $(1 \leq i \leq n)$. For the sake of simplicity, we will sometimes omit brackets from the argument of unary functions (e.g. $f(g(X))$ may be written fgX), and if $\bar{t} = t_1, \ldots, t_n$ the sequence $f(t_1), \ldots, f(t_n)$ will be indicated with $f\bar{t}$. Also, given two sequences $\bar{s} = s_1, \ldots, s_n$ and $\bar{t} = t_1, \ldots, t_m$, we will denote by $\bar{s}, \bar{t}$ the sequence $s_1, \ldots, s_n, t_1, \ldots, t_m$.

2.1 Programs

We call (logic) program a finite set of Horn clauses and goals. If the program has no goals, it is called a *proper program*.

Let P be a program, P_2 the set of all the goals in P, and $P_1 = P \setminus P_2$. We call *derivation of* P a derivation of $P_1 \cup \{G\}$, when G is in P_2. Analogously, we call *computed answer substitution of* P a computed answer substitution of $P_1 \cup \{G\}$, where G is in P_2. The set of all the computed answer substitutions of P is denoted by $cas(P)$.

This formalism is well suited when formulating program properties, since it allows not to treat separately the clauses and the goal. Moreover, it is a natural choice when using *regular* properties (see below), that essentially do not distinguish between goals and clauses.

Since we are going to transform logic programs in term rewriting systems, the same convention will be adopted for TRSs as well (reading 'goal' as 'term', 'clause' as 'rewrite rule' and 'derivation' as 'reduction'): hence an extended TRS is a finite set of rules and a possibly infinite set of terms. The analogous of the set $cas(P)$ for a TRS T is the set $NF(T)$ of its normal forms.

The class of these 'extended' logic programs (resp. TRSs) will be denoted by LPext (resp. TRSext).

2.2 Regularity

Definition 2.1 A property $\mathcal{P}$ is said *regular* if

$$\leftarrow B_1, \ldots, B_n \in \mathcal{P} \;\Leftrightarrow\; \texttt{START} \leftarrow B_1, \ldots, B_n \in \mathcal{P}$$

where START is a new nullary predicate symbol. □

Using regularity will prove useful to elegantly shorten the description of the properties we will use in this paper, like well modedness for example, and also the definition of the transforms, since from now on *we identify a goal with the corresponding clause having the special symbol* START *in the head, and analogously for TRSs, we identify a term with the corresponding rule having the special symbol START in the left hand side* (cf. Subsection 4.1 for the language setting).

Although this will not concern us, it is worthwhile to notice that regularity is far from being merely a 'syntactic sugaring' tool, but has an importance for its own (see [Mar94]).

2.3 Modes and Well Modedness

Definition 2.2 A *mode* for a n-ary predicate p is a map from $\{1, \ldots, n\}$ to $\{\text{in}, \text{out}\}$. A *moding* is a map associating to every predicate p a mode for it. □

An argument position of a moded predicate is therefore said input (output) if it is mapped by the mode into in (out). For every predicate p, the number of its arguments (i.e. the arity), input positions and output positions will be denoted respectively by $\#\mathtt{p}$, $\#_{\text{in}}\mathtt{p}$ and $\#_{\text{out}}\mathtt{p}$. Moreover, we adopt the convention to write $\mathtt{p}(\bar{\mathtt{s}}, \bar{\mathtt{t}})$ to denote a moded atom p having its input positions filled in by the sequence of terms $\bar{\mathtt{s}}$, and its output positions filled in by $\bar{\mathtt{t}}$.

Now let us recall the following well-known property:

Definition 2.3 A program is *data driven* if every time an atom is selected in a goal during a derivation, its input arguments are ground. □

We now state the basic notion of well moding:

Definition 2.4 A program satisfies the (regular) *well modedness (WM)* property if for every its clause $C = \mathtt{p}_0(\bar{\mathtt{t}}_0, \bar{\mathtt{s}}_{n+1}) \leftarrow \mathtt{p}_1(\bar{\mathtt{s}}_1, \bar{\mathtt{t}}_1), \ldots, \mathtt{p}_n(\bar{\mathtt{s}}_n, \bar{\mathtt{t}}_n)$ it holds:

$$\forall i \in [1, n+1] : \ \mathrm{Var}(\bar{\mathtt{s}}_i) \subseteq \bigcup_{j=0}^{i-1} \mathrm{Var}(\bar{\mathtt{t}}_j)$$ □

Well Moded programs enjoy the following relevant property:

Theorem 2.5 ([DM85]) *Well Moded programs are data driven.*

2.4 Simply Well Modedness

First we introduce the following class of programs:

Definition 2.6 A program satisfies the (regular) *Simply Modedness (SM)* property if for every its clause $C = \mathtt{p}_0(\bar{\mathtt{s}}_0, \bar{\mathtt{t}}_0) \leftarrow \mathtt{p}_1(\bar{\mathtt{s}}_1, \bar{\mathtt{t}}_1), \ldots, \mathtt{p}_n(\bar{\mathtt{s}}_n, \bar{\mathtt{t}}_n)$ it holds:

- The sequence $\bar{\mathtt{t}}_1, \ldots, \bar{\mathtt{t}}_n$ is linear, composed only by variables and disjoint with $\bar{\mathtt{s}}_0$
- $\forall i \in [1, n] : \mathrm{Var}(\bar{\mathtt{s}}_i) \cap \bigcup_{j=i}^{n} \mathrm{Var}(\bar{\mathtt{t}}_j) = \emptyset$ $\square$

We will call *simply well moded (SWM)* a program which is both SM and WM: this is the main class proved in [AE93] to be unification free. Interestingly, this class is quite large (see for instance the list of programs presented in the paper just cited).

3 Transforms

We introduce here a formal setting for the analysis of fundamental properties of transforms. In this paper, S will always denote a class of logic programs, $GOALS$ and $SUBST$ the sets of all the goals and substitutions, respectively. Moreover, $\wp$ is the usual powerset operator.

Definition 3.1 A *transform* is a map τ : LPext $\rightarrow$ TRSext $\square$

Definition 3.2 A transform τ is *semi-complete (resp. sound) for S* if:
1) If $P \in S$, then P has an infinite derivation $\Longrightarrow$ (resp. $\Longleftarrow$) $\tau(P)$ has an infinite reduction.
2) There is a computable map γ_τ : $GOALS \times$ TRSext $\rightarrow \wp(SUBST)$ such that for every goal $G \in S$ and every proper program $P \in S$ it holds $\gamma(G, NF(\tau(G \cup P))) \supseteq$ (resp. $\subseteq$) $cas(G \cup P)$. $\square$

Notice that transforms in [SRK90, RKS92, GW92] are trivially semi-complete in the sense that their γ map is just $\gamma(G, NF(\tau(G \cup P))) = SUBST$.

Definition 3.3 A transform τ is said *complete for S* if:
1) If $P \in S$, then P has an infinite derivation if and only if $\tau(P)$ has an infinite reduction.
2) There is a computable map α_τ : $GOALS \times$ TRSext $\rightarrow \wp(SUBST)$ (called an *answer map* of τ) such that for every goal $G \in S$ and every proper program $P \in S$ it holds $\alpha_\tau(G, NF(\tau(G \cup P))) = cas(G \cup P)$. $\square$

3.1 Compositionality

A transform, as defined in Def. 3.1, can be arbitrarily complex as far as the computational aspect is concerned. A very important concept in developing a satisfactory transform is the following:

Definition 3.4 A transform τ is *compositional* if for every two programs P, P'

$$\tau(P \cup P') = \tau(P) \cup \tau(P') \qquad \square$$

Thus, a compositional transform of a program is equal to the union of the transforms of its clauses.

The advantages of a compositional definition are well-known: to compute the transform of a program the transforms of its clauses can be executed in parallel. Moreover, to understand the relationship between logic programs and TRSs it suffices to consider a logic program with one clause. Finally, in case a new set of clauses D is added to P obtaining a new program P', to calculate the transformed $\tau(P')$ we don't have to redo the whole computation again, but only to transform D, since $\tau(P') = \tau(P \cup D) = \tau(P) \cup \tau(D)$. This may be of great advantage, especially when the program P is big enough. The general case in which P is modified into P', possibly also *deleting* some of its clauses, will be treated later in Section 8.

4 Setting the Stage

4.1 Languages

A *moded language* is a pair (L, m) where L is a language and m is a moding for L. Every logic program is written in a moded language $\mathcal{L}_{LP} = (\{\mathtt{p}_0, \mathtt{p}_1, \ldots; \mathtt{f}_0, \mathtt{f}_1, \ldots\}, m)$ where the $\mathtt{p}_i$ are the relation symbols and the $\mathtt{f}_i$ are the function symbols (we consider constants like functions of null arity).

For every natural n, we assume that in $\mathcal{L}_{LP}$ there is an n-ary function $\otimes_n$: such functions are used as 'cartesian product' operators.

The functional language $\mathcal{L}_{TRS}$ for the TRS that will be associated with the considered programs is described as follows:

- a function p_i for every $\mathtt{p}_i \in \mathcal{L}_{LP}$, with $\#p_i = \#_{\text{in}}\mathtt{p}_i$
- a function f_i for every $\mathtt{f}_i \in \mathcal{L}_{LP}$, with $\#f_i = \#\mathtt{f}_i$
- functions $(f_i)^{-1}_1, \ldots, (f_i)^{-1}_{\#\mathtt{f}_i}$ for every $\mathtt{f}_i \in \mathcal{L}_{LP}$, with $\#(f_i)^{-1}_j = 1$ (called 'inverse functions')
- an unary function $\mathcal{N}$
- unary functions $\underset{n}{\mathcal{T}}$ (for every $n \in \mathbb{N}$) called 'transition functions'.

$\mathcal{N}$ is a marker associated to terms t (of $\mathcal{L}_{TRS}$): $\mathcal{N}t$ indicates that the term t is in normal form in the considered TRS (hence the name $\mathcal{N}$). Intuitively this means that t has already been computed.

Inverse functions are used to select an argument from a term which is marked (thus already computed): for instance the function $(f_i)^{-1}_j$ applied to the term $\mathcal{N}f_i(t_1, \ldots, t_n)$ yields the term $\mathcal{N}t_j$.

Transition functions are used to rewrite a term which is marked: for example to rewrite a term $\mathcal{N}X$ into the term $f(\pi_1\mathcal{N}X, \mathcal{N}X)$ the function $\underset{\rho(f(\pi_1\bullet,\bullet))}{\mathcal{T}}$ is used, where $\bullet$ is a placeholder (that will be replaced with $\mathcal{N}X$) and $\rho(f(\pi_1\bullet,\bullet))$ is an appropriate natural number encoding of $f(\pi_1\bullet,\bullet)$.

Since the $\otimes_n$ are used to represent cartesian products, the notation π^n_i is used to indicate the inverse function $(\otimes_n)^{-1}_i$; usually instead of π^n_i and $\otimes_n(t_1, \ldots, t_n)$ we will simply write π_i, and $\langle t_1, \ldots, t_n\rangle$, respectively.

The subset $\{f_i \mid \# f_i > 0\}$ of $\mathcal{L}_{TRS}$, that is the corresponding of the proper functions in $\mathcal{L}_{LP}$, will be denoted by $\textsc{Func}_{TRS}$, whereas the set of all terms built on $\mathcal{L}_{TRS}$, i.e. $\mathcal{T}(\mathcal{L}_{TRS})$, will be denoted by $\textsc{Term}_{TRS}$.

4.2 Propagation

The following definition specifies the only rules that rewrite terms of the form $f(t_1, \ldots, t_n)$, with $f \in \textsc{Func}_{TRS}$.

Definition 4.1 The *propagation rules* are:

$$\begin{array}{ll} f(\mathcal{N}X_1, \ldots, \mathcal{N}X_n) \to \mathcal{N}f(X_1, \ldots, X_n) & (f \in \textsc{Func}_{TRS}) \\ (f)_i^{-1}\mathcal{N}f(X_1, \ldots, X_n) \to \mathcal{N}X_i & (f \in \textsc{Func}_{TRS}, 1 \le i \le n = \# f) \end{array} \quad \square$$

The first schema of rules expresses the fact the $\mathcal{N}$ operator 'propagates' thru terms in $\textsc{Func}_{TRS}$.

The second schema is more interesting: it shows that to apply an inverse function we have to wait for the argument to be computed (marked by $\mathcal{N}$). Naïve rules of the form $f_i^{-1}f(X_1, \ldots, X_n) \to X_i$ won't work: intuitively, suppose a term t reduces to two normal forms (i.e. it gives two 'results'), say $\langle t_1, t_2\rangle$ and $\langle s_1, s_2\rangle$, and suppose that the first projection π_1 as well as the second projection π_2 are used. Then by applying the naïve rules $\pi_1\langle X, Y\rangle \to X$ and $\pi_2\langle X, Y\rangle \to Y$ one can obtain the following two reductions $\pi_1 t \twoheadrightarrow \pi_1\langle t_1, t_2\rangle \to t_1$ and $\pi_2 t \twoheadrightarrow \pi_2\langle s_1, s_2\rangle \to s_2$, thus violating the fundamental property of the projections, i.e. $\langle \pi_1 t, \pi_2 t\rangle = t$. The rules of Definition 4.1 allow to avoid such 'splitting' phenomena; we will see later (Subsection 6.3) how to relax these rules whenever these bad cases do not happen.

4.3 Transition

Transition rules will be used in the transform to encode a clause of a program by means of a rewrite rule. In $\underset{n}{\mathcal{T}}$, the index n encodes the transition that has to be performed, using the following (injective and computable) encoding function ρ:

$$\rho : \textsc{Term}(\mathcal{L}_{TRS} \dot\cup \{\bullet\}) \to \mathbb{N}$$

The new symbol $\bullet$ is introduced to represent the argument of the transition function, as will become clear from the following definition:

Definition 4.2 The *transition rules* are defined as follows:

$$\underset{\rho(t)}{\mathcal{T}} \mathcal{N}X \to t[\bullet/\mathcal{N}X] \qquad (t \in \textsc{Term}(\mathcal{L}_{TRS} \dot\cup \{\bullet\})) \quad \square$$

For simplicity, we write $\underset{t}{\mathcal{T}}$ instead of $\underset{\rho(t)}{\mathcal{T}}$; so for example $\underset{f(\pi_1\bullet,\bullet)}{\mathcal{T}}$ will have as corresponding transition rule $\underset{f(\pi_1\bullet,\bullet)}{\mathcal{T}} \mathcal{N}X \to f(\pi_1\mathcal{N}X, \mathcal{N}X)$.

5 The Transform Tswm

We introduce here the *T*ransform for *s*imply *w*ell *m*oded programs, called Tswm.

Roughly, a clause $C : A \leftarrow B_1, \ldots, B_n$ is transformed into a rewrite rule r, where the left hand side (lhs) of r is a TRS term obtained transforming the head of C, while the right hand side (rhs) of r is constructed incrementally in a sequence of $n+1$ steps, where in the i-th step $(1 \leq i \leq n)$ the (partial) rhs so far constructed is modified using the atom B_i, and in the $n+1$-th step the atom A is used. To this end two variables rhs and V are used, where rhs contains the part of the right end side of r so far computed, and V is a sequence of terms in $\textsc{Term}_{TRS}$ used to store the output arguments of the atoms in the clause.

First, we need the following definition to 'extract' arguments from a term:

Definition 5.1 Given a term $t \in \textsc{Term}_{TRS}$ and a variable $X \in \mathrm{Var}(t)$, a *suitable expression of* X *from* t (notation $\mathrm{SE}_X(t)$) is defined as $SE_X(t)(\bullet)$, where:

i) $SE_X(X)(u) = u$

ii) $SE_X(f(t_1, \ldots, t_n))(u) = SE_X(t_i)(f_i^{-1}(u)) \qquad \text{if } X \in \mathrm{Var}(t_i),\ i \in [1, n]$

Generalizing, in case $t, s \in \textsc{Term}_{TRS}$ and $\mathrm{Var}(t) \supseteq \{X_1, \ldots, X_n\} = \mathrm{Var}(s)$ the suitable expression of s from t is defined as $\mathrm{SE}_s(t) = \tilde{s}[X_1/\mathrm{SE}_{X_1}(t), \ldots, X_n/\mathrm{SE}_{X_n}(t)]$, where $\tilde{s}$ is obtained from s replacing every constant c by $\mathcal{N}c$. □

Note that a suitable expression is in general not unique because in point ii) there may be more than one t_i containing X. However this is not a problem, since we don't care about different occurrences of the same variable: to obtain uniqueness, one can simply change the definition taking the minimum i.

Example 5.2 If $t = f(X, g\langle Y \rangle)$ then $\mathrm{SE}_X(t) = f_1^{-1}\bullet$, $\mathrm{SE}_Y(t) = \pi_1 g^{-1} f_2^{-1}\bullet$, and $\mathrm{SE}_{\langle X, f(a,Y)\rangle}(t) = \langle X, f(\mathcal{N}a, Y)\rangle[X/\mathrm{SE}_X(t), Y/\mathrm{SE}_Y(t)] =$
$\langle X, f(\mathcal{N}a, Y)\rangle[Z/f_1^{-1}\bullet, Y/\pi_1 g^{-1} f_2^{-1}\bullet] = \langle f_1^{-1}\bullet, f(\mathcal{N}a, \pi_1 g^{-1} f_2^{-1}\bullet)\rangle$. □

Definition 5.3 (Transform Tswm)
Let the clause be $C = \mathtt{p}(\bar{\mathtt{t}}_0, \bar{\mathtt{s}}_{n+1}) \leftarrow \mathtt{p}_1(\bar{\mathtt{s}}_1, \bar{\mathtt{t}}_1), \ldots, \mathtt{p}_n(\bar{\mathtt{s}}_n, \bar{\mathtt{t}}_n)$.
The lhs of the rewrite rule is $lhs := p\mathcal{N}\langle \bar{t}_0 \rangle$.
The rhs is built iteratively: the atoms in the body are transformed in left to right order from $\mathtt{p}_1$ to $\mathtt{p}_n$, using a sequence V of terms. The following sequence of actions is used:
– $rhs := \mathcal{N}\langle X_1, \ldots, X_k\rangle$ and $V := X_1, \ldots, X_k$ (where $\{X_1, \ldots, X_k\} = \mathrm{Var}(\bar{t}_0)$).
– For $i := 1$ to n do $\left\{ rhs := \underset{\mathrm{SE}_{\langle V, p_i\langle \bar{s}_i\rangle\rangle}\langle V\rangle}{\boldsymbol{\mathcal{T}}}(rhs) \quad ; \quad V := V, \langle \bar{t}_i \rangle \right\}$
– If C is not a goal (viz. $\mathtt{p} \neq \mathtt{START}$) then let $rhs := \underset{\mathrm{SE}_{\langle \bar{s}_{n+1}\rangle}\langle V\rangle}{\boldsymbol{\mathcal{T}}}(rhs)$

The produced rewrite rule is then $lhs \rightarrow rhs$.
This rule has occurrences of functions (inverse or not) and of $\boldsymbol{\mathcal{T}}$ operators: for each of them, include the corresponding propagation and transition rules. □

Assumption: Since propagation and transition rules can be inferred from the rule $lhs \rightarrow rhs$, henceforth we will omit them when writing the TRS obtained from the transform of a program.

Example 5.4 Take the classical append program, with moding ap(out,out,in):

```
ap([],X,X) ←
ap([A|X],Y,[A|Z]) ← ap(X,Y,Z)
```

Then its transform is:

$$\begin{aligned} ap\mathcal{N}\langle X\rangle &\to \underset{\langle \mathcal{N}[\,],\pi_1\bullet\rangle}{\mathcal{T}}\langle \mathcal{N}X\rangle \\ ap\mathcal{N}\langle [A|Z]\rangle &\to \underset{\langle [\pi_1\bullet|\pi_1\pi_3\bullet],\pi_2\pi_3\bullet\rangle}{\mathcal{T}}\ \underset{\langle \pi_1\bullet,\pi_2\bullet,ap\langle\pi_2\bullet\rangle\rangle}{\mathcal{T}}\mathcal{N}\langle A,Z\rangle \end{aligned}$$

□

5.1 Built-in

As seen, the transform Tswm translates each clause C into a (finite) TRS consisting of one rule ($lhs \to rhs$) together with the corresponding propagation and transition rules. We call $lhs \to rhs$ the *main rule* produced by the transform, denoted by $\mathsf{Tswm}_{\mathcal{M}}(C)$, while the remaining rules are called *built-in rules* (notation $\mathsf{Tswm}_{\mathcal{B}}(C)$): hence $\mathsf{Tswm}(C) = \mathsf{Tswm}_{\mathcal{M}}(C) \cup \mathsf{Tswm}_{\mathcal{B}}(C)$.

The intuition is that these built-in rules are just toolbox accessories for the main rule.

Thus, let us say the *built-in* $\mathcal{B}$ is the TRS composed by both the propagation and transition rules (note this is an infinite TRS, so not in TRSext).

Later on, we will improve the transform adding rules to the built-in: *so we assume that, when translating a clause C, its transform is given by the main rule(s) plus the rules from $\mathcal{B}$ that have some symbol in common with the symbols in the main rule(s).* This assumption will stay valid also for the subsequent transform Tfwm of Section 7.

It is evident we could replace $\mathsf{Tswm}_{\mathcal{B}}(C)$ with $\mathcal{B}$ itself without affecting the properties of the TRS produced by Tswm. Thus we can study $\mathcal{B}$ as a whole instead of its finite subTRSs that are generated by Tswm.

A first relevant property $\mathcal{B}$ should enjoy is *completeness*. This is not the case, because $\mathcal{B}$ is terminating but not confluent (consider the term $f^{-1}\mathcal{N}f\mathcal{N}X$: in $\mathcal{B}$ it reduces to two different normal forms $\mathcal{N}\mathcal{N}X$ and $f^{-1}\mathcal{N}\mathcal{N}fX$). To recover completeness, one can add the following rule:

$$\mathcal{N}\mathcal{N}X \to \mathcal{N}X \qquad (\textit{idempotence rule})$$

that is plausible, expressing the fact that marking twice a term (by means of $\mathcal{N}$) is the same than marking it once. Nevertheless, $\mathcal{B}$ is complete *relatively* to the terms produced by Tswm itself.

Summing up, we have the following:

Lemma 5.5 *$\mathcal{B}\cup\{\mathcal{N}\mathcal{N}X \to \mathcal{N}X\}$ is complete, whereas $\mathcal{B}$ is complete only relatively to the terms produced by the transform itself.*

To have a theoretically cleaner treatment, we *include* the idempotence rule in the built-in, being aware it does not play a role operationally. So we can reformulate the above lemma saying that $\mathcal{B}$ is complete.

Another important property the built-in enjoys is the following (roughly speaking, two TRSs *commute* if it does not matter in which order we apply rules from a TRS or from the other, see [Klo92] for a formal definition):

Lemma 5.6 *For every program P, the built-in commutes with* $\mathsf{Tswm}_{\mathcal{M}}(P)$.

5.2 Completeness of Tswm

The transform Tswm satisfies the following fundamental property:

Theorem 5.7 Tswm *is a complete (compositional) transform for SWM.*

The proof of the above theorem is not easy and quite long. The idea is that we define an *embedded sequence* from the sequence of goals $G_0, G_1, , \ldots$ of an LD-derivation $G_0, C_0, G_1, C_1, \ldots$ in the following way ($\downarrow_T$ means normalization w.r.t. the TRS T):

$$\begin{aligned} &\mathcal{E}(G_0) = \mathsf{Tswm}_{\mathcal{M}}(G_0)\downarrow_{\mathcal{B}} \\ &\mathcal{E}(G_i + 1) = (\mathcal{E}(G_i)\downarrow_{\mathsf{Tswm}_{\mathcal{M}}(C_i)})\downarrow_{\mathcal{B}} \end{aligned}$$

Notice that Lemma 5.5 guarantees that the normalization process w.r.t. $\mathcal{B}$ is well-defined, i.e. it terminates and yields an unique result. Moreover, Lemma 5.6 ensures that normalization w.r.t. $\mathcal{B}$ does not prevent the applicability of the main rules. To show that $\downarrow_{\mathsf{Tswm}_{\mathcal{M}}(C_i)}$ is well-defined the following two lemmata are used (in fact, this normalization process reduces to just a one-step reduction):

Lemma 5.8 *A goal $G \in$ SWM and a clause $C \in$ SWM have a resolvent iff the rule* $\mathsf{Tswm}_{\mathcal{M}}(C)$ *can be applied to the term* $\mathsf{Tswm}_{\mathcal{M}}(G)\downarrow_{\mathcal{B}}$.

Lemma 5.9 *For every goal G and clause C, the rule* $\mathsf{Tswm}_{\mathcal{M}}(C)$ *can be applied to the term* $\mathsf{Tswm}_{\mathcal{M}}(G)\downarrow_{\mathcal{B}}$ *at most in one position (that is, there are no ambiguities).*

Then, a map is defined that associates to every member $\mathcal{E}(G_i)$ of the embedded sequence the partial answer substitution of G_i. This map, when $G_i = \square$, gives precisely the answer map α_{Tswm} of Definition 3.3, which is now defined:

If the goal is $G = \leftarrow \mathtt{p}_1(\bar{\mathtt{s}}_1, \bar{\mathtt{t}}_1), \ldots, \mathtt{p}_n(\bar{\mathtt{s}}_n, \bar{\mathtt{t}}_n)$ and $W = \mathrm{Var}(\bar{\mathtt{t}}_1, \ldots, \bar{\mathtt{t}}_n)$, then the answer map α_{Tswm} is

$$\alpha_{\mathsf{Tswm}}(G, \mathit{NF}(\mathsf{Tswm}(G \cup P))) := \{\theta|_W : \langle\langle \bar{t}_1\rangle, \ldots, \langle \bar{t}_n\rangle\rangle\theta = t, \mathcal{N}t \in \mathit{NF}(\mathsf{Tswm}(G \cup P))\}$$

where $\theta|_W$ is the restriction of θ to the variables in W.

6 Simplifying the Transform

In this section we show how, by using simple techniques, the output produced by the transform can be greatly simplified, providing a more readable (and easier to analyze) TRS.

6.1 Normalization

One great simplification comes from normalizing the rhs of the main rewrite rule obtained by the transform w.r.t. its built-in. This in general proves quite useful because in Subsection 6.3 we will increase the power of the built-in adding simplifying rules, that enter in action just thanks to this technique (that is, we will normalize w.r.t. the extended built-in).

Example 6.1 As a first example of what is possible using this technique, consider the program append seen previously in Example 5.4. If the obtained main rules are normalized w.r.t. the built-in, we obtain finally:

$$\begin{array}{l} ap\mathcal{N}\langle X\rangle \rightarrow \mathcal{N}\langle [\,], X\rangle \\ ap\mathcal{N}\langle [A|Z]\rangle \rightarrow \underset{\langle [\pi_1\bullet|\pi_1\pi_2\bullet],\pi_2\pi_2\bullet\rangle}{\mathcal{T}} \langle \mathcal{N}A, \mathcal{N}Z, ap\mathcal{N}\langle Z\rangle\rangle \end{array} \qquad \square$$

6.2 Smart Referencing

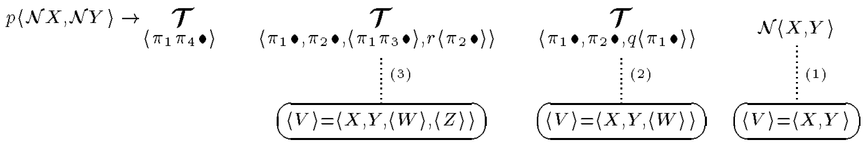
The transition functions used in the construction of the transform maintain a growing sequence (V) of all the output arguments in the body of the clause: however, all this information is not needed in case no further references are made to a variable occurring either in some output argument or in the input arguments of the head of the clause. To prevent such a waste, we can refine the transform letting V be the set of variables really needed in the sequel (*smart referencing*): more formally, the difference with the old transform is that we drop from V the variables no more present neither in the input arguments of the remaining atoms to be processed, nor in the output arguments of the head atom.

An example will clarify the situation.

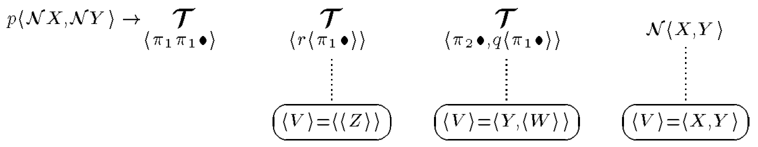
Example 6.2 Consider the clause

```
p(X,Y,Z) ← q(X,W),r(Y,Z)
```

with moding p(in,in,out), q(in,out), r(in,out).
Its transform via Tswm gives


$p\langle \mathcal{N}X,\mathcal{N}Y\rangle \rightarrow \underset{\langle\pi_1\pi_4\bullet\rangle}{\mathcal{T}}$
$\underset{\langle\pi_1\bullet,\pi_2\bullet,\langle\pi_1\pi_3\bullet\rangle,r\langle\pi_2\bullet\rangle\rangle}{\mathcal{T}}$
$\underset{\langle\pi_1\bullet,\pi_2\bullet,q\langle\pi_1\bullet\rangle\rangle}{\mathcal{T}}$
$\mathcal{N}\langle X,Y\rangle$
(3)
(2)
(1)
$\langle V\rangle=\langle X,Y,\langle W\rangle,\langle Z\rangle\rangle$
$\langle V\rangle=\langle X,Y,\langle W\rangle\rangle$
$\langle V\rangle=\langle X,Y\rangle$

But note how in (2) X is not needed, and in (3) Y and W are not needed as well: so the revised transform gives


$p\langle \mathcal{N}X,\mathcal{N}Y\rangle \rightarrow \underset{\langle\pi_1\pi_1\bullet\rangle}{\mathcal{T}}$
$\underset{\langle r\langle\pi_1\bullet\rangle\rangle}{\mathcal{T}}$
$\underset{\langle\pi_2\bullet,q\langle\pi_1\bullet\rangle\rangle}{\mathcal{T}}$
$\mathcal{N}\langle X,Y\rangle$
$\langle V\rangle=\langle\langle Z\rangle\rangle$
$\langle V\rangle=\langle Y,\langle W\rangle\rangle$
$\langle V\rangle=\langle X,Y\rangle$

$\square$

6.3 Relaxing the Built-in

Both the propagation and transition rules aim at constraining the calculus, forcing their argument(s), via the $\mathcal{N}$ operator, to be completely calculated before being further utilized.

As said, it is the possibility of 'splitting' that forces to behave this way (review Subsection 4.2): anyway, when this is not the case we can get rid of this imposed extra structure.

One such case is when unary functions are concerned: thus we add the following rules to the propagation rules that relax their behaviour (recovering the symmetry between a function and its inverse that was missing in the original propagation rules):

$$\begin{array}{ll} f^{-1}\mathcal{N}X \rightarrow \mathcal{N}f^{-1}X & (f \in \textsc{Func}_{TRS}, \#f = 1) \\ f^{-1}fX \rightarrow X & \quad '' \\ ff^{-1}X \rightarrow X & \quad '' \end{array}$$

Regarding the transition rules, instead, the following rules are added:

$$\underset{\rho(t)}{\mathcal{T}} X \rightarrow t[\bullet/X] \qquad \left(\begin{array}{c} t \in \textsc{Term}(\mathcal{L}_{TRS} \,\dot{\cup}\, \{\bullet\}) \text{ with} \\ \text{only one occurrence of } \bullet \end{array} \right)$$

This case is in all similar to the previous one since transition functions are 'masquerade' functions with argument $\bullet$, and hence their arity should be calculated in terms of occurrences of $\bullet$.

Adding all these rules to the built-in is useful because, as said earlier in Subsection 6.1, when we normalize w.r.t. the built-in much of the extra structure imposed disappears, simplifying a lot the obtained rules. Note also that for this extended built-in Lemmata 5.5 and 5.6 still hold.

Example 6.3 Consider the previous Example 6.2:

$$\texttt{p(X,Y,Z)} \leftarrow \texttt{q(X,W),r(Y,Z)}$$

was transformed, using smart referencing, into

$$p\langle \mathcal{N}X, \mathcal{N}Y \rangle \rightarrow \underset{\langle \pi_1\pi_1\bullet \rangle}{\mathcal{T}} \underset{\langle r\langle \pi_1\bullet \rangle\rangle}{\mathcal{T}} \underset{\langle \pi_2\bullet, q\langle \pi_1\bullet \rangle\rangle}{\mathcal{T}} \mathcal{N}\langle X, Y \rangle$$

This rewrite rule, normalized w.r.t. the built-in, is now

$$p\langle \mathcal{N}X, \mathcal{N}Y \rangle \rightarrow r\langle \pi_1\langle \mathcal{N}Y, q\mathcal{N}\langle X \rangle\rangle\rangle \qquad \square$$

Example 6.4 Take the clause

$$\texttt{p(X,Y)} \leftarrow \texttt{q(X,V),r(V,W,Z),s(Z,Y)}$$

moded `p`(in,out), `q`(in,out), `r`(in,out,out) and s(in,out).
Its transform using the naïve Tswm transform gives

$$p\mathcal{N}\langle X \rangle \rightarrow \underset{\langle \pi_1\pi_4\bullet \rangle}{\mathcal{T}} \underset{\langle \pi_1\bullet, \langle \pi_1\pi_2\bullet \rangle, \langle \pi_1\pi_3\bullet, \pi_2\pi_4\bullet \rangle, s\langle \pi_2\pi_3\bullet \rangle\rangle}{\mathcal{T}} \underset{\langle \pi_1\bullet, \langle \pi_1\pi_2\bullet \rangle, r\langle \pi_1\pi_2\bullet \rangle\rangle}{\mathcal{T}} \underset{\langle \pi_1\bullet, q\langle \pi_1\bullet \rangle\rangle}{\mathcal{T}} \mathcal{N}\langle X \rangle$$

whereas using (smart referencing and) the built-in extended with the aforementioned unary functions rules we get

$$p\mathcal{N}\langle X \rangle \rightarrow s\pi_2 rq\mathcal{N}\langle X \rangle \qquad \square$$

6.4 Termination of Logic Programs: an Example

Semi-complete transforms can be used to provide a *sufficient* criterion for proving termination of logic programs: a logic program P terminates if its transformed TRS $\tau(P)$ terminates. Clearly, a *complete* transform provides a *sufficient* and also *necessary* criterion, hence strengthening the analysis power.

Now, consider the following program taken from [RKS92] (and further cited in [GW92, AM93]), computing the transitive closure of a relation `p`:

```
p(a,b) ←          tc(X,Y) ← p(X,Y)
p(b,c) ←          tc(X,Y) ← p(X,Z),tc(Z,Y)
```

with moding `p`(in,out) and `tc`(in,out).

The transform developed in [RKS92] transforms this terminating logic program into a *non-terminating* TRS, whereas the transform Tswm, since the above program is SWM, obtains a TRS

$$\begin{array}{ll} p\mathcal{N}\langle a\rangle \to \mathcal{N}\langle b\rangle & tc\mathcal{N}\langle X\rangle \to p\mathcal{N}\langle X\rangle \\ p\mathcal{N}\langle b\rangle \to \mathcal{N}\langle c\rangle & tc\mathcal{N}\langle X\rangle \to tc\, p\mathcal{N}\langle X\rangle \end{array}$$

that can be proven terminating using standard techniques of term rewriting.

Another application of this technique to prove termination of logic programs is given in Subsection 7.1, using transform Tfwm.

7 The Transform Tfwm

Transform Tswm is complete for SWM. In this section we modify Tswm to get a more powerful, yet more complicated, transform which is complete for the broader class of *Flatly Well Moded* programs (FWM), after [Mar94].

The idea underlying this subclass of well moded programs is that, roughly, every time an atom is selected to resolve the head of a clause, its output arguments are all filled in with *flattening expression*: a flattening expression (briefly *FE*) is either a variable or a ground term or a so-called *steady term*, where a term t is said to be *steady* if t is linear and $t = f_{(1)} \cdots f_{(k)}(X_{(1)}, \ldots, X_{(n)})$ with $k > 0$ and $\#f_{(1)} = \ldots = \#f_{(k-1)} = 1 \leq \#f_{(k)}$. For example, $f(g(h(X,Y)))$ is steady, whereas $f(X,X)$, $f(g(X),Y)$ and $f(g(X),h(Y))$ are not.

FWM has the remarkable property of being not only locally maximal among the unification free classes of programs, but even the locally greatest one containing SWM (see the Introduction and see [Mar94] for details).

To define Tfwm, we need the following concepts:

Definition 7.1 The *degree* of a flattening expression is the map

$$\begin{array}{lll} ||t|| = 0 & \text{if } t \text{ is ground} \\ ||t|| = 1 & \text{if } t \text{ is a variable} \\ ||t|| = k+1 & \text{if } t = f_1 \cdots f_k(X_1, \ldots, X_n) \end{array}$$

□

Definition 7.2 The *truncation of a term t at level k* ($Trunc_k(t)$) is:
$Trunc_0(t) = t$
$Trunc_k(t) = s$ for s an FE, $||s|| = k$, $s\mathcal{U}t$ □

The transform Tfwm acts like the previous Tswm, except that it keeps track of the structure of the terms occurring in the output positions of the selected atom in the goal. When the head of the clause is considered, a number of terms are produced, and each of them is used as lhs of a corresponding rewrite rule. These terms are computed by considering the degrees of all the FEs which unify with the output arguments of the head. Each of these degrees combined with the head yields the lhs of a rewrite rule. To this end for an atom $\mathtt{p}(t,s)$ a triple $\langle t,s,d\rangle$ is used, where t is the input argument, s is the output argument and d is the degree of s. Notice that in Tswm only the input argument t is used to define the lhs of the correspondent rule. The rhs's of these rules are computed in a similar way as in the case of Tswm, only that this time, when processing the atoms in the body, also the structure of the current output, and its degree, is taken into account (the starting one is calculated by means of the clause head).

For example, consider the clause `p(X,f(a,Y))` $\leftarrow$ `q(X,Y)` moded p(in, out), q(in, out); `p(X,f(a,Y))` yields three rules: the first has lhs $p\mathcal{N}\langle X,f(a,Y),0\rangle$ (corresponding to the case that a 0-degree term, i.e. a ground one, is present in the output argument of the atom the clause head unifies with), and rhs $q\mathcal{N}\langle X,Y,0\rangle$ (if $f(a,Y)$ unifies with a ground term then Y is set to a ground term, hence a 0-degree FE as well); the second has lhs $p\mathcal{N}\langle X,Y,1\rangle$ (corresponding to the 1-degree case, i.e. a variable), and rhs $q\mathcal{N}\langle X,Y,1\rangle$ (if $f(a,Y)$ unifies with a variable then Y remains a variable, hence again a 1-degree FE); the third has lhs $p\mathcal{N}\langle X,f(Z,Y),2\rangle$ (corresponding to the 2-degree case, i.e. a steady term), and rhs $q\mathcal{N}\langle X,Y,1\rangle$ (if $f(a,Y)$ unifies with a 2-degree FE, the latter has the form $f(W_1,W_2)$, and so Y remains a variable, viz. a 1-degree FE).

The construction process of the main rule(s) is similar to the one in Tswm, hence for space reasons we simply state the *lhs* and the start *rhs*.

Definition 7.3 (Transform Tfwm**)** For simplicity, we assume that relation symbols have only one output argument, the general case being in all similar. Moreover, we apply at once smart referencing (Subsection 6.2).
So, consider a clause $\mathtt{p}(\bar{\mathtt{t}}_0,\mathtt{s}_{n+1})\leftarrow \mathtt{p}_1(\bar{\mathtt{s}}_1,\mathtt{t}_1),\ldots,\mathtt{p}_n(\bar{\mathtt{s}}_n,\mathtt{t}_n)$; we distinguish the following cases:

1. $\mathrm{Var}(s_{n+1})\subseteq \mathrm{Var}(\bar{t}_0)$
This means, for the data driveness property of WM programs, that s_{n+1} is instantiated to a ground term; the transformed rule is then:

$$p\mathcal{N}\langle \bar{t}_0,Y,Z\rangle \to \underset{\text{likewise Tswm}}{\text{Transition functions}}\, T(s_{n+1},Y,Z,\mathcal{N}\langle\bar{t}_0\rangle)$$

where $T(X,Y,Z,W)$ is a 'Test' operator that checks whether unification between the head and the selected atom in the goal can occur, and in affirmative case yields W. This can be done because we have the information (i.e. Z) about the *degree* of the flattening expression present in the output argument of the goal atom (i.e. Y). T is defined by the following rules:

i)	$T(f(X),f(Y),sss(Z),W)\to T(X,Y,ss(Z),W)$	$(f\in\mathrm{FUNC}_{TRS})$	(deg.≥ 3)
ii)	$T(f(X_1,\ldots,X_n),f(Y_1,\ldots,Y_n),ss(0),W)\to W$	$(f\in\mathrm{FUNC}_{TRS})$	(deg.$=2$)
iii)	$T(X,Y,s(0),W)\to W$		(deg.$=1$)
iv)	$T(X,X,0,W)\to W$		(deg.$=0$)

(all these T-rules are included in the built-in: rules in i) and ii), albeit in infinite number, are manageable by the transform since they are needed only when the corresponding functions of $\textsc{Func}_{TRS}$ are present, and so are treated analogously to the propagation and transition rules).

Note that the output eventually obtained after the T-test has been passed is just $\mathcal{N}\langle \bar{t}_0 \rangle$ and not the full description $\mathcal{N}\langle \bar{t}_0, s_{n+1}, 0\rangle$ because this extra information is utterly superfluous (variables in s_{n+1} are already contained in $\bar{t}_0$).

2. $\mathrm{Var}(s_{n+1}) \not\subseteq \mathrm{Var}(\bar{t}_0)$
Two cases are possible: whether the degree of a flattening expression unifying with s_{n+1} is *bounded* or *unbounded*:

A) $k = \max\{||t|| : t \text{ is an } FE, t\,\mathcal{U} s_{n+1}\} < \infty$ (bounded degree)
Then the transform produces $k+1$ rewrite rules:

$$p\mathcal{N}\langle \bar{t}_0, \mathit{Trunc}_i(s_{n+1}), \underbrace{s\cdots s}_{i}(0)\rangle \rightarrow \begin{array}{c}\text{Transition functions}\\ \text{likewise }\mathsf{Tswm}\end{array} \mathcal{N}\langle \bar{t}_0\rangle \quad (0 \le i \le k)$$

(where $\mathit{Trunc}_i(s_{n+1})$ is required to be disjoint with $\bar{t}_0$). Again, note that just $\mathcal{N}\langle \bar{t}_0\rangle$ is used and not a full description of the output because, for properties of the FWM class, no references to variables in s_{n+1} are possible in this case.

B) $\max\{||t|| : t \text{ is an } FE, t\,\mathcal{U}\, s_{n+1}\} = \infty$ (unbounded degree)
This case can only happen when s_{n+1} is a steady term or a variable, so let $k = ||s_{n+1}||$.
The transform produces $k+2$ rewrite rules.

The first $k+1$ are, analogously to the previous case A,

$$p\mathcal{N}\langle \bar{t}_0, \mathit{Trunc}_i(s_{n+1}), \underbrace{s\cdots s}_{i}(0)\rangle \rightarrow \begin{array}{c}\text{Transition functions}\\ \text{likewise }\mathsf{Tswm}\end{array} \mathcal{N}\langle \bar{t}_0, s'_{n+1}, \underbrace{s\cdots s}_{k}(0)\rangle$$

(with $0 \le i \le k$, $\bar{t}_0$ and $\mathit{Trunc}_i(s_{n+1})$ disjoint), where s'_{n+1} is a ground instance of s_{n+1}: this is done only to ensure the above rules satisfy the condition of a rewrite rule to have all of the variables in the right hand side contained in the left hand side. For instance, if $s_{n+1} = f(g(X,Y))$ and $\mathit{Trunc}_2(s_{n+1}) = f(X)$, to put s_{n+1} in the rhs we would need to introduce another variable (Y) in the lhs.
Acting this way is safe for we are only interested in $\mathit{Trunc}_k(s_{n+1})$ and not in s_{n+1} itself, and so we can pass to s'_{n+1} since $\mathit{Trunc}_k(s_{n+1}) = \mathit{Trunc}_k(s'_{n+1})$ (up to a renaming).

The $k+2$-th rule corresponds to the infinite cases in which the degree of the output part in the goal atom is greater than k: the key fact is just that we can parametrize in a single rewrite rule an otherwise infinite sequence of cases. The rule is

$$p\mathcal{N}\langle \bar{t}_0, s_{n+1}, \underbrace{s\cdots s}_{k+1}(Z)\rangle \rightarrow \begin{array}{c}\text{Transition functions}\\ \text{likewise }\mathsf{Tswm}\end{array} \mathcal{N}\langle \bar{t}_0, s_{n+1}, \underbrace{s\cdots s}_{k+1}(Z)\rangle$$

□

The main result is

Theorem 7.4 *The map* Tfwm *is a complete (compositional) transform for FWM.*

The proof uses the same techniques employed in proving the analogous Theorem 5.7 for Tswm.

7.1 An example

Consider the following logic program after [GW92, AM93]:

```
p(X,g(X)) ←
p(X,f(Y)) ← p(X,g(Y))
```

with moding $\mathtt{p}(\mathrm{in},\mathrm{out})$. This program, albeit very simple, is completely *out of scope* from all the other transforms presented in the literature ([SRK90, RKS92, AM93]) and even the one in [GW92] transforming logic programs into the more powerful conditional TRSs. The problem, as already noticed in [GW92], is that all these transforms consider only the input arguments (which in this example are fixed), and not the output ones. Instead, since the above program belongs to FWM, its transform via Tfwm gives

$$
\begin{array}{l}
p\mathcal{N}\langle X,Y,Z\rangle \to \underset{\langle g\pi_1\bullet\rangle}{\mathcal{T}} T(g(X),Y,Z,\mathcal{N}\langle X\rangle) \\
p\mathcal{N}\langle X,f(Y),0\rangle \to \underset{\langle fg^{-1}\pi_1\bullet\rangle}{\mathcal{T}} \underset{p\langle\pi_1\bullet,gf^{-1}\pi_2\bullet,\pi_3\bullet\rangle}{\mathcal{T}} \mathcal{N}\langle X,f(Y),0\rangle \\
p\mathcal{N}\langle X,Y,s(0)\rangle \to \underset{\langle fg^{-1}\pi_1\bullet\rangle}{\mathcal{T}} \underset{p\langle\pi_1\bullet,g\pi_2\bullet,\pi_3\bullet\rangle}{\mathcal{T}} \mathcal{N}\langle X,g(c),ss(0)\rangle \\
p\mathcal{N}\langle X,f(Y),ss(0)\rangle \to \underset{\langle fg^{-1}\pi_1\bullet\rangle}{\mathcal{T}} \underset{p\langle\pi_1\bullet,g\pi_2\bullet,\pi_3\bullet\rangle}{\mathcal{T}} \mathcal{N}\langle X,g(c),ss(0)\rangle \\
p\mathcal{N}\langle X,f(Y),sss(Z)\rangle \to \underset{\langle fg^{-1}\pi_1\bullet\rangle}{\mathcal{T}} \underset{p\langle\pi_1\bullet,gf^{-1}\pi_2\bullet,\pi_3\bullet\rangle}{\mathcal{T}} \mathcal{N}\langle X,f(Y),sss(Z)\rangle
\end{array}
$$

that is a *terminating* TRS, whose termination can be proved for every goal belonging to FWM (these include, e.g., `←p(a ground term,X)`, and all the goals of the form `←p(a ground term,a flattening expression)`).

The transform Tfwm enhanced with the 'normalizing' technique of Subsection 6.1 gives finally:

1. $p\mathcal{N}\langle X,Y,Z\rangle \to \langle g\pi_1 T(g(X),Y,Z,\mathcal{N}\langle X\rangle)\rangle$
2. $p\mathcal{N}\langle X,f(Y),0\rangle \to \langle fg^{-1}\pi_1 p\mathcal{N}\langle X,g(Y),0\rangle\rangle$
3. $p\mathcal{N}\langle X,Y,s(0)\rangle \to \langle fg^{-1}\pi_1 p\mathcal{N}\langle X,g(c),ss(0)\rangle\rangle$
4. $p\mathcal{N}\langle X,f(Y),ss(0)\rangle \to \langle fg^{-1}\pi_1 p\mathcal{N}\langle X,g(c),ss(0)\rangle\rangle$
5. $p\mathcal{N}\langle X,f(Y),sss(Z)\rangle \to \langle fg^{-1}\pi_1 p\mathcal{N}\langle X,g(Y),sss(Z)\rangle\rangle$

Now we illustrate some reductions in the TRS, to clarify how the mimicking of the logic program is performed:

$$\begin{tikzcd}
\leftarrow \mathtt{p(a,X)} \arrow[r, mapsto] & \langle p\mathcal{N}\langle a,X,s(0)\rangle\rangle \arrow[r, "1"'] \arrow[d, "3"] & \langle\langle g\pi_1 T(g(a),X,s(0),\mathcal{N}\langle a\rangle)\rangle\rangle \arrow[d, "\mathcal{B}"] \\
& \langle\langle fg^{-1}\pi_1 p\mathcal{N}\langle a,g(c),ss(0)\rangle\rangle\rangle \arrow[d, "\mathcal{B}"] & \langle\langle g\pi_1\mathcal{N}\langle a\rangle\rangle\rangle \arrow[d, "\mathcal{B}"] \\
& \langle\langle fg^{-1}\pi_1\langle g\pi_1 T(g(a),g(c),ss(0),\mathcal{N}\langle a\rangle)\rangle\rangle\rangle \arrow[d, "\mathcal{B}"] & \boxed{\mathcal{N}\langle\langle g(a)\rangle\rangle} \\
& \boxed{\mathcal{N}\langle\langle f(a)\rangle\rangle} &
\end{tikzcd}$$

$$\begin{tikzcd}
\leftarrow \mathtt{p(a,f(X))} \arrow[r, mapsto] & \langle p\mathcal{N}\langle a,f(X),ss(0)\rangle\rangle \arrow[r, "4"'] \arrow[d, "1"] & \langle\langle fg^{-1}\pi_1 p\mathcal{N}\langle a,g(c),ss(0)\rangle\rangle\rangle \arrow[d, "1"] \\
& \langle\langle g\pi_1 T(g(a),f(X),ss(0),\mathcal{N}\langle a\rangle)\rangle\rangle & \langle\langle fg^{-1}\pi_1\langle g\pi_1 T(g(a),g(c),ss(0),\mathcal{N}\langle a\rangle)\rangle\rangle\rangle \arrow[d, "\mathcal{B}"] \\
& & \langle\langle fg^{-1}\pi_1\langle g\pi_1\mathcal{N}\langle a\rangle\rangle\rangle\rangle \arrow[d, "\mathcal{B}"] \\
& & \boxed{\mathcal{N}\langle\langle f(a)\rangle\rangle}
\end{tikzcd}$$

$$\begin{tikzcd}
\leftarrow \mathtt{p(a,f(a))} \arrow[r, mapsto] & \langle p\mathcal{N}\langle a,f(a),0\rangle\rangle \arrow[r, "2"'] \arrow[d, "1"] & \langle\langle fg^{-1}\pi_1 p\mathcal{N}\langle a,g(a),0\rangle\rangle\rangle \arrow[d, "1"] \\
& \langle\langle g\pi_1 T(g(a),f(a),0,\mathcal{N}\langle a\rangle)\rangle\rangle & \langle\langle fg^{-1}\pi_1\langle g\pi_1 T(g(a),g(a),0,\mathcal{N}\langle a\rangle)\rangle\rangle\rangle \arrow[d, "\mathcal{B}"] \\
& & \langle\langle fg^{-1}\pi_1\langle g\pi_1\mathcal{N}\langle a\rangle\rangle\rangle\rangle \arrow[d, "\mathcal{B}"] \\
& & \boxed{\mathcal{N}\langle\langle f(a)\rangle\rangle}
\end{tikzcd}$$

$$\begin{tikzcd}
\leftarrow \mathtt{p(a,fh(X,Y))} \arrow[r, mapsto] & \langle p\mathcal{N}\langle a,fh(X,Y),sss(0)\rangle\rangle \arrow[r, "5"'] & \langle\langle fg^{-1}\pi_1 p\mathcal{N}\langle a,gh(X,Y),sss(0)\rangle\rangle\rangle \arrow[d, "1"] \\
& & \langle\langle fg^{-1}\pi_1\langle g\pi_1 T(g(a),gh(X,Y),sss(0),\mathcal{N}\langle a\rangle)\rangle\rangle\rangle \arrow[d, "\mathcal{B}"] \\
& & \langle\langle fg^{-1}\pi_1\langle g\pi_1 T(a,h(X,Y),ss(0),\mathcal{N}\langle a\rangle)\rangle\rangle\rangle
\end{tikzcd}$$

$$\begin{tikzcd}
\leftarrow \mathtt{p(a,fh(a,g(a)))} \arrow[r, mapsto] & \langle p\mathcal{N}\langle a,fh(a,g(a)),0\rangle\rangle \arrow[r, "1"'] \arrow[d, "2"] & \langle\langle g\pi_1 T(g(a),fh(a,g(a)),0,\mathcal{N}\langle a\rangle)\rangle\rangle \\
& \langle\langle fg^{-1}\pi_1 p\mathcal{N}\langle a,gh(a,g(a)),0\rangle\rangle\rangle &
\end{tikzcd}$$

8 Remarks

– We mentioned in Subsection 3.1, among the advantages of a compositional transform, that in case of modifications of a program P adding new rules to obtain a new program P', only the transform of the difference program has to be calculated.

The general case, in which from P we pass to an arbitrary P', is readily much more worthwhile in case of practical applications. It turns out that, for the transform here developed, this case can be coped satisfactorily as before.

For two TRSs T_1 and T_2, define $T_1 \setminus_{\mathcal{M}} T_2$ as the TRS $\{r | (r \in T_1 \setminus T_2) \vee (r \in T_1 \cap \mathcal{B})\}$, that is we cut from T_1 only its main rules. Then it holds (τ stands for Tswm or Tfwm)

$$\tau(P') = \tau(P \setminus (P \setminus P') \cup (P' \setminus P)) = \tau(P) \setminus_{\mathcal{M}} \tau(P \setminus P') \cup \tau(P' \setminus P)$$

and since $\tau(P)$ was already calculated, only $\tau(P \setminus P')$ and $\tau(P' \setminus P)$ remain; hence we can generalize the case treated in Subsection 3.1 saying that when from P we pass to P', to calculate the transform of P' once given the one of P only the transforms of the *difference programs* need to be calculated.

– In developing the transforms of this paper we paid attention to completeness only. Nevertheless, semi-completeness is important in deriving sufficient criterions for termination. It turns out that we can easily extend our transforms to make them semi-complete for the broader class of Well Moded programs.

The problem is that the result of Lemma 5.8 does not hold any more, since only the input arguments discriminate the application of a rule to a term. To cope with this, we introduce *conditional operators* defined via the rules

$$\mathcal{C}_t(t) \to t \quad (t \in \text{TERM}_{TRS})$$

that check whether a given argument has a certain structure (provided by t): if it has not, the computation stays blocked (that is $\mathcal{N}$ cannot 'propagate').

The modification of the transform Tswm is simply to put

$$rhs := \underset{\text{SE}_{\langle V, \mathcal{C}_{\mathcal{N}\langle \bar{t}_i \rangle} p_i \langle \bar{s}_i \rangle \rangle} \langle V \rangle}{\boldsymbol{\mathcal{T}}} (rhs) \quad \text{in place of} \quad rhs := \underset{\text{SE}_{\langle V, p_i \langle \bar{s}_i \rangle \rangle} \langle V \rangle}{\boldsymbol{\mathcal{T}}} (rhs)$$

viz. every time $p_i \langle \bar{s}_i \rangle$ is computed it is checked whether the result is compatible with the term structure of $\bar{t}_i$. The same technique is used for Tfwm.

We gain only semi-completeness and not completeness because the check is performed only after the computation of $p_i(\bar{s}_i)$ is finished, and therefore new infinite computations could be added, destroying soundness.

– So far, all the proposals of transformation from logic programs to TRSs have been staying always within well moded programs. However, Transform Tswm has been built in such a way to support a broader class, that is the class of *Simply Well Typed* (SWT) programs introduced in [AE93]: this is the generalization of the SWM class to types, much like the class of Well Typed programs introduced in [BLR92] is for Well Moded programs; thanks to the introduction of types, it allows *non-ground inputs*, thus giving much more flexibility than the SWM class. Using almost the same proof used for the SWM case, it can be shown that Tswm is *complete* for SWT as well.

References

[AE93] K.R. Apt and S. Etalle. On the unification free Prolog programs. In S. Sokołowski, editor, *MFCS'93*, LNCS, pp. 1–19. Springer, 1993.

[AM93] G. Aguzzi and U. Modigliani. Proving termination of logic programs by transforming them into equivalent term rewriting systems. *FST&TCS'93*, LNCS. Springer, 1993.

[Apt90] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10, pp. 495–574. Elsevier – MIT Press, 1990.

[BLR92] F. Bronsard, T.K. Lakshman, and U.S Reddy. A framework of directionality for proving termination of logic programs. In K.R. Apt, editor, *JICSLP'92*, pp. 321–335. MIT Press, 1992.

[Der87] N. Dershowitz. Termination of rewriting. *JSC*, 3:69–116, 1987.

[DJ90] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pp. 243–320. Elsevier – MIT Press, 1990.

[DM85] P. Dembinski and J. Małuszyński. AND-parallelism with intelligent backtracking for annotated logic programs. *ILPS'85*, pp. 29–38, 1985.

[GW92] H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In M. Rusinowitch and J.L Rémy, editors, *CTRS'92*, pp. 216–222, July 1992.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pp. 1–116. Clarendon Press, Oxford, 1992.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, 2nd ed., 1987.

[Mar93] M. Marchiori. Logic programming, matching and term rewriting systems. Master's thesis, Dep. of Pure and Applied Mathematics, University of Padova, Italy, July 1993. In Italian.

[Mar94] M. Marchiori. Localizations of unification freedom through matching directions. Submitted, March 1994.

[MK85] J. Małuszyński and H.J. Komorowski. Unification-free execution of logic programs. *IEEE Symposium on Logic Programming*, pp. 78–86, 1985.

[RKS92] K. Rao, D. Kapur, and R.K. Shyamasundar. A transformational methodology for proving termination of logic programs. In *CSL'92*, volume 626 of *LNCS*, pp. 213–226, Berlin, 1992. Springer.

[SRK90] R.K. Shyamasundar, K. Rao, and D. Kapur. Termination of logic programs. Computer science group, Tata Institute of Fundamental Research, Bombay, India, 1990. Revised version: Rewriting concepts in the study of termination of logic programs, *ALPUK*, 1992.