Proving Properties of Term Rewrite Systems via Logic Programs

Sébastien Limet[1] and Gernot Salzer[2]

[1] Université d'Orléans, France
limet@lifo.univ-orleans.fr
[2] Technische Universität Wien, Austria
salzer@logic.at

Abstract. We present a general translation of term rewrite systems (TRS) to logic programs such that basic rewriting derivations become logic deductions. Certain TRS result in so-called cs-programs, which were originally studied in the context of constraint systems and tree tuple languages. By applying decidability and computability results of cs-programs we obtain new classes of TRS that have nice properties like decidability of unification, regular sets of descendants or finite representations of R-unifiers. Our findings generalize former results in the field of term rewriting.

1 Introduction

Term rewrite systems (TRS) are fundamental to fields like theorem proving, system verification, or functional-logic programming. Applications there require decision procedures e.g. for R-unifiability (for terms t and t', is there a substitution σ such that $t\sigma \rightarrow_R^* u \; {}_R^*\!\!\leftarrow t'\sigma$?) or for reachability (is term t' reachable from term t by a rewriting derivation?). Most desired properties depend on the ability to compute $R^*(E)$ (the set of terms reachable from elements in E) by means of tree languages which have a decidable emptiness test and are closed under intersection. Authors like [3, 11, 10] studied classes of TRS which effectively preserve recognizability, i.e. $R^*(E)$ is regular if E is regular.

Recognizability is usually preserved by encoding TRS derivations as tree automata and by exploiting the properties of the class of TRS under consideration. Another method encodes first the rewriting relation by means of tree tuple languages [4, 8], i.e. it computes a tree tuple language $\{ (t, t') \mid t \rightarrow_R^* t' \}$, and then obtains recognizability by projection.

This paper applies the second method using a restricted class of logic programs to handle tree tuple languages. We first define a translation of rewrite systems to logic programs that maps a restricted form of rewriting, called basic rewriting, to deductions. It preserves essential structural properties such that classes of TRS correspond naturally to certain classes of logic programs, allowing to transfer results between the two formalisms.

We restrict our attention to basic rewriting, i.e., to rewriting that never modifies parts of terms considered as data. This fits the logic programming paradigm where there is a clear distinction between data (function symbols) and operations on them (predicate symbols). This allows to obtain logic programs that preserve the structure of rewrite systems well. Note that basic rewriting coincides with rewriting for large classes of term rewrite systems such as right-linear ones. Right-linearity is required for most classes that preserve recognizability.

The classes of TRS we study in this paper are defined via the notion of *possible redexes* (i.e. parts of terms that may be rewritten). This notion generalizes constructor-based TRS where some symbols, called constructors, cannot be rewritten. The required properties are right-linearity and no nesting of possible redexes. The two classes differ in the form of possible redexes in the right-hand-sides. The first class, called quasi-cs TRS, requires that the depth of a variable under a possible redex is less than or equal to the depth in the right-hand-side. The second one, called instance-based TRS, requires that possible redexes are less instantiated than left-hand-sides. Instance-based TRS extend the class of TRS described in [4] and [10], and quasi-cs TRS is a new class not studied before. As in [10] we require that each term of the input language E contains a bounded number of basic positions. Instance-based TRS extend also the class of layered transducing TRS of [11]. The resulting logic program has the additional property that the projection to the last argument of each predicate is a regular language. This implies that recognizability for the basic rewriting relation is preserved.

The main interest of the framework presented in this paper compared to previous work is the replacement of very technical ad-hoc encodings of rewriting relations as tree automata by a very general translation to a high-level language like logic programming. The preservation of recognizability then reduces to a termination proof of the algorithm presented in [5] that transforms logic programs to so-called cs-programs.

Section 2 recalls some basic definitions of TRS, Section 3 describes cs-programs and their properties, and Section 4 presents the translation of basic rewriting to logic programming. Section 5 uses this translation to define two classes of TRS that preserve recognizability. The last section gives an outlook on future work. Due to space restrictions not all proofs are included in the paper; they can be found in [6].

2 Preliminaries

We recall some basic notions and notations concerning term rewrite systems; for details see [1].

Let Σ be a finite set of symbols with arity, Var be an infinite set of variables, and $\mathcal{T}(\Sigma, Var)$ be the first-order term algebra over Σ and Var. Σ consists of two disjoint subsets: the set $\mathcal{F}$ of defined function symbols (or function symbols), and the set $\mathcal{C}$ of constructor symbols. The terms of $\mathcal{T}(\mathcal{C}, Var)$ are called data-terms. A term is linear if no variable occurs more than once in it.

For a term t, $Pos(t)$ denotes the set of positions in t, $|t| = |Pos(t)|$ the size of t, and $t|_u$ the subterm of t at position u. The term $t[u \leftarrow s]$ is obtained from t by replacing the subterm at position u by s. $Var(t)$ is the set of variables occurring in t. The set $\Sigma Pos(t) \subseteq Pos(t)$ denotes the set of non-variable positions, i.e., $t|_u \notin Var$ for $u \in \Sigma Pos(t)$ and $t|_u \in Var$ for $u \in Pos(t) \setminus \Sigma Pos(t)$. A substitution is a mapping from Var to $\mathcal{T}(\Sigma, Var)$, which extends trivially to a mapping from $\mathcal{T}(\Sigma, Var)$ to $\mathcal{T}(\Sigma, Var)$. The domain of a substitution σ, $Dom(\sigma)$, is the set $\{ x \in Var \mid x\sigma \neq \sigma \}$. For $V \subseteq Var$, $\sigma|_V$ denotes the restriction of σ to the variables in V, i.e., $x\sigma|_V = x\sigma$ for $x \in V$ and $x\sigma|_V = x$ otherwise. If term t is an instance of term s, i.e. $t = s\sigma$, we say that t matches s and s subsumes t.

Let $CVar = \{ \square_i \mid i \geq 1 \}$ be the set of context variables distinct from Var, where a context is a term in $T(\Sigma, Var \cup CVar)$ such that the i^{th} occurrence of a context variable counted from left to right is labelled by $\square_i$. $\square_1$ (also denoted $\square$) is called the trivial context. A context is called n-context if it contains n context variables. For an n-context C, the expression $C[t_1, \ldots, t_n]$ denotes the term $\{ \square_i \mapsto t_i \mid 1 \leq i \leq n \}C$.

A term rewrite system (TRS) is a finite set of oriented equations called rewrite rules. Lhs and rhs are shorthand for the left-hand and right-hand side of a rule, respectively. For a TRS R, the rewrite relation is denoted by $\rightarrow_R$ and is defined by $t \rightarrow_R s$ if there exists a rule $l \rightarrow r$ in R, a non-variable position u in t, and a substitution σ, such that $t|_u = l\sigma$ and $s = t[u \leftarrow r\sigma]$. Such a step is written as $t \rightarrow_{[u, l \rightarrow r, \sigma]} s$. If a term t cannot be reduced by any rewriting rule, it is said to be irreducible. The reflexive-transitive closure of $\rightarrow_R$ is denoted by $\rightarrow_R^*$, and the symmetric closure of $\rightarrow_R^*$ by $=_R$. The relation $\rightarrow_R^n$ denotes n steps of the rewrite relation. By $t \downarrow_R s$ we denote the derivation $t \rightarrow_R^* s$ such that s is irreducible. For a set of terms, E, we define $R^*(E) = \{ t \mid t' \rightarrow_R^* t \text{ for some } t' \in E \}$ and $R^{\downarrow}(E) = \{ t \mid t' \downarrow_R t \text{ for some } t' \in E \}$. If the lhs (rhs) of every rule is linear the TRS is said to be left-(right-)linear. If it is both left- and right-linear the TRS is called linear. A TRS is constructor based if every rule is of the form $f(t_1, \ldots, t_n) \rightarrow r$ where all t_i's are data-terms.

3 Cs-Programs

We use techniques from logic programming to deal with certain types of rewriting relations. Term rewrite systems are transformed to logic programs preserving their characteristic properties. The programs are manipulated by standard folding/unfolding techniques to obtain certain normal forms like *cs-programs*. Results about the latter lead directly to conclusions about rewrite systems. Therefore this section presents some results about logic programs. While some notions and theorems are just quoted from [5] (and in fact appear in a similar form already in [9]), other results are new. We presuppose basic knowledge about logic programming (see e.g. [7]).

Definition 1. *A program clause is a* cs-clause *if its body is linear and contains no function symbols, i.e., if all arguments of the body atoms are variables occurring nowhere else in the body. A cs-clause is linear if the head atom is linear. A logic program is a (linear)* cs-program *iff all its clauses are (linear) cs-clauses.*

Every logic program $\mathcal{P}$ can be transformed to an equivalent cs-program by applying two rules, unfolding and definition introduction[1]. The rules transform states $\langle \mathcal{P}, \mathcal{D}_{\text{new}}, \mathcal{D}_{\text{done}}, \mathcal{C}_{\text{new}}, \mathcal{C}_{\text{out}} \rangle$ where $\mathcal{D}_{\text{new}}$ are definitions not yet unfolded, $\mathcal{D}_{\text{done}}$ are definitions already processed but still used for simplifying clauses, $\mathcal{C}_{\text{new}}$ are clauses generated from definitions by unfolding, and $\mathcal{C}_{\text{out}}$ is the cs-program generated so far. Syntactically, definitions are written as clauses, but from the semantic point of view they are equivalences. We require the head of a definition to contain all variables occurring in the body[2]. A set of definitions, $\mathcal{D}$, is *compatible with* $\mathcal{P}$, if all predicate symbols occurring in the heads of the definitions occur just there and nowhere else in $\mathcal{D}$ and $\mathcal{P}$; the only exception are tautological definitions of the form $P(\vec{x}) \leftarrow P(\vec{x})$ where P may occur without restrictions throughout $\mathcal{D}$ and $\mathcal{P}$. The predicate symbols in the heads of $\mathcal{D}$ are called the *predicates defined by* $\mathcal{D}$.

We write $S \Rightarrow S'$ if S' is a state obtained from state S by applying one of the rules *unfolding* or *definition introduction* defined below. The reflexive and transitive closure of $\Rightarrow$ is denoted by $\overset{*}{\Rightarrow}$. An *initial state* is of the form $\langle \mathcal{P}, \mathcal{D}, \emptyset, \emptyset, \emptyset \rangle$ where $\mathcal{D}$ is compatible with $\mathcal{P}$. A *final state* is of the form $\langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$. $\mathcal{P}$ and $\mathcal{D}$ are called the input of a derivation, $\mathcal{P}'$ its output. A derivation is *complete* if its last state is final.

Unfolding. Pick a definition not yet processed, select one or more of its body atoms according to some selection rule, and unfold them with all matching clauses from the input program. Formally:

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{new}} \,\dot{\cup}\, \{L \leftarrow \mathcal{R} \dot{\cup} \{A_1, \ldots, A_k\}\},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}} \rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{new}},\, \mathcal{D}_{\text{done}} \cup \{L \leftarrow \mathcal{R} \cup \{A_1, \ldots, A_k\}\},\, \mathcal{C}_{\text{new}} \cup \mathcal{C},\, \mathcal{C}_{\text{out}} \rangle}$$

where $\mathcal{C}$ is the set of all clauses $L \leftarrow (\mathcal{R} \cup \mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k)\mu$ such that $H_i \leftarrow \mathcal{B}_i$ is a clause in $\mathcal{P}$ for $i = 1, \ldots, k$, and such that the simultaneous most general unifier μ of $(A_1, \ldots, A_k)$ and $(H_1, \ldots, H_k)$ exists. Note that the clauses from $\mathcal{P}$ have to be renamed properly such that they share variables neither with each other nor with $L \leftarrow \mathcal{R} \cup \{A_1, \ldots, A_k\}$.

Definition Introduction. Pick a clause not yet processed, decompose its body into minimal variable-disjoint components, and replace every component that is not yet a single linear atom without function symbols by an atom that is either looked up in the set of old definitions, or if this fails is built of a new predicate symbol and the component variables. For every failed lookup introduce a new definition associating the new predicate symbol with the replaced component.

[1] Our version of definition introduction is a combination of definition introduction and folding in the traditional sense of e.g. [13].

[2] We could get rid of the restriction by introducing the notion of *linking variables* like in [9]. Since the restriction is always satisfied in our context we avoid this complication.

Formally:

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\text{new}},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}} \,\dot\cup\, \{H \leftarrow \mathcal{B}_1 \dot\cup \cdots \dot\cup \mathcal{B}_k\},\, \mathcal{C}_{\text{out}} \rangle}{\langle \mathcal{P},\, \mathcal{D}_{\text{new}} \cup \mathcal{D},\, \mathcal{D}_{\text{done}},\, \mathcal{C}_{\text{new}},\, \mathcal{C}_{\text{out}} \cup \{H \leftarrow L_1, \dots, L_k\} \rangle}$$

where $\mathcal{B}_1, \dots, \mathcal{B}_k$ is a maximal decomposition of $\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_k$ into non-empty variable-disjoint subsets,

$$L_i = \begin{cases} L\eta^{-1} & \text{if } L{\leftarrow}\mathcal{B}_i\eta \in \mathcal{D}_{\text{done}} \text{ for some var. renaming } \eta \\ P_i(x_1, \dots, x_n) & \text{otherwise, } \{x_1, \dots, x_n\} \text{ being the vars. of } \mathcal{B}_i. \end{cases}$$

for $1 \leq i \leq k$ and new predicate symbols P_i, and where $\mathcal{D}$ is the set of all $L_i{\leftarrow}\mathcal{B}_i$ such that L_i contains a new predicate symbol[3].

Theorem 1. *Let $\mathcal{P}$ be a logic program and $\mathcal{D}$ be a set of definitions compatible with $\mathcal{P}$. If $\langle \mathcal{P}, \mathcal{D}, \emptyset, \emptyset, \emptyset \rangle \overset{*}{\Rightarrow} \langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$, then $\mathcal{P}'$ is a cs-program whose least Herbrand model semantics coincides with the one of $\mathcal{P}$ for all predicates defined by $\mathcal{D}$.*

In the following we discuss properties of logic programs that can be transformed to finite cs-programs.

Definition 2. *A clause is* quasi-cs*, if the body is linear and for every variable that occurs both in the body and the head, the depth of its occurrence in the body is smaller than or equal to the depth of all occurrences in the head. A program is quasi-cs if all its clauses are.*

Theorem 2. *Let $\mathcal{P}$ be a quasi-cs program, and let $\mathcal{D}_{\mathcal{P}}$ be the set of all tautologies $P(\vec{x}){\leftarrow}P(\vec{x})$ such that P occurs in $\mathcal{P}$. Every $\Rightarrow$-derivation with input $\mathcal{P}$ and $\mathcal{D}_{\mathcal{P}}$ is finite.*

Another, new class of programs that also have finite cs-counterparts are instance-based programs.

Definition 3. *A clause $H \leftarrow \mathcal{B}$ of a logic program $\mathcal{P}$ is* instance-based *(is an ib-clause) if $\mathcal{B}$ is linear and for every pair of atoms H', A where H' is the head of a clause of $\mathcal{P}$ properly renamed and A is an atom in $\mathcal{B}$ such that H' and A unify, H' is an instance of A. A program is instance-based (is an ib-program) if it consists of ib-clauses.*

Theorem 3. *Let $\mathcal{P}$ be an ib-program, and let $\mathcal{D}_{\mathcal{P}}$ be the set of all tautologies $P(\vec{x}){\leftarrow}P(\vec{x})$ such that P occurs in $\mathcal{P}$. Every $\Rightarrow$-derivation with input $\mathcal{P}$ and $\mathcal{D}_{\mathcal{P}}$ is finite.*

Proof. We show that for ib-clauses all definitions occurring in a derivation are of the form $L \leftarrow A$, where A is a single linear atom which subsumes any head clause it unifies with. This implies that only finitely many definitions are generated up

[3] A substitution η is a variable renaming for a set of atoms $\mathcal{R}$, if there exists a substitution η^{-1} such that $\mathcal{R}\eta\eta^{-1} = \mathcal{R}$.

to variable renaming, i.e., complete derivations are finite. Note that the initial definitions in $\mathcal{D}_{\mathcal{P}}$ are of this particular form.

Let $H \leftarrow \mathcal{B}$ be a ib-clause of $\mathcal{P}$, and let $\mu = \text{mgu}(A, H)$ be the most general unifier involved in unfolding a definition $L \leftarrow A$. We may divide μ into two parts, μ_A and μ_H, such that $A\mu_A = H\mu_H$. Since A is linear, $x\mu_H$ is a linear term for all x, and the variables in $x\mu_H$ do not occur in $\mathcal{B}$ (definitions and clauses are renamed apart prior to unfolding). Therefore $\mathcal{B}\mu_H$, the body of the new clause $(L \leftarrow \mathcal{B})\mu = L\mu_A \leftarrow \mathcal{B}\mu_H$, consists of linear atoms not sharing any variables with each other. Hence a maximal decomposition of $\mathcal{B}$ consists of singletons only.

It remains to show that each atom of $\mathcal{B}$ and therefore of the new definitions subsumes all the head clauses it unifies with. A subsumes H therefore $H\mu = H$ which means that $\mathcal{B}\mu = \mathcal{B}$ and therefore atoms of $\mathcal{B}$ satisfies ib property, so it subsumes all the head clauses it unifies with. □

A logic program is called monadic if it contains only unary predicate symbols. A unary predicate is monadic if it is defined by a monadic program.

Lemma 1. *The set of ground terms S is a regular tree language iff there exists a linear monadic cs-predicate P such that $P(t)$ is true for all terms t in S.*

Definition 4. *Let P be a predicate defined by a program $\mathcal{P}$. The i^{th} argument of P is said to be* free *iff for every horn clause $P(t_1, \dots, t_n) \leftarrow A_1, \dots, A_k$ of $\mathcal{P}$, t_i is linear, $\forall x \in Var(t_i)$, x appears once in the head and once in the body and $\forall j \in 1, k$, either $Var(t_i) \cap Var(A_j) = \emptyset$ or $Var(t_i) \cap Var(A_j)$ is a single variable which occurs in a free argument of A_j.*

Lemma 2. *Let P be a predicate which i^{th} argument is free, of a cs-program $\mathcal{P}$. The set $\{\, t \mid \mathcal{P} \models P(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n) \,\}$ is defined by the set of clauses constructed as follows:*

- *$C_0 = \{\}$, $\mathcal{P}_0 = \{P'\}$*
- *$C_i = \{Q'(s) \leftarrow \{R'(s') | Q' \in \mathcal{P}_{i-1}, Q(\dots, s, \dots) \leftarrow \mathcal{B} \in \mathcal{P}$ and $\mathcal{B}$ has at least one model, $R(\dots, s', \dots) \in \mathcal{B}$ and $Var(s) \cap Var(s') \neq \emptyset\}$,*
  $\mathcal{P}_i = \{Q'$ occurring in a body of a clause of $C_i\} \setminus \bigcup_{j<i} \mathcal{P}_j\}$

Proof. It is obvious that if $\mathcal{P} \models P(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n)$ then $\mathcal{P}' \models P'(t)$. Now, let us prove that if $\mathcal{P}' \models P'(t)$ then there is some $t_1, \dots, t_n$ such that $\mathcal{P} \models P(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n)$. It is done by induction on the height of the proof tree.

- $h = 0$ obvious by construction
- $h > 0$ Let $P'(s) \leftarrow \mathcal{B}'$ be the top clause of the proof tree $P(\dots, s, \dots) \leftarrow \mathcal{B}$ the corresponding clause of $\mathcal{P}$ and $\sigma = \text{mgu}(t, s)$. By construction, we have $\mathcal{P} \models \mathcal{B}\mu$, this means that $\forall Q(\vec{t})$ s.t. $Var(Q(\vec{t})) \cap Var(s) = \emptyset$, we have $\mathcal{P} \models Q(\vec{t})\mu$. $\forall Q'(s')\sigma \in \mathcal{B}'\sigma$, $\mathcal{P}' \models Q'(s')\sigma$. Since s appears at a free position of P, s' occurs at a free position of Q, so by induction hypothesis $\mathcal{P} \models Q(\dots, s', \dots)\sigma'$ with $s'\sigma' = s'\sigma$. Since all atoms of $\mathcal{B}$ are variable disjoint, the substitution $\mu' = \sigma' \uplus \mu|_{\{x \notin Dom(\sigma')\}}$ is such that $Q(\dots, s', \dots)\mu' = Q(\dots, s', \dots)\sigma'$ if $Q'(s') \in \mathcal{B}'$ and $Q(\vec{t})\mu' = Q(\vec{t})\mu$ if $Var(Q(\vec{t})) \cap Var(t) = \emptyset$. So $\mathcal{P} \models \mathcal{B}\mu'$ this implies $\mathcal{P} \models P(\dots, s, \dots)\mu'$ where $s\mu' = t$.

In the following, $Reg(\mathcal{P})$ denotes the set of clauses constructed in Lemma 2.

Definition 5. *Let $\mathcal{P}$ be a cs-program. A clause $P(\vec{x}) \leftarrow P'(\vec{t}) \cup \mathcal{B}_{reg}$ is called a* regular join definition *compatible with $\mathcal{P}$ if*

- *$P(\vec{x})$ is linear and $\vec{x} \subseteq Var(P'(\vec{t})) \cup Var(\mathcal{B}_{reg})$*
- *$P'(\vec{t})$ is linear*
- *the elements of $\mathcal{B}_{reg}$ are linear monadic predicates.*

$P'(\vec{t})$ is called the key atom. If all atoms of $\mathcal{B}_{reg}$ are of depth 0 *the definition is said to be a* flat regular join definition.

The strategy which chooses atoms of maximal depth terminates but does not preserve freeness of arguments. For example, consider the clause

$$A(s(c(x,y_1)), s(c(x,y_2)), c(x_3,y_3)) \leftarrow A(x,y_1,x_3), B(y_2,y_3)$$

where the last arguments of A and B are free. When unfolding the regular join definition $P(z_1,z_2,z_3) \leftarrow A(s(z_1), s(z_2), z_3), C(z_1), C(z_2)$ where C is linear monadic, we get

$$P(c(x,y_1), c(x,y_2), c(x_3,y_3)) \leftarrow A(x,y_1,x_3), B(y_2,y_3), C(c(x,y_1)), C(c(x,y_2)).$$

The body of this clause cannot be decomposed, so the new clause in $\mathcal{C}_{\rm new}$ is $P(c(x,y_1), c(x,y_2), c(x_3,y_3)) \leftarrow Q(x,y_1,y_2,x_3,y_3)$. But now the variables x_3 and y_3 are no longer "independent". This could have been avoided if $C(c(x,y_1))$ and $C(c(x,y_2))$ had been unfolded before introducing the new clause in $\mathcal{C}_{\rm new}$. Indeed suppose that $C(c(x',x'')) \leftarrow C(x'), C(x'')$ is in the definition of C, then the next unfolding step yields

$$P(c(x,y_1), c(x,y_2), c(x_3,y_3)) \leftarrow A(x,y_1,x_3), B(y_2,y_3), C(x), C(y_1), C(y_2)$$

where the two free variable are separated into two different components. This observation leads to the following restricted rule that unfolds atoms without introducing clauses in $\mathcal{C}_{\rm new}$. Together with a suitable strategy it preserves freeness of arguments.

Restricted Unfolding.

$$\frac{\langle \mathcal{P},\, \mathcal{D}_{\rm new} \mathbin{\dot\cup} \{L \leftarrow \mathcal{R} \mathbin{\dot\cup} \{A_1,\ldots,A_k\}\},\, \mathcal{D}_{\rm done},\, \mathcal{C}_{\rm new},\, \mathcal{C}_{\rm out}\rangle}{\langle \mathcal{P},\, \mathcal{D}_{\rm new} \cup \mathcal{C},\, \mathcal{D}_{\rm done},\, \mathcal{C}_{\rm new},\, \mathcal{C}_{\rm out}\rangle}$$

Let A be an atom and $\mathcal{P}$ be a logic program. A is said to be *sufficiently instantiated* if it is an instance of all clause heads of $\mathcal{P}$ that unify with it (if there is no such head, the condition is vacuously satisfied). A variable in an atom is said to be not sufficiently instantiated if the variable gets instantiated by the most general unifier of the atom with some clause head.

A freeness preserving strategy for $\Rightarrow$-derivations starting from a regular join definition consists in unfolding atoms of $\mathcal{B}_{reg}$ by Restricted Unfolding and the key atom by general Unfolding. The atom for the unfolding operation is selected in the following way: choose an atom of $\mathcal{B}_{reg}$ with a depth greater than 0; if there is none, choose one whose variables are not sufficiently instantiated; otherwise choose the key atom.

Before giving the result, we make some observation concerning the unification problems occurring in the discussion.

Let t be a term and a variable $x \in \mathit{Var}(t)$, we denote by $\mathit{MinDepth}(x, t)$ the minimal depth at which x occurs in t and by $\mathit{\Delta Depth}(t)$ the maximal depth difference between two occurrences of the same variable in t. Note that for linear, variable disjoint terms s and t unifiable by $\mu = \mathrm{mgu}(s, t)$ we have $\mathit{Depth}(s\mu) \leq \max(\mathit{Depth}(s), \mathit{Depth}(t))$.

Property 1. Let t be a term and s be a linear term such that $\mathit{Var}(t) \cap \mathit{Var}(s) = \emptyset$. Let $\mu = \mathrm{mgu}(s, t)$. Then $x\mu$ is linear and $\mathit{Depth}(x\mu) \leq \mathit{Depth}(s)$ for all $x \in \mathit{Var}(t)$, and $\mathit{Depth}(x\mu) \leq \max(\mathit{Depth}(t), \mathit{Depth}(s) + \mathit{\Delta Depth}(t))$ for all $x \in \mathit{Var}(s)$. Moreover, $\mathit{\Delta Depth}(t\mu) = \mathit{\Delta Depth}(t)$ and the maximal number of occurrences of a variable in $t\mu$ is the same as in t.

Lemma 3. *Any regular join definition can be transform into a flat one using restricted unfolding.*

Theorem 4. *Any $\Rightarrow$-derivation using a freeness preserving strategy starting with a cs-program and a flat regular join definition is terminating.*

Proof. Let τ be the maximal depth of an atom occurring in the whole program and in the first definition. According to the freeness preserving strategy, the key atom K will be unfolded only if all the variables it shares with $\mathcal{B}_{reg}$ are sufficiently instantiated. Since all predicates of $\mathcal{B}_{reg}$ are non-copying monadic, they define each a regular tree language. Let us call intersection problem $\bigcup_{0 \leq i \leq n} C_i[P_1^i(x_1^i), \ldots, P_n^i(x_{n_i}^i)]$ where C_i is a context which depth is inferior to $2 \times \tau$ and P_j^i and predicate symbols and x_j^i are pairwise different variables. The solution of such a problem is $\{t | \forall i \in 0, n, t = C^i[t_1^i, \ldots, t_n^i], \mathcal{P} \models \bigcup_{1 \leq i \leq n_i} P_j^i(t_j)\}$. since $\mathcal{P}$ is finite there is only a finite number of intersection problems. For a unification problem UP $\mathit{MinSol} = \mathit{Min}(\{\mathit{Depth}(t) | t$ solution of $UP\})$. Let us call h the maximum of all the MinSol.

We prove that $\mathit{Depth}(K)$ is less or equal to h. We first have to remark that all subterms of K which occurs at a depth greater than τ, has been generated by one or more non-copying monadic predicates. A variable x of K is not sufficiently instantiated if either it occurs at an occurrence of K which is also an occurrence of one clause head it unifies with or it occurs at an occurrence greater than one duplicated variable of one clause head. The first case can be solved by unfolding this atom and its "descendants" to make this branch growing. The depth of the result is less than $2 \times \tau$. For the second case let $PB_{H,x} = \{C_i[\vec{t_i}] = K|_u | H|_u = x\}$ where $\mathit{Depth}(C_i) \leq 2 \times \tau$ and each terms of the $\vec{t_i}$ have been generated by one non-copying monadic predicate. $PB_{H,x}$ is an intersection problem, therefore its minimal solution has a depth inferior or equal to h. This means that unfolding atoms involved in this problem leads to an instance of depth inferior to $\tau + h$ (x is a variable of H so it occurs at a depth inferior to τ).

Let $H' \leftarrow \mathcal{B}'$ the clause used for unfolding the key atom and $\mu = \mathrm{mgu}(K, H')$. Since all variables shared by the key atom are sufficiently instantiated $x\mu = x$ for $x \in \mathit{Var}(\mathcal{B}_{reg})$ so $\mathcal{B}_{reg}\mu = B'_{reg}$ and is of depth 0. The variables of H' are instantiated by terms of depth less than h, which means that $\mathcal{B}'\mu$ is of depth less than h. Since K is linear $x\mu$ is linear for $x \in Var(H')$ and so $\mathcal{B}'\mu$ is linear. Finally since since atoms of $\mathcal{B}_{reg}$ are unary and of depth 0 they share variables with at most one argument of atoms of $\mathcal{B}'\mu$. The new maximal decomposition of $\mathcal{B}_{reg}\mathcal{B}'\mu$ in variable disjoint components contains at most one atom of $\mathcal{B}'\mu$ so they are all body of regular join definition compatible with $\mathcal{P}$.

Since the depth of the key atom is bounded and linear, number of body of regular join definition compatible with $\mathcal{P}$ is bounded so the algorithm terminates.

Theorem 5. *Let $P(\vec{x}, x) \leftarrow P'(\vec{t}, x), \mathcal{B}_{reg}$ be a regular join definition, such that x occurs at a free argument of P' and $x \notin Var(\mathcal{B}_{reg})$. The argument where x occurs in P is a free argument of P in the resulting cs-program.*

Proof. Since x occurs once in the body it will be instantiated only when unfolding the key atom and in this case each variable of the instance will occur in a different key atom at free argument. So all clauses introduced in $\mathcal{C}_{\text{new}}$ are of the form $P(\vec{t}, t) \leftarrow P_1(\vec{x_1}, x_1), \ldots, P_n(\vec{x_n}, x_n)$ where $x_1, \ldots, x_n$ are variables of t and t do not share variables with $\vec{t}$. So t and the x_is occur at a free argument.

4 Encoding Basic Rewriting by Logic Programs

In this section we present the way we encode rewriting relation by a logic program. This translation intends to obtain logic programs for which it is possible to deduce recognizability preservation, namely cs-programs for which one argument is free. The translation presented here works for any TRS but does not lead to a cs-program in general.

One of the main differences between term rewriting and logic programming formalisms is the clear distinction in logic programming between the predicate symbols and the function symbols i.e. between the data and the operations applied on them. This distinction is usually not made in term rewriting system. For example, considering the TRS $f(x) \to g(x, x), a \to b, a \to c, g(b, c) \to c$ and the term $f(a)$, we have the following derivation $f(a) \to g(a, a) \to g(a, b) \to c$. In the first rewriting step a is considered as a "data" of f and in the second and third steps a is an "operation". In pure logic programming such a symbol which is sometimes a predicate symbol and sometimes a function symbol does not exist. Since our aim is to prove TRS properties using cs-programs, we intend to encode the TRS derivations by a logic program which is as close as possible to the original TRS. This is why we define a transformation procedure which tends to preserve the structure of the terms. The price to pay is to encode only a restricted form of rewriting relation which fits well to logic programming formalism, namely basic rewriting. Roughly speaking, in basic rewriting it is forbidden to rewrite a subterm which has been considered as data in a former step. Fortunately basic rewriting and rewriting relations coincides for large classes of TRS.

4.1 Basic Rewriting

A rewriting derivation $t_0 \to_R t_1 \ldots \to_R t_n$ is said to be P-basic if the set of basic position of t_0 (denoted $BasPos(t_0)$) is P and for each step $t_i \to_{[u_i, l_i \to r_i, \sigma_i]}$

t_{i+1}, u_i belongs to the set $BasPos(t_i)$ and the set of basic positions of t_{i+1} is $(BasPos(t_i) \setminus \{ v \in Pos(t_i) \mid u_i \leq v \}) \cup \{ u_i.w \mid w \in \Sigma Pos(r_i) \}$. Each step of a basic derivation is denoted $\rightarrow_{bas}$ and we write $t \rightarrow_{bas}^{P*} s$ if t rewrites into s with a P-basic derivation. $\rightarrow_{bas}^{*}$ denotes the relation $\{(t, t') | t \rightarrow_{bas}^{Pos(t)*} t'\}$. For a set of terms E, $R_{bas}^{*}(E) = \{ t \mid \exists t' \in E, t' \rightarrow_{bas}^{*} t \}$. Most of the time P is abusively ommited in the following.

Basic rewriting and rewriting coincides for large classes of TRS, in particular for right-linear TRS.

Lemma 4. *Let R be right-linear TRS. Then $\overset{R}{\rightarrow}^{*}$ and $\overset{R}{\rightarrow}_{bas}^{*}$ are the same relation.*

Note that for the TRS given in introduction of this section, $\overset{R}{\rightarrow}^{*}$ and $\overset{R}{\rightarrow}_{bas}^{*}$ are different since $f(a)$ cannot be rewritten in c with a basic derivation.

The following definitions will be needed further down. They allow to point out positions in a term that may be rewritten. Let R be a TRS and t a term. A position u of t is called a *possible redex position* if $t|_u$ is of the form $C[t_1, \ldots, t_n]$ where C is neither trivial nor a variable, does not contain any possible redex position and C unifies with at least one lhs of R and u_i, the position of t_i, for $1 \leq i \leq n$ is a redex position. C is called the *possible redex* at occurrence u of t. The set of all possible redex positions of t is denoted $PRedPos_R(t)$ and the set of all possible redexes of t is denoted $PRed_R(t)$ (R may be ommited if clear from context). $PRedVar_R(t) = Var(t) \cap (\bigcup_{C \in PRed_R(t)} Var(C))$ is the set of variables of t occurring in one of its possible redexes. For a variable x of $PRedVar_R(t)$ $PRedDepth_R(x)$ is the maximal depth of x in a possible redex of t. The context C that does not contain any possible redex and that is such that $t = C[t_1, \ldots, t_n]$ where u_i for $1 \leq i \leq n$ is a redex position, is called the *irreducible part* of t and is denoted $Irr_R(t)$. If $u < v$ are two possible redex postions of t then v is said to be *nested.*

Example 1. For the TRS $R = \{f(s(x)) \rightarrow c(f(p(x)), f(f(x)))\}$ and the term $t = c(f(p(x)), f(f(x)))$ we have $PRedPos_R(t) = \{2, 2.1\}$, $PRed_R(t) = \{f(\Box), f(x)\}$, $Irr_R(t) = c(f(p(x)), \Box_1)$, $PRedVar(t) = \{x\}$, and $PRedDepth(x) = 1$. t contains a nested redex.

Notice that for any term t, $Pos(t)$-basic derivations and $PRedPos(t)$-basic derivations are the same since positions of t which are not in $PRedPos(t)$ cannot be rewritten.

4.2 Translating TRS to Logic Programs

Table 1 specifies the rules for transforming terms and rewrite rules to clause logic. For a TRS R, let $\mathcal{LP}(R)$ denote the logic program consisting of the clauses obtained by applying the fourth rule to all rewrite rules in R. For sake of simplicity, we will denote by x_u the fresh variable introduced in the third rule for the subterm $f(s_1, \ldots, s_n)$ at occurrence u of a rhs s and A_u the atom produced by this rule.

Table 1. Converting rewrite rules to clause logic

$$\frac{\top}{v \rightsquigarrow \langle v, \emptyset \rangle} \quad \text{if } v \in \mathit{Var}$$

$$\frac{s_1 \rightsquigarrow \langle t_1, \mathcal{G}_1 \rangle \ldots s_n \rightsquigarrow \langle t_n, \mathcal{G}_n \rangle}{f(s_1, \ldots, s_n) \rightsquigarrow \langle f(t_1, \ldots, t_n), \bigcup_i \mathcal{G}_i \rangle} \quad \text{if } \varepsilon \notin \mathit{PRedPos}_R(f(s_1, \ldots, s_n))$$

$$\frac{s_1 \rightsquigarrow \langle t_1, \mathcal{G}_1 \rangle \ldots s_n \rightsquigarrow \langle t_n, \mathcal{G}_n \rangle}{f(s_1, \ldots, s_n) \rightsquigarrow \langle x, \bigcup_i \mathcal{G}_i \bigcup \{P_f(t_1, \ldots, t_n, x)\} \rangle} \quad \text{if } \varepsilon \in \mathit{PRedPos}_R(f(s_1, \ldots, s_n))$$

$$\frac{s \rightsquigarrow \langle t, \mathcal{G} \rangle}{f(s_1, \ldots, s_n) \to s \rightsquigarrow P_f(s_1, \ldots, s_n, t) \leftarrow \mathcal{G}} \quad \text{if } f(s_1, \ldots, s_n) \to s \in \mathcal{R}$$

Example 2. The following rewrite rules and clauses specify multiplication and addition.

$$\begin{array}{ll}
*(0,x) \to 0 & \rightsquigarrow P_*(0,x,0) \leftarrow \\
(s(x),y) \to +(y,(x,y)) & \rightsquigarrow P_*(s(x),y,x_\varepsilon) \leftarrow P_+(y,x_2,x_\varepsilon), P_*(x,y,x_2) \\
+(0,x) \to x & \rightsquigarrow P_+(0,x,x) \leftarrow \\
+(s(x),y) \to s(+(x,y)) & \rightsquigarrow P_+(s(x),y,s(x_1)) \leftarrow P_+(x,y,x_1)
\end{array}$$

Let $\mathcal{P}_{Id} = \{\, P_f(x_1, \ldots, x_n, f(x_1, \ldots, x_n)) \leftarrow \ \mid f \in \mathcal{F} \,\}$. $\mathcal{P}_{Id}$ allows to stop any derivation any time.

Theorem 6. *Let R be a TRS, s a term such that $s \rightsquigarrow \langle s', \mathcal{G} \rangle$. $s \to^*_{bas} t$ iff $\mathcal{LP}(R) \cup \mathcal{P}_{Id} \models \mathcal{G}\mu$ and $t = s'\mu$.*

Unfortunately – but this is not a surprise – the transformation of any term rewriting system does not usually lead to a cs-program. This is mainly due to the non linearity of the bodies of resulting clauses as well as their non flatness. Non-linearity has itself two causes. The first one is the non-linearity of the rhs of the rewriting rule, the second is due to nested redexes. Therefore term rewriting systems for which basic derivations can be expressed by a cs, should have linear rhs with no nested possible redexes. In Section 5, we present two classes of TRS where non-flatness have been weaken thanks to quasi-cs-programs and ib-programs.

5 Term Rewrite Systems Preserving Recognizability

In this section, we give two classes of TRS for which the encoding presented section 4 leads to a cs-program which has the additionnal property that the last argument of each predicate is free (i.e. it is a regular language). This argument encodes the resulting term of the rewrite derivation, this allows to deduce recognizability preservation for certain kind of input languages.

Definition 6. *A quasi-cs-TRS is a TRS with the following properties:*

- *it is right linear*
- *no rhs contains nested possible redexes*
- *For $l \to r \in R$, $x \in PRedVar(r)$ implies that either $x \notin Var(l)$ or $PRedDepth(x)$ is less or equal to the minimal depth at which x occurs in l.*

Definition 7. *An ib-TRS is a TRS with the following properties:*

- *it is right linear*
- *each rhs does not contain nested possible redexes*
- *if C is a possible redex of a right-hand-side and l a left-hand-side then l is an instance of C*

Theorem 7. *For both ib-TRS and quasi-cs TRS, $\to^*$ can be represented by a cs-program.*

Proof. Since both ib-TRS and quasi-cs-TRS are right linear, $\to^*$ and $\to^*_{bas}$ are equal for this two classes. Nowt is sufficient to prove that the logic program obtained from these classes of TRS are ib- or quasi-cs-programs. Since ib- and quasi-cs-TRS are right linear and do not contain nested possible redexes in the rhs, the logic program obtained contains only clauses with linear bodies. Moreover, for quasi-cs-TRS, the depth of the variable in the possible redexes of the rhs is less or equal to its minimal depth in the lhs, the depth of the variable in the bodies is less or equal to the minimal depth of this variable in the head. For ib-TRS, the fact that possible redexes of the rhs subsume lhs, ensures the ib property of the resulting logic program.

Lemma 5. *The resulting program of a quasi-cs-TRS or an ib-TRS is such that the last component of each predicate is free.*

In fact, this lemma is not true for the rules as given in Table 1. A rule like $+(x,0) \to x$ is transformed to the clause $P_+(x,0,x) \leftarrow$; obviuosly, the 3^{rd} position of P_+ is not free because x occurs twice in the head of the clause. This problem appears for each rewrite rule $l \to r$ such that $Irr_R(r) \backslash PRedVar_R(r) \neq \emptyset$. Notice that since $l \to r$ is right linear, $x \in PRedVar(r)$ occurs once in the last argument of the head and once in the body. So for all $f(s_1,\ldots,s_n) \to r$ such that the resulting clause is $P_f(s_1,\ldots,s_n,s) \leftarrow \mathcal{B}$, we define $\sigma_r = \{\, x \mapsto x_r \mid x \in Var(s) \setminus \mathcal{B} \,\}$ and $\mathcal{B}_r = \{\, P_{Id}(x,x_r) \mid x \in Dom(\sigma_r) \,\}$ and we transform the clause $P_f(s_1,\ldots,s_n,s) \leftarrow \mathcal{B}$ to $P_f(s_1,\ldots,s_n,s\sigma') \leftarrow \mathcal{B}_r \cup \mathcal{B}$. This new clause is still a quasi or ib-clause and the last argument is linear and does not share variables with other arguments. If P_{Id} is defined by the set of clauses (which are both ib- and quasi-cs-clause) $P_{Id}(f(\vec{x}), f(\vec{x'})) \leftarrow P_{Id}(\vec{x}, x')$ then the semantics of P_f remains unchanged with regard to the Herbrand Models.

For example, considering the rewriting rule $+(x,0) \to x$, we have $\sigma_r = \{x \mapsto x_r\}$, $\mathcal{B}_r = \{P_{Id}(x,x_r)\}$, and the transformed clause is $P(x,0,x_r) \leftarrow P_{Id}(x,x_r)$.

Now we show how to use these results to compute $R^*_{bas}(E)$ for R which is either a quasi-cs or an ib TRS and E a regular tree language. In [10] it has been shown that $R^*(E)$ is not regular for the TRS $R = \{f(g(x)) \to g(f(x))\}$ which is both ib and quasi-cs, and the regular tree language $E = (fg) * 0$. $R^*(E)$ is the set of terms which contains as much f as g, in particular $R^{\downarrow}(E) = g^n f^n 0$, which is not a regular language. One can remark that terms of E contains unbounded number of possible redexes. So E is to be restricted to a regular tree language with a bounded number of possible redexes.

Definition 8. *A regular tree languages E is* possible-redex-bounded *if there exists an integer k such that every term t in E contains at most k possible redexes.*

Theorem 8. *Let E be a possible redex bounded regular tree language and R be a right-linear TRS which is either instance-based or quasi-cs. Then $R^*_{bas}(E)$ is a regular tree language.*

Proof. A possible redex bounded regular tree language can be described by a logic program with two kinds of clauses:
$P(f(x_1, \dots, x_n)) \leftarrow P_1(x_1), \dots, P_n(x_n)$ for positions that are not rewritten and $P(y) \leftarrow P_f(x_1, \dots, x_n, y), P_1(x_1), \dots, P_n(x_n)$ for possible redex positions (notice that this clause is almost a regular join definition) in this case body of clauses defining P do not contain P and no other clauses is headed by P.

The set of non-regular definitions L_* can be stratified in the following way:
$L_0 = \{P(y) \leftarrow P_f(x_1, \dots, x_n, y), P_1(x_1), \dots, P_n(x_n)$ s.t. the definition of the P_i does not contain any join definition $\}$.
$L_i = \{P(y) \leftarrow P_f(x_1, \dots, x_n, y), P_1(x_1), \dots, P_n(x_n) \notin \bigcup_{j<i} L_j$ s.t. the definition of the P_i contains no non regular definitions but those of $\bigcup_{j<i} L_j\}$.

Then for a right-linear irreducible-based TRS R which is either ib or quasi-cs and tree language defined as above by a logic program *Input*, we first compute $\mathcal{LP}(R)$ and then transform it into the cs-program $cs(\mathcal{LP}(R))$, then we apply the following algorithm:

```
$\mathcal{P}_0 = (Input \setminus L_*)$, $Done = \emptyset$, $k = 1$
while $Done \neq L_*$
  Let $i = Min(\{ j \mid L_j \setminus Done \neq \emptyset \})$
  Let $C \in L_i \setminus Done$
  Let $\mathcal{P}'$ be the cs-program computed from $C$ and $P_{k-1} \cup cs(\mathcal{LP}(R))$
  $P_k = Reg(\mathcal{P}')$
  $k = k + 1$
end while
```

By the definition of *Input*, $\mathcal{P}_0$ is a linear monadic program. So the first iteration of the algorithm is a regular join definition transformation. If for a step k C is a regular join definition compatible with $P_k \cup cs(\mathcal{LP}(R))$, we have from freeness property the lonely argument of each predicate of L_i will be free, therefore $Reg(\mathcal{P}')$ produces only linear monadic clauses, so $\mathcal{P}_{k+1}$ linear monadic.

As a consequence $\mathcal{P}_*$ which defines the language $R^*_{bas}(E)$, describes a regular tree language.

This result can be used for example to compute the set of descendants of ground constructor instances of a linear term, for a constructor based TRS.

Indeed, let the term $f(s(g(x))$ where constructor symbols are s and 0. Its ground constructor instances contains only two possible redexes. The program is

$P(x) \leftarrow P_f(x_1, x), P_s(x_1)$ $P_s(s(x)) \leftarrow P'(x)$ $P'(x) \leftarrow P_g(x_1, x), P_*(x_1)$
$P_*(0) \leftarrow$ $P_*(s(x)) \leftarrow P_*(x)$

But it is also possible to compute the set of descendants of the regular language $s^*(f(s^*(g(x))))$ where f and g may occur at any depth in the terms of the language. This weaken the restriction of [10] on the kind of regular language allowed for computing $R^*(E)$, and can be useful for reachability problems issued from infinite state system verification for example.

Our result on ib-TRS extends those of [10] because all constructor based TRS satisfying Réty restrictions are ib-TRS but the TRS containing the single rule $f(s(x)) \rightarrow f(f(p(x)))$ does not satisfy the condition on nested function symbols. Quasi cs-TRS is neither included nor includes Réty's TRS. Indeed the former TRS is an instance-based one. On the other hand $\{f(x) \rightarrow g(s(x)), g(s(p(x))) \rightarrow g(x)\}$ respects Réty's restrictions but is not an ib-TRS. Another main class of TRS which has been studied is right-linear finite path overlapping TRS defined in [12]. This class allows nested possible redexes which is forbidden for cs-TRS. On the other hand rewriting rules like $f(s(x)) \rightarrow f(x)$ are not allowed in [12], but as the name of the class indicates, right-linearity is required, therefore basic rewriting relation is equivalent to rewrite relation, so it should be possible to handle this kind of TRS in our framework. More recently [11] defines the class of layered transducing TRS (LT-TRS for short) which are preserving recognizability under some conditions. A LT-TRS is a linear TRS working over a signature where some unary function symboles are distinguished as markers. The rule of LT-TRS are of one of the following two forms: $f(q_1(x_1), \ldots, q_n(x_n)) \rightarrow q(t)$ or $q(x) \rightarrow q'(t)$ where $q, q', q_1, \ldots, q_n$ are markers and t, t' do not contain any markers. The class of LT-TRS is a strict subclass of ib-cs TRSs since t and t' do not contain any possible redexes. The authors of this article define two conditions on LT-TRS to obtain the recognizability preservation. The first one corresponds to the conditions of theorem 8 on the input language. The second condition defining the IO separated LT-TRS, is on the marker symbols and allows to get the preservation for any input languages which is not the case for ib-cs TRSs.

6 Conclusion and Future Work

The translation of basic rewriting to logic programming presented in this paper provides a simple way to obtain finite presentations for derivations that allow to study properties of term rewrite systems. Its generality allows to extend already known results as well as to get new ones. Other classes of logic programs also transform to finite cs-programs. Pseudo-regular programs, for example, extend regular relations of [2] by weakening restriction on copying clauses. This should lead to further classes of term rewrite systems preserving recognizability. Pseudo-regular tuple languages are closed under intersection, therefore we expect that the corresponding class of term rewrite systems is less restrictive on the right-hand-sides. Our final aim is to give a complete characterization of term rewrite systems preserving regularity that correspond to finite cs-programs.

A nice side effect of translating everything to logic programs is that we obtain without much effort prototype implementations in Prolog which allow to experiment with the results[4].

Acknowledgements

We would like to thank the four referees for their substantial comments and valuable suggestions. We are particularly grateful for the reference to [9]: while this work predates some of our results in [5] and makes them obsolete, it supports our conviction that well-established notions and techniques from logic programming are of advantage also to other term-oriented fields like term rewriting or tree tuple languages.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, United Kingdom, 1998.
2. H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications (TATA).* `http://www.grappa.univ-lille3.fr/tata`, 1997.
3. P. Gyenizse and S. Vágvölgyi. Linear generalized semi-monadic rewrite systems effectively preserve recognizability. *Theoretical Computer Science*, 194:87–122, 1998.
4. S. Limet and P. Réty. E-unification by means of tree tuple synchronized grammars. *Discrete Mathematics and Theoretical Computer Science*, 1:69–98, 1997.
5. S. Limet and G. Salzer. Manipulating tree tuple languages by transforming logic programs. In Ingo Dahn and Laurent Vigneron, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.
6. S. Limet and G. Salzer. Proving properties of term rewrite systems via logic programs. Technical report, RR-2004-5, LIFO Université d'Orléans, www.univ-orleans.fr/SCIENCES/LIFO/prodsci/publications/lifo2004.htm.en, 2004.
7. J.W. Lloyd. *Foundations of Logic Programming.* Springer, 1984.
8. D. Lugiez and Ph. Schnoebelen. The regular viewpoint on pa-processes. *Theoretical Computer Science*, 274(1-2):89–115, 2002.
9. M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.
10. P. Réty. Regular sets of descendants for constructor-based rewrite systems. In *Proc. of the 6th conference LPAR*, number 1705 in LNAI. Springer, 1999.
11. H. Seki, T. Takai, Y. Fujinaka, and Y. Kaji. Layered transducing term rewriting system and its recognizability preserving property. In S. Tison, editor, *Proc. of 13th Conference RTA*, volume 2378 of *LNCS*. Springer, 2002.
12. T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *11th International Conference RTA*, volume 1833 of *LNCS*, pages 270–273. Springer, 2000.
13. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S. Tärnlund, editor, *Proc. 2nd Int. Logic Programming Conf. (ICLP)*, pages 127–138. University of Uppsala, Sweden, 1984.

[4] See e.g. `www.logic.at/css` for a Prolog program for computing cs-programs.