UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

Solving First Order Formulae of Pseudo-Regular Theory

Sébastien Limet et Pierre Pillot
Université d'Orléans, LIFO

Rapport N^o **2005-02**

Solving First Order Formulae of Pseudo-Regular Theory

Sébastien Limet and Pierre Pillot

LIFO, Université d'Orléans, France
{limet|pillot}@lifo.univ-orleans.fr

Abstract. In this paper, we study the class of pseudo-regular relations which is an extension of regular relations that weakens some restrictions on the "synchronization" between tuple components of the relation. We choose logic programming as formalism to describe tree tuple languages (i.e. relations) and logic program transformation techniques for computing operations on them. We show that even if pseudo-regular cs-programs are syntactically less restrictive than regular ones, they define the same class of tree tuple languages. However, pseudo-regular relations allow one to define classes of term rewrite systems the transitive closure of which is a regular relation. We apply this result to give a decidable class of first order formulae based on the joinability predicate $\downarrow_R^?$ where R is a pseudo-regular term rewrite system.

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', are there a substitution σ and a term u 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?). In this paper, we are interested in solving formulae where the only predicate is the R-joinability $\downarrow_R^?$ where R is a term rewrite system. R-joinability coincides with R-unification for confluent rewrite systems.

Solving a single equation $s \downarrow_R^? t$ is known to be undecidable without any strong restrictions on the TRS [4, 5]. Some positive results have been shown restricting the TRS to obtain a finitary equational problem (i.e. insuring that the set of minimal solutions is finite). For example shallow or standard theories [3, 12] impose that the depth of all variables in the (un-oriented) TRS is one. In fact standard property weakens a little this restriction by allowing variables at depth more than one when they are not under a defined function symbol. For infinitary theories (i.e. theories where the set of solutions of an equation $s \downarrow_R^? t$ may be infinite) many results have been given to decide the existence of solutions using the reachability problem (e.g. [14, 15]) but very few of them [5, 8] give a finite representation of the solutions and even less go beyond solving a single equation.

In order to reach our aim, we have first to solve the problem of the representation of a possibly infinite set of solutions of an equation $s \downarrow_R^? t$. The most common tools used for such a representation are tree languages recognizers [2] such as tree automata or tree grammars. In our context, a tree language is a set of ground terms that is represented by a logic program as in e.g. [6, 11]. More precisely, we use the class of cs-programs of [6].The operations on the languages are computed thanks to logic program transformations inspired from [13].

Many of the tree tuple languages are not closed under all the set operations. As far as we know the only class that is closed under those operation and have decidable emptiness and membership tests are the regular relations [2]. Pseudo-regular relations are tree tuple languages that allow some kind of duplications between the components of the tuple. In [6], the emptiness and membership tests as well as the intersection, projection, union and join (which can be seen as a composition of relations) of pseudo-regular relations have been described in term of logic program transformations, but no result about the complement of a pseudo-regular relation is given. In this paper, we give an algorithm that computes the complement of pseudo-regular relations.

Once the finite representation of infinite sets of solutions has been chosen, we have to give an algorithm to compute a representation of the solutions of equations $s \downarrow_R^? t$. We use the method described in [7], where a term rewrite system is transformed into a logic program that encodes the transitive closure of the rewrite relation $\rightarrow_R^*$. Thanks to this transformation, we can define a class of TRS which corresponding logic programs are pseudo-regular. Then thanks to the closure of pseudo-regular relation under the set operations, we can compute the solutions of any formulae where $\downarrow_R^?$ is the only predicate when R belongs to this class.

2 Preliminaries

We recall some basic notions and notations concerning terms, term rewrite systems and logic programming; for details see [1, 9].

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. A term is *linear* if no variable occurs more than once in it and a term without variable is called a *ground term*. Σ consists of three disjoint subsets: the set $\mathcal{F}$ of *defined function symbols*, the set $\mathcal{C}$ of *constructor symbols* and the set $\mathcal{P}r$ of *predicate symbols*. The terms of $\mathcal{T}(\mathcal{C}, Var)$ are called *data-terms* and those of the form $P(\vec{t})$ where P is a predicate symbol of arity n and $\vec{t}$ is a vector of $\mathcal{T}(\mathcal{F} \cup \mathcal{C}, Var)^n$ are called *atoms*.

A *position* p is a list of integers which length is denoted by $|p|$. 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 $u \in Pos(t) \setminus \Sigma Pos(t)$. The *depth of a term* $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{C}, Var)$ denoted $Depth(t)$ is 0 if $t \in Var$ and $Max(\{\, |p| \mid p \in \Sigma Pos(t)\,\}$

otherwise. The *depth of an atom* $P(\vec{t})$ denoted $Depth(P(\vec{t}))$ is $Max(\{ Depth(s) \mid s \in \vec{t} \})$.

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 $\{ x \in Var \mid x\sigma \neq x \}$. 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 $\forall x \in Dom(\sigma)$, $x\sigma$ is a data-term then σ is called a *data substitution*. 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 $CVar = \{ \Box_i \mid i \geq 1 \}$ be the set of *context variables* distinct from Var, a *n-context* is a term t in $\mathcal{T}(\Sigma, Var \cup CVar)$ such that each $\Box_i$ $1 \leq i \leq n$ occurs once and only once in t and no other element of $CVar$ occurs in t. $\Box_1$ (also denoted $\Box$) is called the *trivial context*. For an n-context C, the expression $C[t_1, \ldots, t_n]$ denotes the term $C\{ \Box_i \mapsto t_i \mid 1 \leq i \leq n \}$.

A *term rewrite system* (TRS) is a finite set of oriented equations built over $\mathcal{T}(\mathcal{F} \cup \mathcal{C}, Var)$ and called *rewrite rules*. Lhs and rhs are shorthands for the left-hand and right-hand side of a rule, respectively. A TRS is *constructor based* iff every rule is of the form $f(t_1, \ldots, t_n) \to r$ where all t_i's are data-terms and f is a defined function symbol. For a TRS R, the *rewrite relation* is denoted by $\to_R$ and is defined by $t \to_R s$ iff there exists a rule $l \to r$ 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 $t \to_{[u, l \to r]} s$. If σ is a data-substitution then the step is called a *data-step*. If a term t cannot be reduced by any rewriting rule, it is said to be *irreducible*. The reflexive-transitive closure of $\to_R$ is denoted by $\to_R^*$.

The *joinability relation* $\downarrow_R$ is defined by $t \downarrow_R s$ iff $t \to_R^* u$ and $s \to^* u$ for some term u. Notice that R-joinability is equivalent to R-unifiability for confluent rewrite systems. In the context of constructor based TRS, a *data-solution* of a joinability equation $s \downarrow_R^? t$ is a data-substitution σ such that $s\sigma \to_R^* u$ and $t\sigma \to_R^* u$ where u is a data term and all rewriting steps are data-steps.

If $H, A_1, \ldots, A_n$ are atoms then $H \leftarrow A_1, \ldots, A_n$ is a *Horn clause*, H is called the head and $A_1, \ldots, A_n$ is called the body of the clause. The elements of $Var(A_1, \ldots, A_n) \setminus Var(H)$ are called *free variables*. A *logic program* is a set of Horn clauses. The body of the clause $H \leftarrow \mathcal{B}$ is said linear iff every variable occurs at most once in $\mathcal{B}$. A clause is said to be *linear* if both the head and the body are linear. The *Herbrand domain* is the set of all ground atoms. A set of ground atoms S is an *Herbrand model* of the clause $H \leftarrow \mathcal{B}$ iff $\forall \sigma$ such that $\mathcal{B}\sigma \subseteq S$, $H\sigma \in S$. S is an Herbrand model of the logic program $\mathcal{P}$ if it is a model of all clauses of $\mathcal{P}$. For a logic program $\mathcal{P}$ and a ground atom A we write $\mathcal{P} \models A$ if A belongs to the least Herbrand model of $\mathcal{P}$. The language described by a n-ary predicate symbol P w.r.t. a program $\mathcal{P}$ is the set $\{ (t_1, \ldots, t_n) \mid \mathcal{P} \models P(t_1, \ldots, t_n) \}$ of n-tuples of ground terms.

3 Pseudo-regular relations

This section first presents the logic programming formalism we use to represent the tree tuple languages and then some new results about pseudo-regular relations.

3.1 Representing tree tuple languages

A *cs-program* is simply a logic program consisting in Horn clauses with linear bodies and without function symbols. [6] introduces two subclasses of cs-programs, called respectively regular and pseudo-regular programs. The first class corresponds to the regular relations of [2], and the second one weakens the syntax of regular programs.

Definition 1. *Let $H \leftarrow \mathcal{B}$ be a clause such that $\mathcal{B}$ contains no function symbols.*

- *$H \leftarrow \mathcal{B}$ is called* pseudo-regular-like *(PR-like for short) iff each argument of H is of the form $f(x_1, \ldots, x_{\mathrm{ar}(f)})$ where f is a function symbol and the x_is are pairwise distinct variables and there exists a mapping π: $Var \mapsto \mathbb{N}$ such that $\pi(x_l) = l$ for all $l = 1, \ldots, \mathrm{ar}(f)$ and $\pi(x) = \pi(y)$ for all variables x and y occurring in the same body atom.*
- *$H \leftarrow \mathcal{B}$ is called* pseudo-regular *(PR for short) iff it is PR-like and $\mathcal{B}$ is linear.*
- *$H \leftarrow \mathcal{B}$ is called* regular *(R for short) iff it is PR and H is linear*
- *$H \leftarrow \mathcal{B}$ is called* shared pseudo-regular *(shared-PR for short) iff it is PR-like and contains no free variable*
- *$H \leftarrow \mathcal{B}$ is called* shared regular *(shared-R for short) iff it is shared-PR and H is linear.*

A program is PR-like, PR, R, shared-PR or shared-R if all its clauses are of the corresponding type.

Example 1. The clause $P(c(x_1, y_1), c(x_1, y_2)) \leftarrow P_1(y_1), P_2(x_1, y_2)$ is not PR-like since x_1 and y_2 occur in the same body atom but they do not occur at the same position in the head of the clause.

The clause $P(c(x_1, y_1), c(x_1, y_2)) \leftarrow P_2(y_1, z), P_2(y_2, y_1)$ is PR like but it is neither PR because the body is not linear nor shared-PR since z is a free variable.

The following logic program is PR. Notice that the clauses defining $Id2$ are regular since their heads are linear

$P_f(s(x_1), s(x_2), s(x_3)) \leftarrow Q(x_1, x_2, x_3)$	$Id_2(c(x_1, y_1), c(x_2, y_2)) \leftarrow Id_2(x_1, x_2), Id_2(y_1, y_2)$
$P_f(0, 0, 0) \leftarrow$	$Id_2(s(x), s(y)) \leftarrow Id_2(x, y)$
$P_g(s(x), s(x))$	$Id_2(0, 0) \leftarrow$

$Q(c(x_1, y_1), c(x_1, y_2), c(x_2, y_3)) \leftarrow P_g(x_1, x_2) P_f(y_1, y_2, y_3)$

Decidability of membership and emptiness tests as well as closure under intersection of pseudo-regular relations have been shown in [6] using the logic program transformation rules summarized below (see [6] for more details).

Definition 2. *A* definition *is an equivalence of the form $H \leftrightarrow \mathcal{B}$ (abusively written $H \leftarrow \mathcal{B}$) where H is a single atom of depth* 0 *and $\mathcal{B}$ is a set of atoms.*

A set of definitions, $\mathcal{D}$ is compatible with *a logic program $\mathcal{P}$, if all predicate symbols occurring in the heads of the definitions occur in exactly one head and nowhere else in D and $\mathcal{P}$ the only exception are tautological definitions of the form $P(x) \leftarrow P(x)$ where P may occur without restriction throughout $\mathcal{D}$ and $\mathcal{P}$.*

A definition $H \leftarrow \mathcal{B}$ is called a general join-definition *if H is linear and $H, \mathcal{B}$ do not contain function symbols.*

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 $\mathcal{P}$ is the input logic program, $\mathcal{D}_{\text{new}}$ 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.

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. 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.

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

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.

Definition Introduction.

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

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

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

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.

The closure of pseudo-regular relations under intersection can be proved thanks to the following result of [6].

Theorem 1. *[6] Let $\mathcal{P}$ be a PR program, and let D_p a general join definition. Any complete derivation with input $\mathcal{P}$ and $\{D_p\}$ that unfolds in each unfolding step all atoms simultaneously is finite and its output is a PR program.*

Example 2. Let us consider the PR program of Example 1. The general join definition $I(x, y) \leftarrow P_g(x, y), Id_2(x, y)$ defines the intersection of the relations defined respectively by P_g and Id_2. The $\Rightarrow$-derivation with input program $\mathcal{P}$ and the former join definition can be summarized as follows $\mathcal{D}_{\text{done}}$ is omitted and the last column indicates which rule has been applied:

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rule
$I(x, y) \leftarrow P_g(x, y),$ $Id_2(x, y)$			U
	$I(s(x), s(x)) \leftarrow Id_2(x, x)$		D
$I'(x) \leftarrow Id_2(x, x)$		$I(s(x), s(x)) \leftarrow I'(x)$	U
	$I'(0) \leftarrow$ $I'(s(x)) \leftarrow Id_2(x, x)$ $I'(c(x, y)) \leftarrow Id_2(x, x),$ $Id_2(y, y)$		D
		$I'(0) \leftarrow$ $I'(s(x)) \leftarrow I'(x)$ $I'(c(x, y)) \leftarrow I'(x), I'(y)$	

The output program consists in the clauses of $\mathcal{C}_{\text{out}}$.

The decidability of membership test and emptiness test of language defined by PR programs come from the fact that PR-programs are cs-programs which have these properties [6]. The closure under union is obvious.

Remark 1. Consider that the Definition Introduction rule is modified in the following way

Definition Introduction.

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

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

$$L_i = \begin{cases} L\eta^{-1} & \text{if } (L \leftarrow \mathcal{B}_i)\eta \in \mathcal{D}_{\text{done}} \text{ for some var. renaming } \eta \text{ and} \\ & \quad Var(\mathcal{B}_i\eta^{-1}) \cap Var(H) = Var(\mathcal{B}_i\eta^{-1}) \cap Var(L\eta^{-1}) \\ P_i(x_1, \ldots, x_n) & \text{otherwise, } \{x_1, \ldots, x_n\} \text{ being } Var(\mathcal{B}_i) \cap Var(H). \end{cases}$$

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.

In other word, we consider that a componant $\mathcal{B}_i$ is equivalent to the definition $L \leftarrow \mathcal{B}_i$ if the free variables of $\mathcal{B}_i$ in $H \leftarrow \mathcal{B}_1 \dot\cup \cdots \dot\cup \mathcal{B}_k$ are the same as the free variables of $\mathcal{B}_i$ in $L \leftarrow \mathcal{B}_i$ in the one hand and in the other hand only the non-free variables of $\mathcal{B}_i$ are kept in the new definition.

The Theorem 1 still holds with this new version of Definition Introduction since the termination criterion is based on the number of occurrences of variables in each $\mathcal{B}_i$ which is not modified in this version. It does not change the "regularity" of the resulting program since each definition $P(\vec{x}) \leftarrow \mathcal{B}$ introduced by this new version, is such that $\mathit{Var}(P(\vec{x}) \subseteq \mathit{Var}(\mathcal{B})$. The interesting property of this new version of the rule is that each clause $H \leftarrow L_1, \ldots, L_k$ introduced in $\mathcal{C}_{\text{out}}$ is such that $\mathit{Var}(H) = \mathit{Var}(L_1, \ldots, L_k)$ which means that it contains no free variables.

3.2 Shared pseudo-regular programs

In this section, we show how to compute the complement of a pseudo-regular language using logic program transformation techniques. For that we use the class of shared-PR programs. A program of this class can be transformed into a PR program. Moreover if the clause heads of the input program are all linear then the result is a regular program. This result proves the equivalence of pseudo-regular relations and regular relations.

Theorem 2. *Let $\mathcal{P}$ be a shared-PR program, and let $\mathcal{D}_p$ the set of all tautologies $P(x) \leftarrow P(x)$ such that P occurs in $\mathcal{P}$. Any $\Rightarrow$-derivation with input $\mathcal{P}$ and $\mathcal{D}_{\mathcal{P}}$ is finite and its output is a PR program. If the input program $\mathcal{P}$ is shared regular then the result is a regular program.*

Proof. of theorem 2

The proof is based on the fact that only a finite number of general join definitions can be constructed in $\mathcal{D}_{\text{new}}$ when the number of duplicated variables and the maximal number of occurrences of a variable are bounded. In our case it directly results from the limitation of the arity of the clause head predicate of general join definition without free variables by the greatest arity of logic program predicate.

Unfolding. Let be $P(x_1, \ldots, x_n) \leftarrow A_1, \ldots, A_k$ a general join definition. Unfolding this definition using the clauses $H_i \leftarrow B_{i,1}, B_{i,2}, \ldots$ for $1 \leq i \leq k$ we obtain the clause $P(x_1, \ldots, x_n)\mu \leftarrow \mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_m$ where μ is the most general simultaneous unifier of $(A_1, \ldots, A_k)$ and $(H_1, \ldots, H_k)$ and $\mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_m$ is the maximal variable-disjoint decomposition of the body of the clause.

First, let us observe carefully the unification problem involved in the unfolding step. Each H_i is of the form $P_i(f_{i1}(\vec{y}_{i1}), f_{i2}(\vec{y}_{i2}), \ldots)$ where each $f_{ij}(\vec{y}_{ij})$ is a linear term and each A_i is of the form $P_i(\vec{x}_i)$. Moreover, since the clauses are renamed apart prior to unification, $f_{ij}(\vec{y}_{ij})$ and $f_{i'j'}(\vec{y}_{i'j'})$ share no variables for $i \neq i'$. Therefore the computation of μ amounts to unify simultaneously all pairs $(x_{ij}, f_{ij}(\vec{y}_{ij}))$ where each $f_{ij}(\vec{y}_{ij})$ is a linear term. This unification problem may be solved as follows: First, for each $x \in \mathit{dup}(A_1, \ldots, A_k)$, we have the unification problem $\mathit{mgu}_x(\{x_{ij} = f_{ij}(\vec{y}_{ij}) | x_{ij} = x\})$ where $\mathit{dup}(A_1, \ldots, A_k)$ denotes the set of variables that occur at least twice in $A_1, \ldots, A_k$. The decomposition of these problems gives the set of equalities $Eq_x = \{\vec{y}_{ij}|_l = \vec{y}_{i'j'}|_l | x_{ij} = x_{i'j'}\}$. It remains to compute the solution of the unification problem $\cup_{x \in \mathit{dup}(A_1,\ldots,A_k)} Eq_x$. We know that two variables $\vec{y}_{ij}|_l$ and $\vec{y}_{i'j'}|_{l'}$ are equal only if $i = i'$ so from pseudo-regular definition $\pi(\vec{y}_{ij}|_l) = \pi(\vec{y}_{i'j'}|_{l'}) = l = l'$. Therefore if $\vec{y}_{i_1 j_1}|_{l_1} = y_{i'_1 j'_1}|_{l_1}$ and $\vec{y}_{i_2 j_2}|_{l_2} = y_{i'_2 j'_2}|_{l_2}$ of $\cup_{x \in \mathit{dup}(A_1,\ldots,A_k)} Eq_x$ have a variable in common, $l_1 = l_2$.

From this, we can deduce that $x\mu$ is a variable for $x \in Var(H_1 \dots H_k)$ so no function symbols occurs in $\mathcal{B}_1 \dot\cup \cdots \dot\cup \mathcal{B}_m$. We can also see that for any variable $x \in Var(H_1, \dots, H_k)$, $\pi(x) = \pi(x\mu)$ and $x\mu$ occurs at the position $\pi(x\mu)$ in the arguments of $H_1\mu, \dots, H_k\mu$. This induces that all variables of any atom $B_{ij}\mu$ that occurs in $H_i\mu$ have the same image by π. Since $(A_1\mu, \dots, A_k\mu) = (H_1\mu, \dots, H_k\mu)$ and $Var(P(x_1, \dots, x_n)) \subseteq Var(A_1, \dots, A_k)$, we have $Var(P(x_1, \dots, x_n)\mu) \subseteq Var(H_1\mu, \dots, H_k\mu)$. Therefore all variables of any atom of a component $\mathcal{B}_l$ of $P(x_1, \dots, x_n)\mu \leftarrow \mathcal{B}_1 \dot\cup \cdots \dot\cup \mathcal{B}_m$ that occurs in $P(x_1, \dots, x_n)\mu$ have the same image by π and this value is the position in the arguments of $P(x_1, \dots, x_n)\mu$ where they occur. Moreover since the clauses of logic program are shared-PR and since $P(x_1, \dots, x_n) \leftarrow A_1, \dots, A_k$ is a general join definition, they do not contain any free variables. Therefore, $P(x_1, \dots, x_n) \leftarrow \mathcal{B}_1 \dot\cup, \dots, \dot\cup, \mathcal{B}_m$ contains also no free variables.

Definition introduction. The new clauses added to $\mathcal{C}_{\mathrm{out}}$ are of the form $P(x_1, \dots, x_n)\mu \leftarrow L_1, \dots, L_m$ where the variables of each L_i are those of the corresponding component $\mathcal{B}_i$ in the unfolding step. Therefore if a variable occurs at position p of one of the argument of $P(x_1, \dots, x_n)\mu$ it has p as image by π, and all variables of each L_i have the same image by π so it is a pseudo-regular clause. Furthermore all variables of each L_i belong to $P(x_1, \dots, x_n)\mu$ since $Var((\mathcal{B}_1, \dots, \mathcal{B}_m)) \subseteq Var((H_1, \dots, H_k))$, therefore the arity of the predicate of L_i is bounded to n (due to pseudo-regularity property, each argument of $P(x_1, \dots, x_n)\mu$ may contain at most one variable of a particular L_i). Since the number of predicate symbols in $\mathcal{P}$ is finite and fixed, the number of distinct (up to a variable renaming) sets of atoms built over these two bounded sets is finite.

Corollary 1. *Any shared regular (resp. PR) program $\mathcal{P}$ can be transformed into an equivalent regular (resp. PR) program.*

Remark 2. According to Remark 1, we can obtain a PR or a R program that does not contain clauses with free variables from a shared-PR or a shared-R program by using the modified version of Definition Introduction.

If the input set of join definitions is the set of all tautologies $P(\vec{x}) \leftarrow P(\vec{x})$ such that P occurs in $\mathcal{P}$, then the $\Rightarrow$-derivation produces a regular (or PR) program that defines the same predicates as $\mathcal{P}$.

Corollary 2. *Any PR program can be transformed into an equivalent finite regular program.*

Proof. Without loss of generality we can consider that the PR program does not contain clauses with free variables (see Remark 2). Let us observe that any PR clause $H \leftarrow \mathcal{B}$ is equivalent to the shared regular clause $H' \leftarrow \mathcal{B}' \dot\cup Id$ where

- H' is a linear atom such that $H = H'\sigma$ where σ is a substitution from *Var* to *Var*, $\mathcal{B} = \mathcal{B}'\sigma$ and
- $Id = \{Id_n(x_1, \dots, x_n) | n > 1$ and $\{x_1, \dots, x_n\}$ is the set of variables that have the same image by $\sigma\}$ where each Id_i are predicates defined by R programs and define the set of i-tuples of identical terms.

$H' \leftarrow \mathcal{B}' \dot\cup Id$ is a shared regular since it has no free variables, H' is linear and $\forall x, y \in Var(A)$ with $A \in \mathcal{B}' \dot\cup Id$, x and y occur at the same position in the arguments of H'. This shared-R program can be transformed into a R program.

The equivalence of regular relations and pseudo-regular relations does not decrease the interest for PR programs since they are syntactically less restrictive than regular ones as Section 4 will show.

Now shared-PR programs are used to give an algorithm that computes the complement of tree tuple language defined by a predicate P of a PR program $\mathcal{P}$ using the $\Rightarrow$ transformation. This algorithm is useful to compute the solutions of pseudo-regular formulae thanks to logic programs transformations. It consists in two steps. The first step computes from $\mathcal{P}$ a new shared regular program $\overline{\mathcal{P}}$ that defines the complement of P. The second step uses Corollary 1 to obtain a regular program equivalent to $\overline{\mathcal{P}}$.

Let P be a predicate defined by the PR program $\mathcal{P}$. Without loss of generality we can consider that $\mathcal{P}$ has neither unproductive clauses (i.e. clauses that do not contribute to the least Herbrand model), nor clauses with free variables. Then $\mathcal{P}$ is transformed into the shared regular program $\mathcal{P}'$ by the technique used in proof of Corollary 2.

For each predicate Q of arity n of $\mathcal{P}'$ and each n-tuple $(f_1, \dots, f_n)$ of $\mathcal{C}^n$ we define the set $AC(Q, (f_1, \dots, f_n))$ of all clauses of $\mathcal{P}$ which head is $Q(f_1(\vec{x}_1), \dots, f_n(\vec{x}_n))$ as $\{Q(f_1(\vec{x}_1), \dots, f_n(\vec{x}_n)) \leftarrow \mathcal{B} \in \mathcal{P}'\}$. Notice that since the clause heads of $\mathcal{P}'$ are linear all the clause heads of $AC(Q, \vec{f})$ are equal up to a variable renaming. In the following we consider them equal. For an atom $A = P(\vec{t})$, $\overline{A}$ stands for $\overline{P}(\vec{t})$.

Let $\vec{f} = f_1, \dots f_n$, $\vec{x}_i$ $(1 \leq i \leq n)$ be pairwise distinct vectors of distinct variables and $AC(Q, \vec{f}) = \{H \leftarrow \mathcal{B}_1, \dots, H \leftarrow \mathcal{B}_k\}$. We define

$$\overline{AC}(Q, \vec{f}) = \begin{cases} \emptyset & \text{if } H \leftarrow \in AC(Q, \vec{f}) \\ Q(f_1(\vec{x}_1), \dots, f_n(\vec{x}_n)) \leftarrow & \text{if } AC(Q, \vec{f}) = \emptyset \\ \{\, \overline{H} \leftarrow \overline{A_1}, \dots, \overline{A_k} \mid \forall i \in [1, k] A_i \in \mathcal{B}_i \,\} & \text{otherwise} \end{cases}$$

Example 3. Let consider the logic program consisting in the three following clauses $P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(x_1, x_2)$, $\quad P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(y_1, y_2)$ and $P(0, 0) \leftarrow$

$AC(P, (0, c)) = \emptyset$	$\overline{AC}(P, (0, c)) = \{\overline{P}(0, c(x, y)) \leftarrow\}$
$AC(P, (c, 0)) = \emptyset$	$\overline{AC}(P, (c, 0)) = \{\overline{P}(c(x, y), 0) \leftarrow\}$
$AC(P, (0, 0)) = \{P(0, 0) \leftarrow\}$	$\overline{AC}(P, (0, 0)) = \emptyset$
$AC(P, (c, c)) = \{$ $P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(x_1, x_2),$ $P(c(x_1, y_1), c(x_2, y_2)) \leftarrow P(y_1, y_2)\}$	$\overline{AC}(P, (c, c)) = \{\ \overline{P}(c(x_1, y_1), c(x_2, y_2)) \leftarrow$ $\overline{P}(x_1, x_2), \overline{P}(y_1, y_2)\}$

Lemma 1. *Let $\mathcal{P}$ be a shared regular program without any unproductive clauses. The program $\overline{\mathcal{P}}$ consisting in the set of clauses $\bigcup_{P \in \mathcal{P}, \vec{f} \in \mathcal{C}^n} \overline{AC}(P, \vec{f})$ is such that $\forall P \in \mathcal{P}, \forall (t_1, \dots, t_n) \in T^n_{\mathcal{C}}, \mathcal{P} \models P(t_1, \dots, t_n)$ iff $\overline{\mathcal{P}} \models \overline{P}(t_1, \dots, t_n)$.*

Proof. We show that $\overline{\mathcal{P}}$ is obtained from $\mathcal{P}$ just using logical equivalences. Let us denote $H \leftarrow$ by $H \leftarrow true$. $\mathcal{P}$ is equivalent to $\{\, H \leftarrow false \bigvee_{H \leftarrow \mathcal{B} \in \mathcal{P}} \mathcal{B} \mid$

$H = P(f_1(\vec{x}_1), \ldots, f_n(\vec{x}_n)), P \in \mathcal{P}, (f_1, \ldots, f_n) \in \mathcal{C}^n \}$ i.e. for any possible atom $H = P(f_1(\vec{x}_1), \ldots, f_n(\vec{x}_n))$, we collect the set of bodies of the clauses which have the head H. These bodies are connected by the logical operator or. Notice that since the clauses head are linear the symbol $\leftarrow$ here is indeed an equivalence $\leftrightarrow$. The negation of these formulae are $\{\overline{H} \leftrightarrow \mathit{true} \bigwedge_{H \leftarrow \mathcal{B} \in \mathcal{P}} \neg \mathcal{B}\}$. If one $\mathcal{B}$ is the constant true then $\overline{H}$ is equivalent to false, therefore no clause is headed by $\overline{H}$ in $\overline{\mathcal{P}}$. If the definition is reduced to $H \leftarrow \mathit{false}$ meaning that none of the ground instances of H are in the model of $\mathcal{P}$ then $\overline{H} \leftarrow$ is in $\overline{\mathcal{P}}$, meaning that all ground instances of $\overline{H}$ are in the model of $\overline{\mathcal{P}}$. Finally, in the other cases, $\mathit{true} \bigwedge_{H \leftarrow \mathcal{B} \in \mathcal{P}} \neg \mathcal{B}$ is reduced into a disjunction of conjunctions of negative atoms. Each conjunction is of the form $\neg A_1 \wedge \ldots \wedge \neg A_k$ where each A_i belongs to one different $\mathcal{B}_i$. These definitions produce the clauses $\{\, \overline{Q}(f_1(\vec{x}_1), \ldots, f_n(\vec{x}_n)) \leftarrow \overline{A_1}, \ldots, \overline{A_k} \mid AC(Q, (f_1, \ldots, f_n)) = \{H \leftarrow \mathcal{B}_1, \ldots, H \leftarrow \mathcal{B}_k\}$ and $\forall i \in [1, k] A_i \in \mathcal{B}_i \,\}$ of $\overline{\mathcal{P}}$.

Lemma 2. *Let $\mathcal{P}$ be a shared regular program without any unproductive clauses. The program $\overline{\mathcal{P}}$ is a shared regular program.*

The proof of this lemma is obvious since no substitution but variable renamings is used to define $\overline{\mathcal{P}}$ so the mappings used to check the shared regularity of $\mathcal{P}$ are preserved in $\overline{\mathcal{P}}$.

Theorem 3. *Pseudo-regular relations are closed under complement*

Proof. Let $\mathcal{L}$ be a pseudo-regular relation defined by the PR program $\mathcal{P}$. First compute an equivalent regular program $\mathcal{P}'$ without neither unproductive clause nor free variable in the body of the clauses. $\overline{\mathcal{P}'}$ defines the complement of $\mathcal{L}$ in $\mathcal{T}_{\mathcal{C}}^n$ if $\mathcal{L}$ is of arity n.

For two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ we have $\mathcal{L}_1 \setminus \mathcal{L}_2 = \overline{L_2} \cap \mathcal{L}_1$ therefore we have the following corollary.

Corollary 3. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ two PR programs, $\mathcal{L}_1$ and $\mathcal{L}_2$ two languages defined by the predicates $P_{\mathcal{L}_1}$ of $\mathcal{P}_1$ and $P_{\mathcal{L}_2}$ of $\mathcal{P}_2$. The program that represents the language $\mathcal{L}_1 \setminus \mathcal{L}_2$ is computed in two phases. First compute the program $\overline{\mathcal{P}_2}$ and then compute $\mathcal{P}_1 \cap \overline{\mathcal{P}_2}$ from $\mathcal{P}_1 \cup \overline{\mathcal{P}_2}$ and the definition $P_{\mathcal{L}_1 \setminus \mathcal{L}_2}(\vec{x}) \leftarrow P_{\mathcal{L}_1}(\vec{x}), P_{\overline{\mathcal{L}_2}}(\vec{x})$ using $\Rightarrow$-derivation.*

Notice that the input definition of the second point is a general join definition.

3.3 Non-Greibach logic programs

Even if PR programs are less restrictive than regular ones, their syntax is still very strict. This section aims at defining a class of logic programs that can be transformed into a finite PR one. Our aim is to allow more than one function symbol in the clause heads.

Definition 3. *A Horn clause is called* non-Greibach *(NG for short) if at least one of the arguments of the head is of depth more than one. Let $H \leftarrow \mathcal{B}$ be a clause such that $\mathcal{B}$ contains no function symbols.*

- $H \leftarrow \mathcal{B}$ *is called* NGPR-like *iff none of the arguments of H are variables and there exists a mapping π: $Var \mapsto I\!N^+$ such that $\pi(x) = u$ then all occurrences of x in the arguments of H are at position u and $\pi(x) = \pi(y)$ for all variables x and y occurring in the same body atom.*
- $H \leftarrow \mathcal{B}$ *is called* NGPR *iff it is NGPR-like and $\mathcal{B}$ is linear.*
- $H \leftarrow \mathcal{B}$ *is called* NGR *iff it is NGPR and H is linear*
- $H \leftarrow \mathcal{B}$ *is called* NG-shared-PR *iff it is NGPR-like and it contains no free variable*
- $H \leftarrow \mathcal{B}$ *is called* NG-shared-R *iff it is NG-shared-PR and H is linear.*

A program is NGPR-like, NGPR, R, NG-shared-PR or NG-shared-R if all its clauses are of the corresponding type.

For example, the clause $P(c(s(x), y), s(s(z))) \leftarrow P(x, z), Q(y)$ is NGPR since x and z occur both at occurrence 1.1 in the arguments of the head. (It is more precisely NGR since the head is linear).

Lemma 3. *Any NG-xxx clause $H \leftarrow B$ has an equivalent finite set of xxx clauses (xxx being either PR or shared-PR or R or shared-R).*

Proof. Let $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n})) \leftarrow \mathcal{B}, \mathcal{B}_1 \dot{\cup} \ldots \dot{\cup} \mathcal{B}_k$ be a NGPR-like clause where $\mathcal{B}$ is the set of atoms the variables of which occur all at depth one in the argument of the head and each $\mathcal{B}_i$ is the set of atoms not occurring in $\mathcal{B}$ and the variables of which have an image by π of the form $i.u$ ($u \neq \varepsilon$).

Let us consider the atom $P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n}))$ where $\vec{x_i}|_j = \vec{t_i}|_j$ if $\vec{t_i}|_j$ is a variable and $\vec{x_i}|_j$ is a new variable [1] otherwise. Let σ be such that $P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n}))\sigma = P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$

$P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n})) \leftarrow \mathcal{B}, \mathcal{B}_1 \dot{\cup} \ldots \dot{\cup} \mathcal{B}_k$ is equivalent to the set of clauses $\{P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n})) \leftarrow \mathcal{B}, P_1(\vec{y_1}), \ldots, P_k(\vec{y_k})\} \bigcup_{i \in [1,k]} \{P_i(\vec{y_i})\sigma \leftarrow \mathcal{B}_i\}$ where the P_is are new predicate symbols and $\vec{y_i}$ is the vector of all the variables z occurring at position i in $P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n}))$ and such that $z\sigma \neq z$.

The clause $P(f_1(\vec{x_1}), \ldots, f_n(\vec{x_n})) \leftarrow \mathcal{B}, P_1(\vec{y_1}), \ldots, P_k(\vec{y_k})$ is PR-like by construction and the clauses $P_i(\vec{y_i})\sigma \leftarrow \mathcal{B}_i$ are NGPR-like and have heads of depth strictly inferior to the depth of $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$. Therefore iterating this process, we obtain a set of PR-like clauses. One can remark that

- no free variable is introduced by the process so if $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n})) \leftarrow \mathcal{B}, \mathcal{B}_1 \dot{\cup} \ldots \dot{\cup} \mathcal{B}_k$ is NG-shared-PR, the resulting clauses are all NG-shared-PR
- the new variables introduced by the process are all different, therefore if $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n})) \leftarrow \mathcal{B}, \mathcal{B}_1 \dot{\cup} \ldots \dot{\cup} \mathcal{B}_k$ is NGPR so are the resulting clauses and moreover if $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$ is linear the heads of the new clauses are also linear.

[1] i.e. new introduced variables are pairwise different and also different from the variables of $P(f_1(\vec{t_1}), \ldots, f_n(\vec{t_n}))$.

Example 4. Consider the NGPR-shared-PR clause

$$P_f(s(c(x_1,y_1)), s(c(x_1,y_2)), s(c(x_2,y_3))) \leftarrow P_f(y_1,y_1,y_3), P_g(x_1,x_2)$$

the equivalent set of shared-PR clauses is

$$P_f(s(x), s(y), s(z)) \leftarrow P_1(x,y,z)$$
$$P_1(c(x_1,y_1), c(x_1,y_2), c(x_2,y_3)) \leftarrow P_f(y_1,y_1,y_3), P_g(x_1,x_2)$$

4 Solving pseudo-regular formulae

In this section, we show how our results on pseudo-regular relations can be used in the context of term rewriting theory. The aim is to solve formulae based on joinability equations for the class of pseudo-regular TRS. First we define the class of TRS and the type of formulae we deal with and then we give an algorithm to solve such formulae.

4.1 Pseudo-regular TRS and formulae

The next definition uses the notion of possible redex for constructor based rewrite system[2], i.e. subterms of a term that may be rewritten.

Let R be a TRS and t a term. A position u in t is called a *possible redex position* if $t|_u$ is of the form $f(C_1,\dots,C_k)[f_1(\vec{t_1}),\dots,f_n(\vec{t_n})]$ where $f, f_1, \dots f_n \in \mathcal{F}$ and $C_1,\dots,C_k \in T(\Sigma, \mathit{Var} \cup \mathit{CVar})$. $f(C_1,\dots,C_k)$ is called the *possible redex* at occurrence u of t. The context C that does not contain any possible redex and that is such that $t = C[t_1,\dots,t_n]$ where u_i for $1 \le i \le n$ is a possible redex position, is called the *irreducible part* of t and is denoted $\mathit{Irr}_R(t)$.

Definition 4. *A constructor based TRS R is called pseudo-regular if for all rule $f(t_1,\dots,t_n) \to r \in R$:*

- *there exists a mapping $\pi\colon \mathit{Var} \mapsto \mathbb{N}^+$, such that $\pi(x) = u$ implies that all occurrences of x in each $t_1,\dots,t_n$ and $Irr(r)$ are at position u,*
- *each possible redex C at position u of r is of the form $f(x_1,\dots,x_n)$ where $\forall 1 \le i \le n\ x_i \in Var(f(t_1,\dots,t_n))$ and $\pi(x_i) = u$.*

Example 5. $R = \{f(s(c(x,y)), s(c(x,z))) \to p(c(g(x), f(y,y))), f(s(0), s(0)) \to 0, g(s(x)) \to s(x)\}$ is pseudo-regular. The irreducible part of the lhs of the first rule is $p(c(\square_1,\square_2))$ and it contains two possible redex positions namely 1.1 and 1.2 corresponding to the possible redexes $g(x)$ and $f(y,y)$. Notice that Definition 4 does not forbid duplicated variables in a single possible redex.

Definition 5. *Let R be a pseudo regular TRS. A pseudo-regular joinability equation $s \downarrow_R^? t$ is an equation such that s and t are built over $T(\mathcal{F}, \mathit{Var})$.*

[2] A similar notion was introduced in [7] for general TRS

Let R be a pseudo-regular TRS, pseudo-regular R-formulae are defined by the following grammar: $e ::= s \downarrow_R^? t | \neg e | e \vee e | e \wedge e | \exists x e | \forall x e$
where $s \downarrow_R^? t$ is a pseudo-regular joinability equation.

The set of solutions of such a formula is defined as follows:
$SOL(s \downarrow_R^? t)$ is the set of data-solutions of $s \downarrow_R^? t$
$SOL(\neg e) = \{ \sigma \mid \sigma \notin SOL(e) \}$
$SOL(e_1 \wedge e_2) = \{ \sigma \mid Dom(\sigma) = Var(e_1 \wedge e2), \sigma|_{Var(e_1)} \in SOL(e_1), \sigma|_{Var(e_2)} \in SOL(e_2) \}$
$SOL(e_1 \vee e_2) = \{ \sigma \mid Dom(\sigma) = Var(e_1 \vee e_2), \sigma|_{Var(e_1)} \in SOL(e_1)$ or $\sigma|_{Var(e_2)} \in SOL(e_2) \}$
$SOL(\exists x e) = \{ \sigma \mid Dom(\sigma) = Var(e) \setminus \{x\}, \exists \sigma' \in SOL(e), \sigma = \sigma'|_{Var(e) \setminus \{x\}} \}$
$SOL(\forall x e) = \{ \sigma \mid Dom(\sigma) = Var(e) \setminus \{x\}, \forall \sigma' \in SOL(e), \sigma = \sigma'|_{Var(e) \setminus \{x\}} \}$

Even if the constructors are forbidden in the joinability equations, many wanted properties for TRS can be expressed by a pseudo-regular formula. Among them we can cite

- confluency $\forall x, y \neg(f(x,y) \downarrow_R^? z \wedge f(x,y) \downarrow_R^? z') \vee z \downarrow_R^? z'$,
- commutativity $\forall x, y f(x,y) \downarrow_R^? f(y,x)$,
- associativity $\forall x, y, z f(x, f(y,z)) \downarrow_R^? f(f(x,y),z)$,
- idempotency $\forall x g(g(x)) \downarrow_R^? x$,
- fixed points of a function $g(x) \downarrow_R^? x$.
- search for neutral elements $\forall x f(x,y) \downarrow_R^? x$.

4.2 Solving Pseudo-regular Formulae

Our algorithm is based on the results given in Section 3. The first step consists in computing the solutions of a pseudo-regular joinability equation. For that we use a technique introduced in [7] that encodes the rewrite relation by a logic program. This translation intends to obtain logic programs that preserve as best as possible syntactic properties of the TRS. It works for any term rewrite system but the obtained logic program encodes only a subset of the rewriting relation called basic rewriting. Fortunately, basic rewriting and rewriting relations coincide when considering only data-steps.

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 $f(s_1, \dots, s_n)$ at occurrence u of a rhs s and A_u the atom produced by this rule.

For example, the first rewrite rule of Example 5 is transformed into
$$P_f(s(c(x,y)), s(c(x,z)), p(c(x_1,x_2))) \leftarrow P_g(x, x_1), P_f(y, y, x_2).$$

Theorem 4. *Let R be a TRS, s a term such that $s \leadsto \langle s', \mathcal{G} \rangle$. $s \rightarrow^* t$ iff t is a data-term $\mathcal{LP}(R), \models \mathcal{G}\mu$ and $t = s'\mu$ where μ is a data substitution.*

The rules of Table 1 slightly differ from those presented in [7] because [7] deals with general TRS whereas we are working on constructor based TRS in

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

$$\frac{s_1 \rightsquigarrow \langle t_1, \mathcal{G}_1 \rangle \dots s_n \rightsquigarrow \langle t_n, \mathcal{G}_n \rangle}{f(s_1, \dots, s_n) \rightsquigarrow \langle f(t_1, \dots, t_n), \bigcup_i \mathcal{G}_i \rangle} \quad \text{if } f \in \mathcal{C}$$

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

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

Table 1. Converting rewrite rules to clause logic

the present paper. Theorem 4 in [7] is also a little bit different because in [7] the set of clauses $\mathcal{P}_{Id}$ is added to $\mathcal{LP}(R)$ to be able to stop rewrite derivation at any step. For computing data-solutions of a joinability equation, we need to compute rewrite derivation until reaching a data term so $\mathcal{P}_{Id}$ has been removed.

Lemma 4. *Let R be a pseudo-regular TRS then $\mathcal{LP}(R)$ is a NG-shared-PR logic program.*

Proof. Each rule of a pseudo regular TRS is of the form

$$f(t_1, \dots, t_n) \to C[f_1(\vec{x}_1), \dots, f_k(\vec{x}_k)]$$

where C is the irreducible part of the lhs of the rule, and $f, f_1, \dots, f_k$ are function symbols. C and each t_i contain no defined function symbols. Therefore the clause produced for this rule is

$$P_f(t_1, \dots, t_n, C[x_{u_1} \dots, x_{u_k}]) \leftarrow P_{f_1}(\vec{x}_1, x_{u_1}), \dots, P_{f_k}(\vec{x}_k, x_{u_k})$$

Since all variables of $f_1(\vec{x}_1), \dots, f_k(\vec{x}_k)$ are variables of $f(t_1, \dots, t_n)$, the clause contains no free variable. By extending the mapping π of the pseudo-regular rewrite rule $f(t_1, \dots, t_n) \to C[f_1(\vec{x}_1), \dots, f_k(\vec{x}_k)]$ to $\pi' = \pi \cup_{1 \le i \le k} \{x_{u_i} \mapsto u_i\}$ we obtain that the clause is NG-shared-PR.

The data solutions of pseudo-regular joinability equations expressed as a pseudo-regular language can be computed using the following algorithm.

Algorithm 1 *Let R be a pseudo-regular TRS and $s \downarrow_R^? t$ a pseudo-regular joinability equation, $\vec{x}$ the variables occurring in $s \downarrow_R^? t$. Let $\mathcal{LP}(s \downarrow_R^? t) = Id_2(x_s, x_t), G_s, G_t$ where $s \rightsquigarrow (x_s, G_s)$ and $t \rightsquigarrow (x_t, G_t)$ (see Table 1) and Id_2 is a pseudo regular predicate that define equality of terms like in Example 1.*

1. *Compute $\mathcal{LP}(R)$*
2. *Compute $\mathcal{LP}(s \downarrow_R^? t)$*
3. *Use $\Rightarrow$ and Lemma 3 to compute a pseudo-regular program $\mathcal{P}$ from $\mathcal{LP}(R)$.*
4. *Use $\Rightarrow$ to compute from $\mathcal{P}$ and the join definition $P_{s\downarrow_R^? t, \vec{x}}(\vec{x}) \leftarrow \mathcal{LP}(s \downarrow_R^? t)$ the PR program $\mathcal{P}_{s\downarrow_R^? t}$.*

Theorem 5. *Let R be a pseudo-regular TRS and $s \downarrow_R^? t$ a pseudo-regular joinability equation and $\mathcal{P}_{s\downarrow_R^? t}$ the resulting program of Algorithm 1. $\mathcal{P}_{s\downarrow_R^? t} \models P_{s\downarrow_R^? t,\vec{x}}(\vec{t})$ iff $\{ \vec{x}|_i \mapsto \vec{t}|_i \mid 1 \leq i \leq length(\vec{x}) \} \in SOL(s \downarrow_R^? t)$.*

Proof. From Lemma 4, we know that $\mathcal{LP}(R)$ is a NG-shared-PR program. According to Lemma 3 and Theorem 2, point 3 of Algorithm 1 produces a finite PR program that is equivalent to $\mathcal{LP}(R)$. The definition $P_{s\downarrow_R^? t,\vec{x}}(\vec{x}) \leftarrow \mathcal{LP}(s \downarrow_R^? t)$ is a general join definition since all predicates of $\mathcal{LP}(s \downarrow_R^? t)$ are defined by a PR program and $\mathcal{LP}(s \downarrow_R^? t)$ contains no function symbols since s and t contain no constructor symbol. Therefore, from Theorem 1, the point 4 of the algorithm produces a finite PR program.

From Theorem 4 we know that for any predicate P_f of $\mathcal{LP}(R)$, $\mathcal{LP}(R) \models P_f(t_1, \ldots, t_n, t)$ iff $f(t_1, \ldots, t_n) \rightarrow^* t$. $P_f(\ldots, x_u, \ldots)$ and $P_g(\vec{x}, x_u)$ belong to $\mathcal{LP}(s \downarrow_R^? t)$ iff either s or t contains the subterm $f(\ldots, g(\vec{t}), \ldots)$ therefore $\mathcal{LP}(s \downarrow_R^? t)$ simulate the composition of functions contained in s and t. Moreover the atom $Id_2(x_s, x_t)$ insures that the respective results of the rewrite derivations from s and t are equal.

Solving pseudo-regular formulae can be done using the following algorithm

Algorithm 2 *Let R be a pseudo-regular TRS and e a pseudo-regular formula.*

1. *Solve each elementary formula $s \downarrow_R^? t$ by computing $\mathcal{P}_{s\downarrow_R^? t}$*
2. *If $\mathcal{P}_e$ defines $SOL(e)$ then $\overline{\mathcal{P}_e} = \mathcal{P}_{\neg e}$ defines $SOL(\neg e)$.*
3. *If $\mathcal{P}_{e_1}$ and $\mathcal{P}_{e_2}$ define respectively $SOL(e_1)$ and $SOL(e_2)$ then the program $\mathcal{P}_{e_1 \wedge e_2}$ obtained by a complete derivation of $\Rightarrow$ with $\mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}$ and $P_{e_1 \wedge e_2,\vec{x}}(\vec{x}) \leftarrow P_{e_1,\vec{x}_1}(\vec{x}_1), P_{e_2,\vec{x}_2}(\vec{x}_2)$ where $\vec{x} = Var(e_1 \wedge e_2)$ as inputs, is a PR program that defines $SOL(e_1 \wedge e_2)$*
4. *If $\mathcal{P}_{e_1}$ and $\mathcal{P}_{e_2}$ define respectively $SOL(e_1)$ and $SOL(e_2)$ then the program $\mathcal{P}_{e_1 \vee e_2}$ obtained by a complete derivation of $\Rightarrow$ with $\mathcal{P}_{e_1} \cup \mathcal{P}_{e_2}$ and $\{P_{e_1 \vee e_2,\vec{x}}(\vec{x}) \leftarrow P_{e_1,\vec{x}_1}(\vec{x}_1), P_{e_1 \vee e_2,\vec{x}}(\vec{x}) \leftarrow P_{e_2,\vec{x}_2}(\vec{x}_2)\}$ where $\vec{x} = Var(e_1 \wedge e_2)$ as inputs, is a PR program that defines $SOL(e_1 \vee e_2)$.*
5. *If $\mathcal{P}_e$ defines $SOL(e)$ then the program $\mathcal{P}_{\exists xe}$ obtained from $\mathcal{P}_e$ by a complete $\Rightarrow$-derivation with $P_{\exists xe,\vec{y}}(\vec{y}) \leftarrow P_{e,\vec{x}'}(\vec{x}')$ as input definition is a PR program that defines $SOL(\exists xe)$ since it is a projection on all arguments of $P_{e,\vec{x}'}$ but the one defining the variable x.*

Solving a $\forall xe$ can be done by using well-known identity $\forall xe \equiv \neg \exists \neg e$.

Theorem 6. *Let R be a pseudo-regular TRS and e a pseudo-regular formula and $\mathcal{P}_e$ the resulting program of Algorithm 2. $\mathcal{P}_e \models P_{e,\vec{x}}(\vec{t})$ iff $\{ \vec{x}|_i \mapsto \vec{t}|_i \mid 1 \leq i \leq length(\vec{x}) \} \in SOL(e)$.*

Proof. It is done by an easy induction on the structure of the formula. From Theorem 5, we know that for each elementary formula $s \downarrow_R^? t$, $\mathcal{P}_{s\downarrow_R^? t} \models P_{s\downarrow_R^? t,\vec{x}}(\vec{t})$ iff $\{ \vec{x}|_i \mapsto \vec{t}|_i \mid 1 \leq i \leq length(\vec{x}) \} \in SOL(s \downarrow_R^? t)$. Let e_1 and e_2 be two formulae such that $\mathcal{P}_{e_1}$ and $\mathcal{P}_{e_2}$ computes the solutions of e_1 and e_2 respectively.

– From Theorem 3, we know that $\mathcal{P}_{\neg e_1}$ defines $SOL(\neg e_1)$ and is regular.
– $P_{e_1 \wedge e_2, \vec{x}}(\vec{x}) \leftarrow P_{e_1,\vec{x}_1}(\vec{x}_1), P_{e_2,\vec{x}_2}(\vec{x}_2)$ is a general join definition since $P_{e_1,\vec{x}_1}$ and $P_{e_2,\vec{x}_2}$ are defined by a PR program, therefore from Theorem 1 $\mathcal{P}_{e_1 \wedge e_2}$ is a PR program that defines the solutions of $e_1 \wedge e_2$.
– $P_{e_1 \vee e_2, \vec{x}}(\vec{x}) \leftarrow P_{e_1,\vec{x}_1}(\vec{x}_1)$ and $P_{e_1 \vee e_2, \vec{x}}(\vec{x}) \leftarrow P_{e_2,\vec{x}_2}(\vec{x}_2)$ are general join definitions since $P_{e_1,\vec{x}_1}$ and $P_{e_2,\vec{x}_2}$ are defined by a PR program, therefore from Theorem 1 $\mathcal{P}_{e_1 \wedge e_2}$ is a PR program that defines the solutions of $e_1 \vee e_2$.
– $P_{\exists x e_1, \vec{y}}(\vec{y}) \leftarrow P_{e_1,\vec{x}'}(\vec{x}')$ is a general join definition since $P_{e_1,\vec{x}'}(\vec{x}')$ is defined by a PR program, therefore from Theorem 1 $P_{\exists x e_1, \vec{y}}$ is defined by a PR program. Moreover $P_{\exists x e_1, \vec{y}}(\vec{y})$ defines the solutions of $\exists x e_1$ since $P_{e_1,\vec{x}'}(\vec{x}')$ defines $SOL(e_1)$ and therefore $P_{\exists x e_1, \vec{y}}(\vec{y})$ defines $SOL(e_1)|_{\{}x\}$.

5 Conclusion and future work

The main contributions of this paper are the definition of an algorithm to compute the complement of a pseudo-regular relation represented by a logic program and the use of pseudo-regular relations to solve pseudo-regular formulae. The representation of tree tuple languages by logic programs simplifies the connection between term rewriting systems and tree languages. Indeed, it would have been quite difficult to obtain the definition of pseudo regular TRS and pseudo-regular formulae without NG-Shared-PR logic programs.

Since we know other classes of logic programs that can be transformed into a finite pseudo-regular one, allowing some function symbols in the body of the clauses, it would be possible to weaken some restrictions on the possible redexes of the pseudo-regular TRS as well as on the pseudo-regular joinability equations.

We should study the actual expressiveness of pseudo-regular TRS by encoding some problems coming from fields like theorem proving or system verification. For example, we already have encoded the semantics of the so-called process algebra of [10] by a regular TRS. We intend to study what pseudo-regularity can bring to the model checking of process algebra.

It would be also interesting to implement the resolution of pseudo-regular formulae. A prototype should be soon available since $\Rightarrow$ and $\leadsto$ (the transformation of a TRS into a logic program) have already been implemented in Prolog.

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. H. Comon, M. Haberstrau, and J.-P. Jouannaud. Syntacticness, cycle-syntacticness, and shallow theories. *Information and Computation*, 111(1):154–191, 1994.

4. Heinz Faßbender and Sebastian Maneth. A strict border for the decidability of e-unification for recursive functions. *Journal of Functional and Logic Programming*, 1998(4), 1998.
5. S. Limet and P. Réty. E-unification by means of tree tuple synchronized grammars. *Discrete Mathematics and Theoretical Computer Science*, 1:69–98, 1997.
6. 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.
7. S. Limet and G. Salzer. Proving properties of term rewrite systems via logic programs. In V. van Oostrom, editor, *Proc. 15th Int. Conf. on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *LNCS*, pages 170–184. Springer, 2004.
8. S. Limet and F. Saubion. A general framework for R-unification. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *proc of the Conf. on Principle of Declarative Programming (PLILP-ALP)*, volume 1490 of *LNCS*, pages 266–281. Springer, 1998.
9. J.W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.
10. D. Lugiez and Ph. Schnoebelen. The regular viewpoint on PA-processes. *Theoretical Computer Science*, 274(1-2):89–115, 2002.
11. F. Nielson, H. Riis Nielson, and H. Seidl. Normalizable horn clauses, strongly recognizable relations and spi. In *Proc. SAS'02*, number 2477 in Lecture Notes in Computer Science, pages 20–35. Springer-Verlag, 2002.
12. Robert Nieuwenhuis. Decidability and complexity analysis by basic paramodulation. *Information and Computation*, 147:1–21, 1998.
13. 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.
14. P. Réty. Regular sets of descendants for constructor-based rewrite systems. In *Proc. of the 6th conference LPAR*, number 1705 in LNAI. Springer, 1999.
15. T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *Proc. 11th Int. Conf. on Rewriting Techniques and Applications (RTA'00)*, volume 1833 of *LNCS*, pages 270–273. Springer, 2000.