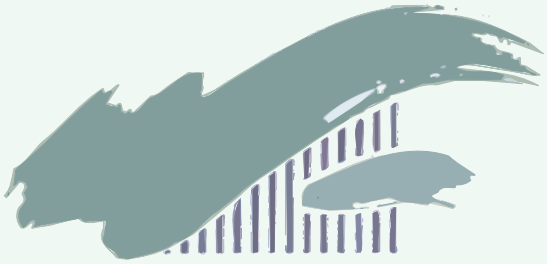

UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

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/

Proving Properties of Term Rewrite Systems via Logic Programs

Sébastien Limet et Gernot Salzer (TU Wien)
Université d'Orléans, LIFO

Rapport N^o **2004-05**

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

First-order terms and term tuples (also called tree tuples) are the basic data structure in many areas of logic and computer science. What integers and reals are to numeric computing, terms are to formal verification, automated deduction, logic programming, and many other fields. Usually we are confronted not just with single terms or tree tuples but with infinite sets thereof when e.g. describing models in automatic deduction and logic programming, approximating calculi, or detecting infinite loops.

For theoretical and practical reasons we are interested in finite presentations of these infinite sets and in ways to manipulate the finite presentations in place of the infinite sets. Typical problems to solve are

- how to construct finite representations from the initial problem either statically by syntactic transformations, or dynamically by providing e.g. loop detection mechanisms during a computation, and
- how to perform operations like union or intersection on finite representations and how to test for properties like membership or emptiness.

The main difficulty is to find a good balance between the expressiveness of the chosen formalism and the tractability of the operations defined on it.

A prominent example are tree automata which define the class of regular tree languages (tree here means ground term). They are closed under standard set operations like union, intersection, and complement, both the membership and the emptiness problem are decidable, and the computational complexity of these operations is low. Therefore tree automata are widely used and have found applications in all areas mentioned above [4, 19]. The drawback of regular tree

languages is their weak expressiveness, which led to extensions like tree automata with constraints [5], tree set automata [11], and regular relations [4, 20].

Tree automata and tree grammars can also be used to handle tree tuple languages (or relation languages). In [4], tree tuples are encoded by overlapping all components of the tuple within a single tree (Fig. 1) and by considering

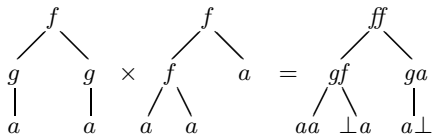

Fig. 1. Encoding the tuple $(f(g(a), g(a)), f(f(a, a), a))$ as a single term

automata over the domain $T_{(\Sigma\cup\bot)^n}$. For instance, the tree tuple language $Id = \{\, (t, t) \mid t \in T_{\{f,a\}} \,\}$ is recognized by a tree automaton with transitions $aa \to q$ and $ff(q, q) \to q$, where q is the final state. This approach preserves the well-known results from tree automata theory, but it does not work beyond regularity. [25] shows that virtually all extensions of relation automata with constraints have an undecidable emptiness test.

Formalisms extending tree grammars were introduced in [17, 22]. There, tree tuple languages are generated from a tuple axioms by applying sets of productions simultaneously. For example, the language Id is generated from the axiom (X, Y) by a grammar that contains two sets of productions, $\{X \Rightarrow f(X, X), Y \Rightarrow f(Y, Y)\}$ and $\{X \Rightarrow a, Y \Rightarrow a\}$. An additional mechanism synchronizes the replacement of non-terminals in different positions. Such Tree Tuple Synchronized Grammars turned out to be powerful but unwieldy. Therefore grammar productions were subsequently replaced by set constraints [14], leading to so-called *constraint systems*. Synchronized languages are able to represent finitely the solutions of certain R-unification problems and can be applied to R-disunification and one-step-rewriting [23]. Moreover, synchronized languages are able to express the transitive closure of relations induced by certain process algebras with communications and so can be used for model checking in this context [14]. In general, synchronized languages do not possess the same nice properties as regular tree languages do, but they are considerably more expressive, and some relevant sub-classes are closed under operations like join or projection which are sufficient for many applications.

In this paper we go beyond constraint systems and introduce *inductive definitions over language expressions* (Section 2). Language expressions specify tuple languages by the usual set operations and by two term tuple operations, construction and filtering. Construction builds more complicated tuples out of simpler ones, while filtering selects certain tuples from a language. After fixing some notations from logic programming in Section 3, we translate inductive definitions to Horn logic and vice versa in Section 4. This translation allows us to deal with constraint systems in a uniform framework using notions and nota-

tions from clause logic. Section 5 defines a rule system that transforms any logic program into a – not necessarily finite – cs-program, the Horn-equivalent of a constraint system. Section 6 identifies several classes of programs for which the derivation process provably terminates, leading to new decidability results for tree tuple languages. We show in particular that the emptiness test for linear filter operations (a generalization of the membership test) is decidable for all languages definable by constraint systems. Moreover, we introduce pseudo-regular constraint systems and show that the corresponding tuple languages are closed under intersections, joins, and self-joins. To our knowledge this is the first class of non-linear constraint systems with such properties.

2 Tree tuple languages

Let Σ be a finite set of symbols with arities, and let T_Σ denote the set of all ground terms over Σ. The subsets of ${T_\Sigma}^n$ are called *n-ary tree tuple languages over Σ* (n-tuple languages for short). Union, intersection, and Cartesian product are denoted by $\cup$, $\cap$, and $\times$, respectively. We regard $A \times (B \times C)$, $(A \times B) \times C$, and $A \times B \times C$ as equivalent and omit parentheses; the product of two tuple languages of arities m and n thus results in a set of $(m+n)$-tuples. The notation $A_1 \times \cdots \times A_k$ is shorthand for ${T_\Sigma}^0 = \{()\}$ (the set containing just the zero-tuple) if $k = 0$, for A_1 if $k = 1$, and for $A_1 \times (A_2 \times \cdots \times A_k)$ otherwise.

A *template* is a ground term that additionally may contain positive integers as constant symbols. Formally, if ω denotes the set of positive integers, then $T_{\Sigma \cup \omega}$ is the set of templates over Σ.[3] The integers are called *indices* and are used to refer to particular components of tuple languages. The arity of an object O is denoted by $\mathrm{ar}(O)$, where O may be a function or predicate symbol, a language variable, or a language expression. As usual we denote substitutions as sets, i.e., $O\{s_1 \mapsto t_1, \ldots, s_l \mapsto t_l\}$ is obtained from O by replacing within O simultaneously all occurrences of s_i by t_i, where O denotes any syntactic entity.

Let A be an n-tuple language, and let $\Box$ be a k-tuple of templates with l denoting the largest index occurring in $\Box$, or 0 if there is none. The operations *construction* and *filtering* are defined as:

$$\Box \circ A \stackrel{\text{def}}{=} \{\, \Box\{1 \mapsto t_1, \ldots, l \mapsto t_l\} \mid (t_1, \ldots, t_{n+l}) \in A \times {T_\Sigma}^l \,\}$$

$$A/\Box \stackrel{\text{def}}{=} \{\, (t_1, \ldots, t_l) \in {T_\Sigma}^l \mid \Box\{1 \mapsto t_1, \ldots, l \mapsto t_l\} \in A \,\}$$

Note that the construction operator replaces indices in $\Box$ that exceed the arity of A by arbitrary ground terms. The l-fold product of T_Σ ensures that the arity of $A \times {T_\Sigma}^l$ is not smaller than l, even if A is a nullary language. To some extent construction and filtering are inverse to each other: if the arity of A equals the largest index in $\Box$, i.e. $n = l$, and if $\Box$ contains all indices up to l then $(\Box \circ A)/\Box = A$.

[3] For the sake of notational convenience we do not distinguish between integers and the symbols denoting them. Thus in a given context a variable i may represent a symbol when occurring in a formal entity like a template and at the same time the corresponding integer when occurring in a mathematical expression.

Example 1. Let $\Sigma = \{a/0, f/1, g/2\}$, $A = \{(a, f(a)), (f(a), a)\}$, and let $\square = [g(1,3), 2]$ be a pair of templates. We obtain:

$$\begin{aligned} \square \circ A &= \{\, (g(a,t), f(a)) \mid t \in T_\Sigma \,\} \cup \{\, (g(f(a),t), a) \mid t \in T_\Sigma \,\} \\ A/\square &= \emptyset \\ (\square \circ A)/\square &= \{\, (a, f(a), t) \mid t \in T_\Sigma \,\} \cup \{\, (f(a), a, t) \mid t \in T_\Sigma \,\} \end{aligned}$$

To specify interesting relations like one-step rewriting or the semantics of process algebras by term tuple languages we need some iterative or recursive mechanism. We propose inductive definitions with subset relations over language expressions as general framework. Let $\mathcal{X}$ be a set of language variables, each supplied with an arity. The set of *language expressions* is defined by the grammar

$$e ::= A \mid \{()\} \mid (e \times e) \mid (\square \circ e) \mid (e/\square)$$

for language variables A and template tuples $\square$.

An interpretation λ is a mapping associating an n-ary tuple language with every n-ary language variable. It extends naturally to an interpretation for language expressions: $\lambda(\{()\}) = \{()\}$, $\lambda(e_1 \times e_2) = \lambda(e_1) \times \lambda(e_2)$, $\lambda(\square \circ e) = \square \circ \lambda(e)$, and $\lambda(e/\square) = \lambda(e)/\square$. An interpretation, λ, is smaller or equal than another one, λ', if $\lambda(A) \subseteq \lambda'(A)$ for all $A \in \mathcal{X}$.

An *inductive definition* is a set of constraints of form $A \supseteq e$ such that the arities of A and e coincide.[4] The languages defined by an inductive definition D are given by the smallest interpretation λ such that $\lambda(A) \supseteq \lambda(e)$ for all constraints; it is uniquely defined and can be obtained as the least fixed point of the constraints. We denote the smallest interpretation by $\mathcal{L}_\mathcal{D}$, or just by $\mathcal{L}$ if the inductive definition is clear from context.

Example 2. Operations on term tuples like forming sets of term tuples, computing union, intersection, joins, or projections can be expressed by proper language expressions, and membership tests can be reduced to emptiness tests:

$$\begin{aligned} \{\boldsymbol{t}_1, \ldots, \boldsymbol{t}_n\} &= \boldsymbol{t}_1 \circ \{()\} \cup \cdots \cup \boldsymbol{t}_n \circ \{()\} \\ e \cup f &= A \qquad \text{with the constraints } A \supseteq e \text{ and } A \supseteq f \\ e \cap f &= (e \times f)/[1, \ldots, m, 1, \ldots, n] \qquad \text{for } m = n \\ e \bowtie_{i,j} f &= (e \times f)/[1, \ldots, m+j-1, i, m+j, \ldots, m+n-1] \\ e \bowtie_{i,n} f &= (e \times f)/[1, \ldots, m+n-1, i] \\ \Pi_{i_1,\ldots,i_k} e &= [i_1, \ldots, i_k] \circ e \\ \boldsymbol{t} \in \mathcal{L}(A) &\equiv \mathcal{L}(A') \neq \emptyset \qquad \text{with the constraint } A' \supseteq A/\boldsymbol{t} \end{aligned}$$

where $\boldsymbol{t}, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_n$ represent term tuples or template tuples without indices, e and f are m- and n-ary language expressions, respectively, and $1 \leq i, i_1, \ldots, i_k \leq m$, $1 \leq j < n$.

[4] The arity of a language expression is the number of components in the tuples of the language defined by it; it can be determined in a straightforward manner since the arity of language variables is given.

Example 3. Let $\Sigma = \{a/0, f/2\}$ and $\mathcal{X} = \{Id_2/2, Sym/2\}$. The constraints

$$Id_2 \supseteq \{(a,a)\} \cup [f(1,3), f(2,4)] \circ (Id_2 \times Id_2)$$
$$Sym \supseteq \{(a,a)\} \cup [f(1,3), f(4,2)] \circ (Sym \times Sym)$$

define the languages

$$\mathcal{L}(Id_2) = \{\, (t,t) \mid t \in T_\Sigma \,\}$$
$$\mathcal{L}(Sym) = \{\, (t,t') \mid t \in T_\Sigma,\ t' \text{ is the symmetric tree of } t \,\} \ .$$

The languages definable by inductive definitions are exactly the recursively enumerable tuple languages; there is a direct translation of constraints to Horn clauses and vice versa (see Section 4 below). The drawback of such expressiveness is that properties like emptiness and membership tests get undecidable. Therefore we restrict the formalism of inductive definitions to single out tractable subclasses.

Constraint systems were introduced in [14] as a simplified version of tree-tuple synchronized grammars. They can be viewed as inductive definitions without filter operations. In this case all constraints can be normalized to the form $A \supseteq \square \circ (A_1 \times \cdots \times A_k)$ for language variables $A, A_1, \ldots, A_k$. For constraint systems it is convenient to write indices as pairs of integers, $i.j$, referring to the j-th component of A_i; an index l exceeding the arity of $A_1 \times \cdots \times A_k$, say m, is written as $(k + l - m).1$. A constraint $A \supseteq \square \circ (A_1 \times \cdots \times A_k)$ is called

- *linear* iff no index occurs twice in $\square$;[5]
- *horizontal* iff for any two indices $i.j$ and $i'.j'$ in $\square$, $i = i'$ implies that $i.j$ and $i'.j'$ occur in $\square$ at positions of the same depth;
- *pseudo-regular* iff each component of $\square$ is of the form $f(i_1.j_1, \ldots, i_{\mathrm{ar}(f)}.j_{\mathrm{ar}(f)})$ and there exists a mapping $\pi{:}\,\omega \mapsto \omega$ such that $\pi(i_l) = l$ for $l = 1, \ldots, \mathrm{ar}(f)$.
- *regular* iff it is pseudo-regular and linear.

A constraint system is called linear (horizontal, regular, pseudo-regular) iff all constraints have the respective property.

The condition for pseudo-regularity essentially requires that the language variables A_i can be partitioned into disjoint groups such that there is a 1-1 correspondence between the argument positions of the function symbols and these groups. Obviously regularity implies horizontality and, by definition, linearity. Pseudo-regularity is more liberal than regularity by admitting certain duplicate indices.

Example 4. The constraint system defining Id_2 in Example 3 is regular and therefore linear, horizontal, and pseudo-regular. The constraint system for Sym is horizontal and linear but neither regular nor pseudo-regular; note that $1 = 1.1$, $2 = 1.2$, $3 = 2.1$, and $4 = 2.2$. The constraint

$$X \supseteq [f(1.1, f(2.1, 2.1)), f(1.2, 2.2)] \circ (X \times X)$$

[5] In [14], this property is called *non-copying*.

is neither horizontal (index 2.1 occurs in depth 2, whereas index 2.2 occurs in depth 1) nor linear (index 2.1 occurs twice). The constraint

$$X \supseteq [f(1.1, 2.1), f(1.1, 2.2)] \circ (X \times X)$$

is pseudo-regular but not regular since index 1.1 appears twice at position 1 of the two components. Note that the definition does not require that duplicate indices appear in all components.

3 Logic programming

We assume the reader to be familiar with the basics of logic programming, in particular with the operational and declarative semantics of definite logic programs (see e.g. [18]). We give a few definitions to fix notation.

Let $T_{\Sigma,V}$ denote the set of first-order terms over a signature Σ and over an infinite set of first-order variables, V. If P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an atomic formula (*atom* for short). The set of variables occurring in an object O is denoted by $\mathrm{var}(O)$, where O may be a term, an atom, or a set of terms or atoms. An object O is called *linear* if no variable occurs more than once in it. The *domain* of a substitution σ, denoted by $\mathrm{dom}(\sigma)$, is the set of variables changed by σ, i.e., $\mathrm{dom}(\sigma) = \{\, x \in V \mid x\sigma \neq x \,\}$. The *range* of a substitution σ, denoted by $\mathrm{ran}(\sigma)$, is the set of terms introduced by σ, i.e., $\mathrm{ran}(\sigma) = \{\, x\sigma \mid x \in \mathrm{dom}(\sigma) \,\}$.

Let H be an atom and $\mathcal{B}$ be a finite set of atoms. The expression $H \leftarrow \mathcal{B}$ is called *program clause*, with H being the *head* and $\mathcal{B}$ the *body* of the clause; for $\mathcal{B} = \emptyset$ the clause is called a *fact*. A *logic program* is a finite set of program clauses. A *query* is an expression of the form $\leftarrow \mathcal{B}$, with $\mathcal{B}$ being the *body* of the query. Program clauses and queries are summarized as *Horn clauses*.

The depth of a variable is 0, the depth of a constant is 1, and the depth of a functional term is the maximal depth of its arguments plus 1. The depth of an atom is the maximal depth of its arguments.

The least Herbrand model of a program $\mathcal{P}$ is denoted by $\mathcal{M}(\mathcal{P})$. The set of term tuples, for which a predicate P is true in $\mathcal{M}(\mathcal{P})$, is defined as

$$\mathcal{M}(\mathcal{P})|_P \stackrel{\mathrm{def}}{=} \{\, (t_1, \ldots, t_n) \mid P(t_1, \ldots, t_n) \in \mathcal{M}(\mathcal{P}) \,\} \ .$$

4 Inductive definitions vs. Horn logic

In this section we define translations from language expressions to semantically equivalent Horn clauses and vice versa. This enables us to discuss closure properties and decidability issues of constraint systems in a purely clausal setting and to use results from the areas of logic program transformations and clausal theorem proving.

For any language variable A of arity n, let P_A denote an n-ary predicate symbol uniquely associated with A. The clause corresponding to a constraint is

defined as

$$\mathrm{horn}(A \supseteq e) \stackrel{\text{def}}{=} \begin{cases} \{P_A(s_1,\ldots,s_n) \leftarrow \mathcal{G}\} & \text{if } e \rightsquigarrow \langle (s_1,\ldots,s_n),\, \mathcal{G}\rangle \\ \{\} & \text{otherwise} \end{cases}$$

Relation $\rightsquigarrow$ is specified in Figure 2. For an inductive definition $\mathcal{D}$, i.e., for a set of constraints, we define the corresponding logic program as $\mathrm{horn}(\mathcal{D}) \stackrel{\text{def}}{=} \bigcup_{C\in\mathcal{D}} \mathrm{horn}(C)$.

$$\{()\} \rightsquigarrow \langle (),\, \{\}\rangle$$

$$A \rightsquigarrow \langle (x_1,\ldots,x_{\mathrm{ar}(A)}),\, \{P_A(x_1,\ldots,x_{\mathrm{ar}(A)})\}\rangle$$
where $x_1,\ldots,x_{\mathrm{ar}(A)}$ are pairwise distinct variables.

$$\frac{e \rightsquigarrow \langle (s_1,\ldots,s_m),\, \mathcal{G}\rangle \quad f \rightsquigarrow \langle (t_1,\ldots,t_n),\, \mathcal{H}\rangle}{(e \times f) \rightsquigarrow \langle (s_1,\ldots,s_m,t_1,\ldots,t_n),\, \mathcal{G}\cup\mathcal{H}\rangle}$$
provided the premises share no variables.

$$\frac{e \rightsquigarrow \langle (s_1,\ldots,s_m),\, \mathcal{G}\rangle}{(\Box \circ e) \rightsquigarrow \langle \Box\{1\mapsto s_1,\ldots,m\mapsto s_m,\, m{+}1\mapsto x_1,\, m{+}2\mapsto x_2,\ldots\},\, \mathcal{G}\rangle}$$
where $x_1, x_2, \ldots$ are pairwise distinct variables not occurring in the premise.

$$\frac{e \rightsquigarrow \langle (s_1,\ldots,s_m),\, \mathcal{G}\rangle}{(e/\Box) \rightsquigarrow \langle (x_1,\ldots,x_l)\mu,\, \mathcal{G}\mu\rangle}$$
provided μ, the m.g.u. of $(s_1,\ldots,s_m)$ and $\Box\{1\mapsto x_1,\ldots,l\mapsto x_l\}$, exists, where l denotes the maximal index occurring in $\Box$ or 0 if there is none, and $x_1,\ldots,x_l$ are pairwise distinct variables not occurring in the premise.

Fig. 2. Converting language expressions to clause logic

Example 5. Let $\Sigma = \{a/0, s/1\}$. The constraints

$$\begin{array}{ll} X \supseteq [a,1,1] \circ \{()\} & X \supseteq [s(1),2,s(3)] \circ X \\ Y \supseteq [a,1,a] \circ \{()\} & Y \supseteq [s(4),1,3] \circ ((X \times Y)/[1,2,3,4,1,2]) \end{array}$$

define the languages

$$\begin{aligned} \mathcal{L}(X) &= \{\, (s^m(a), s^n(a), s^{m+n}(a)) \mid m,n \geq 0 \,\} \\ \mathcal{L}(Y) &= \{\, (s^m(a), s^n(a), s^{m*n}(a)) \mid m,n \geq 0 \,\} \end{aligned}$$

We obtain the following Horn clauses for the inductive definition:

$$\begin{array}{ll} P_X(a,x_1,x_1) \leftarrow & P_X(s(x_1),x_2,s(x_3)) \leftarrow P_X(x_1,x_2,x_3) \\ P_Y(a,x_1,a) \leftarrow & P_Y(s(x_4),x_1,x_3) \leftarrow P_X(x_1,x_2,x_3), P_Y(x_4,x_1,x_2) \end{array}$$

The transformation of the language expressions to Horn logic is given in Figure 3.

$$\dfrac{\dfrac{\dfrac{\dfrac{\{()\} \rightsquigarrow \langle (),\, \{\}\rangle}{(a,1,1) \circ \{()\} \rightsquigarrow \langle (a, x_1, x_1),\, \{\}\rangle} \qquad \dfrac{\{()\} \rightsquigarrow \langle (),\, \{\}\rangle}{(a,1,a) \circ \{()\} \rightsquigarrow \langle (a, x_1, a),\, \{\}\rangle}}{X \rightsquigarrow \langle (x_1,x_2,x_3),\, \{P_X(x_1,x_2,x_3)\}\rangle}}{(s(1),2,s(3)) \circ X \rightsquigarrow \langle (s(x_1),x_2,s(x_3)),\, \{P_X(x_1,x_2,x_3)\}\rangle} \qquad Y \rightsquigarrow \langle (x_4,x_5,x_6),\, \{P_Y(x_4,x_5,x_6)\}\rangle}{X \times Y \rightsquigarrow \langle (x_1,x_2,x_3,x_4,x_5,x_6),\, \{P_X(x_1,x_2,x_3), P_Y(x_4,x_5,x_6)\}\rangle}}{X \times Y/[1,2,3,4,1,2] \rightsquigarrow \langle (x_1,x_2,x_3,x_4,x_1,x_2),\, \{P_X(x_1,x_2,x_3), P_Y(x_4,x_1,x_2)\}\rangle}}{[s(4),1,3] \circ (X \times Y/[1,2,3,4,1,2]) \rightsquigarrow \langle (s(x_4),x_1,x_3),\, \{P_X(x_1,x_2,x_3), P_Y(x_4,x_1,x_2)\}\rangle}$$

Fig. 3. Converting the expressions of Example 5 to clause logic

For every predicate symbol P of arity n, let A_P denote an n-ary language variable uniquely associated with P, and let σ be a fixed substitution replacing every first-order variable by a unique positive integer, i.e., $\sigma = \{x_1 \mapsto 1, x_2 \mapsto 2, \ldots\}$. We define the constraint corresponding to a program clause as

$$\begin{array}{l}\text{constraint}(\, P(s_1, s_2, \ldots) \leftarrow P_1(t_{11}, t_{12}, \ldots), P_2(t_{21}, t_{22}, \ldots), \ldots\,) \stackrel{\text{def}}{=} \\ \quad A_P \supseteq [s_1\sigma, s_2\sigma, \ldots] \circ \big((A_{P_1} \times A_{P_2} \times \cdots) \,/\, [t_{11}\sigma, t_{12}\sigma, \ldots, t_{21}\sigma, t_{22}\sigma, \ldots]\big)\end{array}$$

and the inductive definition corresponding to a logic program, $\mathcal{P}$, as

$$\text{indef}(\mathcal{P}) \stackrel{\text{def}}{=} \bigcup_{C \in \mathcal{P}} \{\text{constraint}(C)\}\ .$$

The following proposition states that inductive definitions and logic programs are essentially the same and that the transformations above preserve equivalence with respect to the generated tuple languages.

Proposition 1. *Let $\mathcal{D}$ be any inductive definition and $\mathcal{P}$ any logic program.*
(a) $\mathcal{L}_{\mathcal{D}}(A) = \mathcal{M}(\text{horn}(\mathcal{D}))|_{P_A}$ for every language variable A.
(b) $\mathcal{M}(\mathcal{P})|_P = \mathcal{L}_{\text{indef}(\mathcal{P})}(A_P)$ for every predicate symbol P.

As corollary we obtain that the tuple languages definable by inductive definitions are exactly the recursively enumerable tuple languages.

A program clause $H \leftarrow B_1, \ldots, B_k$ is a *cs-clause* if $B_1, \ldots, B_k$ is linear and contains no function symbols, i.e., if all arguments of the B_i are variables occurring nowhere else in the body. A cs-clause is called

- *linear* iff the head is linear;
- *horizontal* iff any two head variables that are arguments of the same body atom occur at the same depth in the head;
- *pseudo-regular* iff each argument of the head is of the form $f(x_1, \ldots, x_{\text{ar}(f)})$ and there exists a mapping $\pi\colon V \mapsto \omega$ such that $\pi(x_l) = l$ for all $l = 1, \ldots, \text{ar}(f)$ and $\pi(x) = \pi(y)$ for all variables x and y occurring in the same body atom.
- *regular* iff it is pseudo-regular and linear.

A logic program is a *cs-program* iff all its clauses are cs-clauses. It is linear (horizontal, regular, pseudo-regular) iff all its clauses are.

The condition for pseudo-regularity essentially requires that the variables can be partitioned into disjoint groups, where variables occurring in the same body atom have to belong to the same group, such that there is a 1-1 correspondence between the argument positions of the function symbols and these groups.

The next proposition states that the subclasses of inductive definitions correspond to their counterparts in clause logic.

Proposition 2.
(a) If $\mathcal{D}$ is a (linear, horizontal, regular, or pseudo-regular) constraint system, then $\mathrm{horn}(\mathcal{D})$ *is a (linear, horizontal, regular, or pseudo-regular) cs-program.*
(b) If $\mathcal{P}$ is a (linear, horizontal, regular, or pseudo-regular) cs-program then $\mathrm{indef}(\mathcal{P})$ *is a (linear, horizontal, regular, or pseudo-regular) constraint system.*

5 Transforming logic programs to cs-programs

This section presents two rules, *unfolding* and *definition introduction*, that transform logic programs to equivalent cs-programs; they are particular instances of rules studied in the field of logic program transformation [21]. Typically the starting point of the transformation is a cs-program satisfying properties like linearity, and a single non-conforming clause that specifies e.g. the intersection of two sets. If the transformation process terminates the resulting cs-program represents the intersection by a cs-program of a particular kind.

The rules transform states $\langle \mathcal{P}, \mathcal{D}_{\mathrm{new}}, \mathcal{D}_{\mathrm{done}}, \mathcal{C}_{\mathrm{new}}, \mathcal{C}_{\mathrm{out}} \rangle$ where $\mathcal{P}$ is a logic program that remains unchanged, $\mathcal{D}_{\mathrm{new}}$ are definitions not yet unfolded, $\mathcal{D}_{\mathrm{done}}$ are definitions already processed but still used for simplifying clauses, $\mathcal{C}_{\mathrm{new}}$ are clauses generated from definitions by unfolding, and $\mathcal{C}_{\mathrm{out}}$ are the cs-clauses 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. A set of definitions, $\mathcal{D}$, is *compatible with* $\mathcal{P}$, if all predicate symbols occurring in the heads of the definitions of $\mathcal{D}$ occur only once in $\mathcal{D} \cup \mathcal{P}$; the only exception are tautological definitions of the form $P(\boldsymbol{x}) \leftarrow P(\boldsymbol{x})$ where P may occur without restrictions throughout $\mathcal{D}$ and $\mathcal{P}$. The predicate symbols in the heads of $\mathcal{D}$ are called the *predicate symbols defined by* $\mathcal{D}$. Tautological definitions are convenient to trigger the transformation of the clauses defining P without having to introduce a new predicate symbol; the alternative would be to replace the tautology by $P'(\boldsymbol{x}) \leftarrow P(\boldsymbol{x})$ for some new predicate symbol P'.

We write $S \Rightarrow S'$ if S' is a state obtained from state S by one application of rule *unfolding* or rule *definition introduction* (see below for their description). The reflexive and transitive closure of $\Rightarrow$ is denoted by $\stackrel{*}{\Rightarrow}$. 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 called *complete* if its last state is a final one.

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. Formally:

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

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.[6]

Example 6. Let $\mathcal{P}$ be the cs-program

$$\begin{array}{ll} M(x, a, x) \leftarrow & E(a) \leftarrow \\ M(s(x), s(y), z) \leftarrow M(x, y, z) & E(s(s(x))) \leftarrow E(x) \end{array}$$

and let $\mathcal{D} = \{R(x, y, z) \leftarrow M(x, y, z), E(y)\}$. Predicate M defines subtraction (minus), E evenness, and R defines all pairs of numbers with an even difference. Note that this definition is not a cs-clause since its body atoms share variables.

Figure 4 gives a complete derivation starting with input $\mathcal{P}$ and $\mathcal{D}$. We omit $\mathcal{D}_{\text{done}}$ since it consists just of the definitions in $\mathcal{D}_{\text{new}}$. Moreover, we list definitions and clauses in columns $\mathcal{D}_{\text{new}}$ and $\mathcal{C}_{\text{out}}$ only once in the step that adds them. Unfolding selects always the leftmost atom of maximal term depth. The third column, $\mathcal{C}_{\text{out}}$, lists the generated cs-clauses. The last column gives the applied rule: U_P means unfolding the leftmost atom of maximal depth with the clauses

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rule
$R(x,y,z) \leftarrow$ $\quad M(x,y,z), E(y)$			U_M
	$R(x,a,x) \leftarrow E(a)$ $R(s(x),s(y),z) \leftarrow$ $\quad M(x,y,z), E(s(y))$		D
$E' \leftarrow E(a)$	$R(s(x),s(y),z) \leftarrow$ $\quad M(x,y,z), E(s(y))$	$R(x,a,x) \leftarrow E'$	U_E
	$E' \leftarrow$ $R(s(x),s(y),z) \leftarrow$ $\quad M(x,y,z), E(s(y))$		D
	$R(s(x),s(y),z) \leftarrow$ $\quad M(x,y,z), E(s(y))$	$E' \leftarrow$	D
$R'(x,y,z) \leftarrow$ $\quad M(x,y,z), E(s(y))$		$R(s(x),s(y),z) \leftarrow$ $\quad R'(x,y,z)$	U_E
	$R'(x,s(y),z) \leftarrow$ $\quad M(x,s(y),z), E(y)$		D
$R''(x,y,z) \leftarrow$ $\quad M(x,s(y),z), E(y)$		$R'(x,s(y),z) \leftarrow$ $\quad R''(x,y,z)$	U_M
	$R''(s(x),y,z) \leftarrow$ $\quad M(x,y,z), E(y)$		D
		$R''(s(x),y,z) \leftarrow$ $\quad R(x,y,z)$	

Fig. 4. Transforming the program of Example 6 to a cs-program

for predicate P, and D means definition introduction applied to the first clause in $\mathcal{C}_{\text{new}}$.

Theorem 1 (Correctness). *Let $\mathcal{P}$ be a logic program and $\mathcal{D}$ be a set of definitions compatible with $\mathcal{P}$. If $\langle \mathcal{P}, \mathcal{D}, \emptyset, \emptyset, \emptyset \rangle \stackrel{*}{\Rightarrow} \langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$, then $\mathcal{P}'$ is a cs-program with the property that $\mathcal{M}(\mathcal{P}')|_P = \mathcal{M}(\mathcal{P} \cup \mathcal{D})|_P$ for all predicate symbols P defined by $\mathcal{D}$.*

Proof. $\mathcal{P}'$ is a cs-program by construction: subsets of body atoms sharing variables or containing function symbols are replaced by single linear atoms without function symbols. The semantic equivalence of the logic programs $\mathcal{P} \cup \mathcal{D}$ and $\mathcal{P}'$ with respect to the predicate symbols defined by $\mathcal{D}$ follows from the correctness of unfolding and definition introduction in general (see e.g. [21]). □

Remark 1. Definition introduction involves the test whether $\mathcal{D}_{\text{done}}$ contains definitions with bodies equal to components $\mathcal{B}_i$ up to variable renaming. This test is equivalent to the problem of checking two graphs for isomorphism, which is in NP but is conjectured to be neither NP-complete nor contained in P.

Remark 2. For any state derivable from an initial state, $\mathcal{D}_{\text{new}} \cup \mathcal{D}_{\text{done}}$ is compatible with $\mathcal{P} \cup \mathcal{C}_{\text{new}}$. This means in particular that newly introduced predicate symbols do not occur in the bodies of definitions and clauses in $\mathcal{D}_{\text{new}}$, $\mathcal{D}_{\text{done}}$, and $\mathcal{C}_{\text{new}}$.

Remark 3. In general cs-programs generated by $\Rightarrow$-derivations contain unproductive clauses, i.e., clauses that do not contribute to minimal models. Such clauses can be removed by the following procedure. Starting with the facts, mark each clause $P(\boldsymbol{s}) \leftarrow Q_1(\boldsymbol{t}_1), \ldots, Q_k(\boldsymbol{t}_k)$ as productive and its predicate symbol P as non-empty provided that all Q_i have been marked as non-empty before. Repeat the process until no more clauses and predicate symbols can be marked. Clauses not marked productive can be removed without affecting the semantics of the program. In fact this pruning of clauses is sufficient to test for emptiness of minimal models: the set of clauses is empty after removal of unproductive clauses if and only if the minimal Herbrand model is empty. As corollary, the emptiness problem of constraint systems is decidable.

Remark 4. To compute a cs-program equivalent to an arbitrary logic program, $\mathcal{P}$, let $\mathcal{D}_{\mathcal{P}}$ be the set of all tautologies $P(\boldsymbol{x}) \leftarrow P(\boldsymbol{x})$ such that P occurs in $\mathcal{P}$. The output of every complete derivation starting from $\mathcal{P}$ and $\mathcal{D}$ is a cs-program equivalent to $\mathcal{P}$ on all its predicates.

6 Termination and other properties

An algorithm that is able to compute cs-programs for arbitrary programs has to loop necessarily on certain inputs. If it would not we could use it to decide

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

emptiness of minimal Herbrand models for logic programs, known to be an undecidable problem: just compute an equivalent cs-program and test its minimal model for emptiness according to Remark 3 above. In fact, the rules presented in the last section do not terminate for quite simple inputs; take e.g. program $\{P(x) \leftarrow P(s(x))\}$ and definition $P(x) \leftarrow P(x)$. However, in this section we will show that the derivation process terminates for several interesting classes. The point is that the same rules can be used to cope with all of them, i.e., we have a uniform approach to solve various problems considered in the area of tree tuple grammars and constraint systems.

A derivation is bound to be finite if from some point onwards no further definitions are added to $\mathcal{D}_{\text{new}}$ because all required definitions are already there. To show this property for all programs of a class it is sufficient to prove that the variable-disjoint subsets $\mathcal{B}_i$ in rule *definition introduction* satisfy two conditions:

- the number of atoms in $\mathcal{B}_i$ is bounded;
- the maximal depth of atoms in $\mathcal{B}_i$ is bounded.

In this case there is only a finite number of potential $\mathcal{B}_i$'s and therefore also only a finite number of potential definitions up to variable renaming. Note that according to Remark 2 the $\mathcal{B}_i$'s are built over the original signature, i.e., the number of occurring predicate symbols does not grow.

6.1 Quasi-cs programs

The following class is a generalization of cs-programs that allows function symbols also in the body of clauses.

Definition 1. *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.*

Every quasi-cs program can be transformed to an equivalent finite cs-program.

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

The proof uses some properties of linear terms with respect to unification. In the following, τ_t denotes the depth of term t, and $\tau_{x,t}$ denotes the minimal depth of any occurrence of variable x in term t.

Lemma 1. *Let s and t be variable-disjoint linear terms that are unifiable with most general unifier μ. Then* $\text{ran}(\mu)$*, the set of terms introduced by μ, is a linear set of terms,*[7] *and $\tau_{s\mu} = \max(\tau_s, \tau_t)$.*

[7] Note that "a linear set of terms" is more restricted than "a set of linear terms" since in the latter case the terms need not be variable-disjoint.

Proof. Starting from the unification problem $\{s \stackrel{?}{=} t\}$ we compute μ by a rule-based approach as described e.g. in [1]. Since s and t are linear and variable-disjoint, every variable occurs at most once in the unification problem, and the decomposition rule and the deletion rule for trivial equations suffice to compute the unifier. All left- and right-hand sides of the final system are non-overlapping subterms of s and t, and therefore are linear and mutually variable-disjoint. The set $\mathrm{ran}(\mu)$ is a subset of these left- and right-hand sides and thus is linear. Moreover, the set of positions in the unified term $s\mu$ $(= t\mu)$ is the union of the positions in s and t, implying $\tau_{s\mu} = \max(\tau_s, \tau_t)$. □

Iterating Lemma 1 we obtain a similar result for sets of linear terms.

Corollary 1. *Let $\{t_1, \ldots, t_n\}$ be a set of mutually variable-disjoint linear terms that are simultaneously unifiable with most general unifier μ. Then $\mathrm{ran}(\mu)$, the set of terms introduced by μ, is a linear set of terms, and $\tau_{t_1\mu} = \max\{\tau_{t_1}, \ldots, \tau_{t_n}\}$.*

Even if just one of two terms is linear, the most general unifier still has some particular properties.

Lemma 2. *Let s and t be variable-disjoint terms unifiable with most general unifier μ, and let t be linear. Let μ_s be the substitution μ restricted to the variables in s, i.e., $\mu_s = \mu|_{\mathrm{var}(s)}$. Then $\mathrm{ran}(\mu_s)$, the set of terms introduced by μ for variables in s, is a linear set of terms whose variables are among those of t and $\tau_{x\mu_s} \leq \tau_t - \tau_{x,s}$ holds for all variables x in term s.*

Proof. Starting from the unification problem $\{s \stackrel{?}{=} t\}$ we compute μ by a rule-based approach as described e.g. in [1]. The rule applications can be arranged in a way such that unification proceeds in three stages:

1. Apply exhaustively the decomposition rule and remove trivial equations. The resulting problem, P, contains two types of equations: $x \stackrel{?}{=} t'$ where x is a variable of s and t' is a subterm of t, and $s' \stackrel{?}{=} y$ where y is a variable of t and s' is a subterm of s. For a variable x there may be several equations of type $x \stackrel{?}{=} t'$, but for any variable y there is at most one equation of type $s' \stackrel{?}{=} y$ since t is linear. Note that for any two different equations $x_1 \stackrel{?}{=} t_1$ and $x_2 \stackrel{?}{=} t_2$ the terms t_1 and t_2 are variable-disjoint and linear.
2. Let P_x be the set of all equations in P with variable x as left-hand side, say $P_x = \{x \stackrel{?}{=} t_1, \ldots, x \stackrel{?}{=} t_n\}$. The system can be solved by computing the most general unifier μ_x of $\{t_1, \ldots, t_n\}$. Let t_x denote the unified term $t_1\mu_x = \cdots = t_n\mu_x$. Then P_x is equivalent to the solved problem $P'_x = \{x \stackrel{?}{=} t_x\} \cup \mu_x$ where μ_x is regarded as a set of equations.
   All t_i's are linear and mutually variable-disjoint. By Corollary 1, the set $\mathrm{ran}(\mu_x)$ is linear and $\tau_{t_1\mu_x} = \max\{\tau_{t_1}, \ldots, \tau_{t_n}\}$. The t_i's contain only variables from t, hence the variables in $\mathrm{ran}(\mu_x)$ are also among those of t. Moreover, the depth of the t_i's is bounded by $\tau_t - \tau_{x,s}$, since they are subterms of t occurring at the same positions in t as x occurs in s. Consequently $t_x = t_1\mu_x$ is linear and $\tau_{t_x} = \tau_{t_1\mu_x} \leq \tau_t - \tau_{x,s}$. Note that for $x_1 \neq x_2$ the unified terms t_{x_1} and t_{x_2} (corresponding to P_{x_1} and P_{x_2}, respectively) are variable-disjoint.

3. Let P' be the system obtained from P by replacing the sub-problems P_x by P'_x for all variables x in s. In general P' is not yet solved, as the x's may occur within the left-hand side of equations of type $s' \stackrel{?}{=} y$. After eliminating these occurrences by replacing x by t_x we obtain a problem in solved form. Since these variable eliminations leave the sub-problems P'_x unchanged, we have $\mu_s = \{ x \mapsto t_x \mid x \in \mathrm{var}(s) \}$. The properties of μ_s to be proved now follow directly from the discussion in the second step. □

Proof. Proof of Theorem 2 Let τ be the maximal depth of any atom occurring in $\mathcal{P}$. We show that

- the bodies of clauses added by unfolding to $\mathcal{C}_{\mathrm{new}}$ are linear with the depth bounded by τ, and that
- the bodies of definitions added to $\mathcal{D}_{\mathrm{new}}$ are single linear atoms with the depth bounded by τ.

This implies that only finitely many definitions are generated up to variable renaming, i.e., complete derivations are finite. Note that the initial definitions in $\mathcal{D}_{\mathrm{new}}$, i.e. those of $\mathcal{D}_{\mathcal{P}}$, are of this particular form.

Unfolding.
Consider a definition $L \leftarrow A$ in $\mathcal{D}_{\mathrm{new}}$ and a quasi-cs-clause $H \leftarrow \mathcal{B}$ in $\mathcal{P}$ such that A and H are unifiable with most general unifier μ. The clause added to $\mathcal{C}_{\mathrm{new}}$ is $(L \leftarrow \mathcal{B})\mu = L\mu \leftarrow \mathcal{B}\mu = L\mu_A \leftarrow \mathcal{B}\mu_H$, where $\mu_A = \mu|_{\mathrm{var}(A)}$ and $\mu_H = \mu|_{\mathrm{var}(H)}$. The atoms H and A are variable-disjoint (definitions and clauses are renamed apart prior to unfolding) and A is linear, therefore, by Lemma 2, $\mathrm{ran}(\mu_H)$ is a linear set of terms, whose variables are among those of A and thus do not occur in $\mathcal{B}$. Since $\mathcal{B}$ is linear by definition, $\mathcal{B}\mu_H$ is linear, too.

It remains to show that the depth of $\mathcal{B}\mu_H$ is bounded by τ. Let x be a variable occurring both in head H and body $\mathcal{B}$ of the clause. By the definition of quasi-cs-ness we have $\tau_{x,\mathcal{B}} \leq \tau_{x,H}$ and thus $\tau_{x,\mathcal{B}} + \tau_{x\mu_H} \leq \tau_{x,H} + \tau_{x\mu_H}$, Lemma 2 yields $\tau_{x,H} + \tau_{x\mu_H} \leq \tau_A$, and $\tau_A \leq \tau$ holds by induction hypothesis. Chaining the inequalities together we obtain $\tau_{x,\mathcal{B}} + \tau_{x\mu_H} \leq \tau_A$. Therefore the depth of $\mathcal{B}\mu_H$, being the maximum of the depths of atoms in $\mathcal{B}$ and of the values $\tau_{x,\mathcal{B}} + \tau_{x\mu}$ for all variables x shared between head and body, is also bounded by τ.

Definition introduction.
The body of every clause $H \leftarrow \mathcal{B}$ in $\mathcal{C}_{\mathrm{new}}$ is linear, hence its maximal decomposition into variable-disjoint subsets of atoms consists of singletons only. Since the depth of $\mathcal{B}$ is bounded by τ, the new clauses added to $\mathcal{D}_{\mathrm{new}}$ are of the form $L \leftarrow A$ where A is linear and of depth at most τ. □

Definition 2. *For an n-ary language, L, and an n-ary template, $\square$, the emptiness test for filtering is the problem to determine whether $L/\square$ is the empty language, or alternatively, whether L contains any tuple that is an instance of $\square$.*

Note that the membership test "$\boldsymbol{t} \in L$?" for a term tuple $\boldsymbol{t}$ is an instance of the emptiness problem for filters, since it is equivalent to "$L/\boldsymbol{t} \neq \emptyset$?".

Corollary 2. *The emptiness test for linear filter operations (and therefore the membership test) for languages defined by constraint systems is decidable.*

Proof. Let L be a language such that $L = \mathcal{L}_{\mathcal{D}}(A)$ for some constraint system $\mathcal{D}$ and some variable A, and let $\Box$ be a linear template tuple. Consider the constraint system $\mathcal{D}'$ obtained by adding the constraint $A' \supseteq [] \circ (A/\Box)$ to $\mathcal{D}$, for a new nullary variable A'. We have $\mathcal{L}_{\mathcal{D}'}(A') = \emptyset$ if $L/\Box$ is empty, and $\mathcal{L}_{\mathcal{D}'}(A') = \{()\}$ otherwise.

The corresponding logic program is quasi-cs, since it consists of the cs-clauses in horn($\mathcal{D}$) and the single quasi-cs clause

$$P_{A'} \leftarrow P_A(\Box\{1 \mapsto x_1, 2 \mapsto x_2, \ldots\}) \ .$$

(Note that the body is linear since $\Box$ is, and that head and body share no variables since $P_{A'}$ has no arguments.) According to Theorem 2, all complete $\Rightarrow$-derivations starting from this quasi-cs program are finite. After eliminating unproductive clauses according to Remark 3, the resulting cs-program contains either no clause for $P_{A'}$, indicating $\mathcal{L}_{\mathcal{D}'}(A') = \emptyset$, or clauses equivalent to the singleton $P_{A'} \leftarrow$, indicating $\mathcal{L}_{\mathcal{D}'}(A') = \{()\}$. $\Box$

Corollary 2 extends the result given in [14] since the latter considered the membership test for linear constraint systems only.

Example 7. Consider the cs-program constructed in Example 6. To test whether the tuple $(s(a), s(s(a)), a)$ belongs to the language associated with predicate R we add the clause $P \leftarrow R(s(a), s(s(a)), a)$ to the program defining R:

$$\begin{array}{ll} R(x, a, x) \leftarrow & R'(x, s(y), z) \leftarrow R''(x, y, z) \\ R(s(x), s(y), z) \leftarrow R'(x, y, z) & R''(s(x), y, z) \leftarrow R(x, y, z) \end{array}$$

Figure 5 shows the relevant steps of the $\Rightarrow$-derivation; each line corresponds to the application of one unfolding and one definition introduction step. The

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rules
$P \leftarrow P$			U_P, D
$P' \leftarrow R(s(a), s(s(a)), a)$	$P \leftarrow R(s(a), s(s(a)), a)$	$P \leftarrow P'$	U_R, D
$P'' \leftarrow R'(a, s(a), a)$	$P' \leftarrow R'(a, s(a), a)$	$P' \leftarrow P''$	

Fig. 5. Membership test of Example 7

last definition in $\mathcal{D}_{\text{new}}$ cannot be unfolded because its body unifies with no head in the program, so we obtain the program $\{P \leftarrow P', P' \leftarrow P''\}$ which is equivalent to the empty one. Hence $(s(a), s(s(a)), a)$ does not belong to the language associated with R.

Now suppose we want to know whether there is any instance of the template $(s(1), s(2), a)$ that belongs to the language associated with R. We add the clause $P \leftarrow R(s(x), s(y), a)$ and compute again a $\Rightarrow$-derivation (see Figure 6). Using the algorithm in Remark 3 we find that all clauses are productive and that P is non-empty. We conclude that at least one ground instance of $(s(x), s(y), a)$ belongs to the language of R.

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rules
$P \leftarrow P$			U_P, D
$P' \leftarrow R(s(x), s(y), a)$	$P \leftarrow R(s(x), s(y), a)$	$P \leftarrow P'$	U_R, D
$P'' \leftarrow R'(x, y, a)$	$P' \leftarrow R'(x, y, a)$	$P' \leftarrow P''$	$U_{R'}, D$
$P''' \leftarrow R''(x, y, a)$	$P'' \leftarrow R''(x, y, a)$	$P'' \leftarrow P'''$	$U_{R''}, D$
$P'''' \leftarrow R(x, y, a)$	$P''' \leftarrow R(x, y, a)$	$P''' \leftarrow P''''$	U_R, D
	$P'''' \leftarrow$ $P'''' \leftarrow R'(x, y, a)$	$P'''' \leftarrow$	U_R, D
	$P'''' \leftarrow R'(x, y, a)$	$P'''' \leftarrow P''$	D

Fig. 6. Emptiness test of Example 7

6.2 Intersection of pseudo-regular relations

In this section we discuss the class of pseudo-regular cs-programs which correspond to pseudo-regular constraint systems. According to its definition, pseudo-regularity extends regularity by admitting a restricted form of duplicated variables. We show that this class is closed under intersections and joins like regular relations; additionally it is closed under a generalized join operation performing a self-join, i.e., selecting tuples where two or more components are identical.

This is an interesting result since it is the first class of tree tuple recognizers admitting duplication that has a decidable emptiness test and is closed under intersection at the same time.[8] [25] introduces several super-classes of regular relations that are defined via tree automata with equality constraints. They are all closed under intersection (which usually is not the case with constraint systems) but their emptiness test is undecidable: adding just one equality constraint is sufficient for the membership test to become undecidable.

A definition $H \leftarrow B_1, \ldots, B_k$ is called a *join-definition* if it contains no function symbols. Join-definitions are no cs-clauses in general since a variable may occur several times throughout the body, within a single atom as well as in several atoms. This property allows us to represent intersections and joins of two or more tuple languages as a join-definition.

Theorem 3. *Let $\mathcal{P}$ be a pseudo-regular cs-program, and let D be a join-definition compatible with $\mathcal{P}$. Then any complete derivation with input $\mathcal{P}$ and $\{D\}$ that unfolds in each unfolding step all atoms simultaneously is finite and its output is a pseudo-regular cs-program.*

The proof relies on the fact that the complexity of join definitions does not increase during a derivation and that the number of definitions of a given complexity is finite. To make this observation precise we introduce the following notions. For an object O let $\operatorname{nocc}(x, O)$ denote the number of occurrences of x in O and $\operatorname{nocc}(O) = \sum_{x \in \operatorname{var}(O)} \operatorname{nocc}(x, O)$ the total number of variable occurrences. Moreover, let $\operatorname{nvar}(O) = |\operatorname{var}(O)|$ be the number of different variables

[8] Some classes of tree automata with equality constraints such as those presented in [2] have this property as well but they describe only unary languages.

in O. The complexity of a variable in an object is the number of its excess occurrences: $\chi(x, O) = \mathrm{nocc}(x, O) - 1$. The complexity of an object is the sum of all excess occurrences:

$$\chi(O) = \sum_{x \in \mathrm{var}(O)} \chi(x, O) = \mathrm{nocc}(O) - \mathrm{nvar}(O)$$

The complexity of a clause or definition is the complexity of its body.

The complexity of a singleton variable is zero and does not add to the complexity of an object. The complexity of a linear set of atoms is zero, too. Complexity increases when either the number of occurrences of a particular variable increases or when the number of variables with multiple occurrences increases. We note the following properties of χ.

Lemma 3. *Let $\mathcal{A}$ and $\mathcal{B}$ be sets of atoms sharing no variables. Then $\chi(\mathcal{A} \cup \mathcal{B}) = \chi(\mathcal{A}) + \chi(\mathcal{B})$.*

Lemma 4. *If $\mathcal{A}$ is a set of atoms that is not decomposable into variable-disjoint subsets then it contains at most $\chi(\mathcal{A}) + 1$ atoms.*

This bound is tight; take e.g. $\mathcal{A} = \{P_1(x), \ldots, P_k(x)\}$, which is of complexity $k - 1$.

Proof. Proof of Theorem 3 We show that

- unfolding a join-definition in $\mathcal{D}_{\mathrm{new}}$ adds to $\mathcal{C}_{\mathrm{new}}$ clauses that are pseudo-regular except that the body consists of variable-disjoint sets of atoms, $\mathcal{S}_j$, (instead of variable-disjoint linear atoms) which satisfy three conditions: (a) $\mathcal{S}_j$ contains no function symbols, (b) the complexity of $\mathcal{S}_j$ does not exceed the complexity of the definition, and (c) all variables occurring in $\mathcal{S}_j$ are associated with the same functional argument position of the head, i.e., $\pi(x) = \pi(y)$ for all $x, y \in \mathrm{var}(\mathcal{S}_j)$.
- applying definition introduction to a clause satisfying the three conditions above produces pseudo-regular clauses in $\mathcal{C}_{\mathrm{out}}$ and join-definitions in $\mathcal{D}_{\mathrm{new}}$ with a complexity not exceeding the complexity of the initial join-definition $\mathcal{D}$.

The last property guarantees termination: the complexity of new definitions is not greater than the one of the initial definition, $\chi(\mathcal{D})$. This limits the number of atoms in a definition body to $\chi(\mathcal{D}) + 1$ (Lemma 4). Therefore only finitely many definitions are generated up to variable renaming, i.e., complete derivations are finite.

Unfolding.
Let $D = L \leftarrow A_1, \ldots, A_n$ be a join definition, and let $C_i = H_i \leftarrow \mathcal{B}_i$, for $1 \leq i \leq n$, be clauses such that the most general simultaneous unifier of $(A_1, \ldots, A_n)$ and $(H_1, \ldots, H_n)$ exists. The unifer is the solution of the problem $\Pi = \{A_i \stackrel{?}{=} H_i\}_i$, or equivalently of $\{y_{ij} \stackrel{?}{=} f_{ij}(\boldsymbol{z}_{ij})\}_{ij}$ if $A_i = P_i(\cdots y_{ij} \cdots)$ and $H_i = P_i(\cdots f_{ij}(\boldsymbol{z}_{ij}) \cdots)$.[9] (Regarding the details of unification we refer the reader to [1].)

[9] We use $\{O_i\}_i$ as shorthand for the set of all objects O_i where i ranges over all defined values (from 1 to n in this case). Similarly, $\{O_{ij}\}_{ij}$ is the set of all O_{ij} were both i and j take all possible values.

Let $x_1, x_2, \ldots$ be an enumeration of the variables occurring in D; obviously we have $\{x_k\}_k = \{y_{ij}\}_{ij}$. We rewrite the unification problem as $\bigcup_k \Pi_k$ where $\Pi_k = \{\, x_k \stackrel{?}{=} f_k(\boldsymbol{z}_{ij}) \mid y_{ij} = x_k \,\}_{ij}$.[10] Π_k contains as many equations as x_k has occurrences in the body of D. If there is more than one, then Π_k is not yet solved and we have to unify the terms $\{\, f_k(\boldsymbol{z}_{ij}) \mid y_{ij} = x_k \,\}_{ij}$.

To avoid the proliferation of indices we assume in the following discussion that f_k is unary, and write z_{ij} instead of $\boldsymbol{z}_{ij}$. Accordingly, the bodies $\mathcal{B}_i$ of the clauses contain only variables associated with this single functional position, i.e., we have $\pi_i(x) = 1$ for all variables x. The simplification is no loss of generality: By definition of pseudo-regularity (and because of renaming the clauses apart prior to unification) the only multiple occurrences of variables in the clause heads are at identical functional positions and at the same depth. Hence only variables with the same π-value are unified, whereas variables with different π-values do not interfere.

Let ρ_k be the most general unifier of the terms $\{\, f_k(z_{ij}) \mid y_{ij} = x_k \,\}_{ij}$, i.e., of the variables $\{\, z_{ij} \mid y_{ij} = x_k \,\}_{ij}$. It maps all variables to one particular variable in the set, say z_k. Viewing ρ_k as set of equations, the unification problem Π_k is equivalent to the solved problem $\Pi'_k = \{x_k \stackrel{?}{=} f_k(z_k)\} \cup \rho_k$. Observe that the domain of the substitution ρ_k consists of the variables in the set $\{\, z_{ij} \mid y_{ij} = x_k \,\}_{ij}$ minus the chosen representative z_k. The set contains as many variables as there are occurrences of x_k in the definition, or less if some of the z_{ij} happen to be identical. Therefore $|\operatorname{dom}(\rho_k)| \leq \chi(x_k, \{A_1, \ldots, A_n\})$.

If Π_k and Π_l are sub-problems sharing no variables, as e.g. is always the case for regular clauses, the solved problem corresponding to $\Pi_k \cup \Pi_l$ is simply the union of Π'_k and Π'_l:

$$\Pi'_k \cup \Pi'_l = \{x_k \stackrel{?}{=} f_k(z_k),\, x_l \stackrel{?}{=} f_l(z_l)\} \cup \rho_{k,l}$$

where $\rho_{k,l} = \rho_k \cup \rho_l$. Observe that $|\operatorname{dom}(\rho_{k,l})| = |\operatorname{dom}(\rho_k)| + |\operatorname{dom}(\rho_l)|$.

In general, however, Π_k and Pi_l share one or more variables, i.e. $z_{ij} = z_{ij'}$ for $j \neq j'$. The solved problem corresponding to $\Pi_k \cup \Pi_l$ can be obtained from Π'_k and Π'_l by replacing consistently z_l by z_k. We obtain

$$\Pi'_k \cup \Pi'_l\{z_l \leftarrow z_k\} = \{x_k \stackrel{?}{=} f_k(z_k),\, x_l \stackrel{?}{=} f_l(z_k)\} \cup \rho_{k,l}$$

where $\rho_{k,l} = \rho_k \cup \rho_l\{z_l \leftarrow z_k\}$. Observe that $|\operatorname{dom}(\rho_{k,l})| \leq |\operatorname{dom}(\rho_k)| + |\operatorname{dom}(\rho_l)|$; the less-than relation holds if the problems share more than one variable.

Let Π' be obtained by combining all solved sub-problems Π'_k iteratively in the manner described above: simple union if the sub-problems share no variables, or union with variable-merging if they do. Π' is solved and equivalent to the initial problem Π. Let μ be the substitution corresponding to Π', and let ϕ be the part of μ affecting the variables in the definition and ρ be the part affecting the clauses, i.e., $\mu = \phi \dot{\cup} \rho$. The substitution ϕ replaces variables by functional terms of the form $f(z)$, whereas ρ unifies some of the variables occurring in the clause heads. The number of variables replaced by ρ is less than or equal to the

[10] Note that $y_{ij} = y_{i'j'}$ implies $f_{ij} = f_{i'j'}$ since the unifier exists by assumption.

complexity of the definition, since according to the observations above we have

$$|\operatorname{dom}(\rho)| \le \sum_k |\operatorname{dom}(\rho_k)| \le \sum_k \chi(x_k, \{A_1, \ldots, A_n\}) = \chi(D) \ .$$

In the general case, with function symbols of arbitrary arity, ϕ maps variables to terms of the form $f(\boldsymbol{z})$ and ρ consists of disjoint substitutions σ_j, one for each argument position, satisfying $|\operatorname{dom}(\sigma_j)| \le \chi(D)$.

Unfolding adds the clause $(L \leftarrow \mathcal{A})\mu = L\phi \leftarrow \mathcal{A}\rho$ to $\mathcal{C}_{\text{new}}$, where $\mathcal{A} = \bigcup_j \mathcal{A}_j$ and $\mathcal{A}_j = \{\, B \in (\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_n) \mid \pi(B) = j \,\}$. The head $L\phi = P(\cdots f_k(\boldsymbol{z}_k) \cdots)$ is of the form required for pseudo-regular clauses. The sets $\mathcal{A}_j$ are mutually variable-disjoint and contain no function symbols, therefore the sets $\mathcal{A}_j\rho = \mathcal{A}_j\sigma_j =: \mathcal{S}_j$ have the same properties. Though $\mathcal{A}_j$ is linear, $\mathcal{S}_j$ in general is not, since σ_j replaces singleton variables by variables occurring already somewhere else in $\mathcal{S}_j$. Each replacement increases the complexity by one. The number of replaced variables is bounded by $|\operatorname{dom}(\sigma_j)|$, i.e., $\chi(\mathcal{S}_j) \le |\operatorname{dom}(\sigma_j)| \le \chi(D)$.

Definition introduction.
As discussed above, the clauses in $\mathcal{C}_{\text{new}}$ are of the form $P(\cdots f_k(\boldsymbol{z}_k) \cdots) \leftarrow \cdots \cup \mathcal{S}_j \cup \cdots$, where the S_j contain no function symbols and $\pi(x) = j$ for all variables x in $\mathcal{S}_j$. Moreover, the complexity of $\mathcal{S}_j$ is bounded by the complexity of the unfolded definition D, i.e., $\chi(\mathcal{S}_j) \le \chi(D)$.

Definition introduction decomposes each $\mathcal{S}_j$ into variable-disjoint subsets and replaces them by linear atoms. The resulting clause, added to $\mathcal{C}_{\text{out}}$, is obviously pseudo-regular since the body contains no duplicate variables anymore. The bodies of the definitions, added to $\mathcal{D}_{\text{new}}$, consist of the variable-disjoint subsets of $\mathcal{S}_j$. By Lemma 3, the complexity of $\mathcal{S}_j$ is the sum of the complexities of the subsets, i.e., the complexity of each subset and of each new definition is smaller than or equal to $\chi(D)$. Therefore, by an inductive argument, we conclude that the complexity of the definitions is bounded by $\chi(\mathcal{D})$, the complexity of the initial definition. □

Corollary 3. *Tuple languages specified by pseudo-regular constraints are closed under intersection and joins. A pseudo-regular constraint system representing the result of the operation can be computed via $\Rightarrow$-derivations.*

Proof. Language expressions like $X = A \cap B$ and $X = A \bowtie_{i,j} B$ translate to join-definitions. If A and B are defined by pseudo-regular constraint systems, then their logic counterparts are pseudo-regular cs-programs. Let $\mathcal{P}$ be the union of these two cs-programs and D be the join-definition corresponding to $X = A \cap B$ or $X = A \bowtie_{i,j} B$. According to Theorem 3 the derivation process terminates and yields a pseudo-regular cs-program representing X, which can be translated back to a pseudo-regular constraint system. □

Remark 5. Call a join-definition *simple*, if the body consists of linear atoms, i.e., variables may be shared between atoms but occur at most once in each atom. For a regular cs-program and a simple join-definition, Theorem 3 only guarantees that the result of the derivation is pseudo-regular. However, looking at the proof we see that the derivation preserves regularity: the linearity of all

participating atoms propagates to the clauses generated in $\mathcal{C}_{\text{out}}$. Hence the result of the derivation is again a regular cs-program. Translated to tree tuple languages this observation re-proves the well-known fact that tuple languages specified by regular constraints are closed under intersection.

Example 8. Consider the pseudo-regular constraints

$$\begin{array}{ll} S \supseteq \{0\} & S \supseteq [s(1)] \circ S \\ P \supseteq \{(0,0)\} & P \supseteq [c(1,2), c(1,3)] \circ (S \times P) \\ P' \supseteq \{(0,0)\} & P' \supseteq [c(2,1), c(3,1)] \circ (S \times P') \end{array}$$

$\mathcal{L}(P)$ is the set of all pairs (t, t) such that

$$t = c(s^{n_1}(0), c(s^{n_2}(0), \cdots c(s^{n_k}(0), 0) \cdots))$$

whereas $\mathcal{L}(P')$ is the set of all pairs (t', t') such that

$$t' = c(c(\cdots c(0, s^{n_k}(0)) \cdots, s^{n_2}(0)), s^{n_1}(0))$$

for $k, n_1, \ldots, n_k \geq 0$.

Suppose we want to compute the intersection of the two languages, i.e., $Q = P \cap P'$. First we transform the problem into a cs-program:

$$\begin{array}{rc} S(0) \leftarrow & S(s(x)) \leftarrow S(x) \\ P(0,0) \leftarrow & P(c(x,y), c(x,z)) \leftarrow S(x), P(y,z) \\ P'(0,0) \leftarrow & P'(c(y,x), c(z,x)) \leftarrow S(x), P(y,z) \end{array}$$

The constraint for the intersection corresponds to the join-definition

$$Q(x,y) \leftarrow P(x,y), P'(x,y)$$

Figure 7 shows a complete derivation starting from this non-cs-clause. As result we obtain the pseudo-regular cs-program

$$\begin{array}{rr} Q(0,0) \leftarrow & Q'(0) \leftarrow \\ Q(c(x,y), c(x,y)) \leftarrow Q'(x), Q''(y) & Q''(0) \leftarrow \end{array}$$

defining the language $\mathcal{L}(Q) = \{(0,0),\ (c(0,0), c(0,0))\}$.

The results on pseudo-regular constraint systems generalize those for regular relations [4]. Our approach allows one to work in a single uniform framework, namely logic programs and $\Rightarrow$-derivations. In particular, we can use the same algorithm for proving and computing membership, emptiness and all sorts of joins and (partial) intersections.

7 Related Work

The tight connection between set constraints and tree automata on the one hand and logic programs on the other is quite natural and is indeed used frequently in

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	rule
$Q(x,y) \leftarrow$ $P(x,y), P'(x,y)$			U
	$Q(0,0) \leftarrow$ $Q(c(x,y),c(x,y)) \leftarrow$ $S(x), P(y,y),$ $S(y), P'(x,x)$		D
	$Q(c(x,y),c(x,y)) \leftarrow$ $S(x), P(y,y),$ $S(y), P'(x,x)$	$Q(0,0) \leftarrow$	D
$Q'(x) \leftarrow$ $S(x), P'(x,x)$ $Q''(y) \leftarrow$ $S(y), P(y,y)$		$Q(c(x,y),c(x,y)) \leftarrow$ $Q'(x), Q''(y)$	U
$Q''(y) \leftarrow$ $S(y), P(y,y)$	$Q'(0) \leftarrow$		D
$Q''(y) \leftarrow$ $S(y), P(y,y)$		$Q'(0) \leftarrow$	U
	$Q''(0) \leftarrow$		D
		$Q''(0) \leftarrow$	

Fig. 7. Computing the intersection in Example 8

one way or another (see e.g. [10, 24] or [4, Section 7.6]). However, none of these papers exploits the logic programming point of view to answer questions about tree tuple languages.

More interesting parallels to our work can be found in the area of type inference and type checking for logic programs. The type of a variable in any programming language can be defined as the set of values the variable may take. In logic programming this leads to types given by sets of ground first-order terms. Frühwirth et al. [9] show that the types for a logic program can be approximated and described systematically as unary-predicate programs consisting of clauses $H(t) \leftarrow P_1(t_1), \ldots, P_n(t_n)$ where t is linear, and each t_i is either a subterm of t, a strong superterm of t, or variable-disjoint from t. Unfolding techniques simplify these type programs to regular unary-predicate programs which admit type checking (i.e., membership tests). The approach partially extends to higher-order terms [12] and AC tree automata [13].

A main difference between type checking in logic programming and our work is the arity of the languages and predicates considered. While we are interested in tree tuple languages of arbitrary arity and in the interaction of tuple components, types are unary languages. As a consequence our cs-programs may contain predicates of any arity, but require linear clause bodies. Unary-predicate symbols on the other hand are compatible with shared variables and nested terms.

Nielson et al. [20] extend [9] by admitting non-unary predicates to handle relations (i.e. tree tuples). They define so-called strongly recognizable relations by means of logic programs that consist of normalized clauses of the following

form:

$$\begin{array}{ll} P(f(x_1,\ldots,x_n)) & \leftarrow Q_1(x_1),\ldots,Q_k(x_n) \\ P(x_1,\ldots,x_n) & \leftarrow Q_1(x_1),\ldots,Q_k(x_n) \end{array}$$

where the x_i's are distinct variables. Since normalized clauses are quasi-cs, our transformation rules generate a cs-program in a finite number of steps. In fact, it is even regular since the head and the body of each clause are linear. [20] defines several classes of logic programs with equivalent normalized ones.

The basic idea of the transformation algorithms in [20] is to remove unnecessary "synchronized" variables. For example, the clause $P(f(x,y),z) \leftarrow P(x,y), Q(z)$ is equivalent to

$$\begin{array}{r} P(z',z) \leftarrow P'(z'), Q(z) \\ P'(f(x,y)) \leftarrow P'(x), Q(y) \end{array}$$

Our transformation rules are not able to detect such cases and therefore do not work for the classes defined in [20]. It seems worthwhile to try to integrate techniques from that paper into our framework, in order to enlarge the classes of logic programs that can be transformed to cs-programs or one of their subclasses.

Several algorithms have been proposed to decide the satisfiability of set constraints (i.e., to decide the emptiness test), like [3, 15]. These algorithms exploit algebraic properties of set expressions to reduce constraints to a kind of solved form, which in the case of the cited papers is a linear monadic constraint system (i.e. 1-tuples without duplicated variables). The transformation rules in those papers are not directly comparable to our technique since they use specific properties of the set constraints considered. We still have to investigate how our framework handles these classes.

Another connection between set constraint techniques and our work concerns the membership test. It can be approximated by type checking techniques using a set based analysis of logic programs. This approach yields no exact membership test for cs-programs since it approximates the semantics of the program by ignoring the relationship between variables, whereas the aim of synchronized languages is to maintain it. For example, the technique of [9] generates the following program for predicate R of Example 7:

$$\begin{array}{lll} Ty(f_R(x_1,a,x_2)) & \leftarrow All(x_1), All(x_2) & P_x(x) \leftarrow Ty(f_{R'}(x,y,z)) \\ Ty(f_R(s(x),s(y),z)) & \leftarrow P_x(x), P_y(y), P_z(z) & P_y(y) \leftarrow Ty(f_{R'}(x,y,z)) \\ Ty(f'_R(x,s(y),z)) & \leftarrow P'_x(x), P'_y(y), P'_z(z) & P_z(z) \leftarrow Ty(f_{R'}(x,y,z)) \\ Ty(f''_R(s(x),y,z)) & \leftarrow P''_x(x), P''_y(y), P''_z(z) & P'_x(x) \leftarrow Ty(f_{R''}(x,y,z)) \\ & & P'_y(y) \leftarrow Ty(f_{R''}(x,y,z)) \\ & & P'_z(z) \leftarrow Ty(f_{R''}(x,y,z)) \\ & & P''_x(x) \leftarrow Ty(f_R(x,y,z)) \\ & & P''_y(y) \leftarrow Ty(f_R(x,y,z)) \\ & & P''_z(z) \leftarrow Ty(f_R(x,y,z)) \end{array}$$

where All succeeds for all ground terms. Indeed it can be checked that (sa, ssa, a) does not belong to the model of R since the goal $Ty(f_R(sa, ssa, a))$ fails. The

goal $Ty(f_R(sssa, ssa, a))$, however, succeeds even though the tuple $(sssa, ssa, a)$ is not in the model of R.

The clause classes and proof techniques in this paper resemble to a certain degree those used for resolution decision procedures (see e.g. [6, 7]). There, clause classes inspired by classical quantifier-prefix classes are shown to be decidable by proving termination of certain resolution refinements. The termination proofs essentially bound the length and depth of clauses, like we do. A main difference, however, is that resolution decision procedures augment the initial set of clauses with the aim to derive the empty clause, whereas we replace clauses by simpler ones until the whole program belongs to a simpler class.

8 Conclusion

The transformation rules presented in this paper provide a uniform framework to handle tree tuple languages: on the practical side we obtain a single algorithm for computing with tree tuple languages,[11] on the theoretical side the proof of closure properties for classes of tuple languages reduces to giving bounds on the length and depth of the generated clauses. Based on this approach we extended known results and investigated new classes.

Properties beyond those in the last sections can be proved in a similar way, like closure properties of weakly regular relations [23]. We believe that these results can be generalized to new classes of cs-programs corresponding to new classes of constraint systems. Though we focused on constraint systems, our approach also applies to other tree tuple formalisms like tree automata with equality constraints. This could allow one to mix different classes of tree tuple languages in a single scheme.

Our work might also be viewed as contributing to two other fields: clausal model building and logic program transformation. Starting from failed proof attempts clausal model building constructs finite presentations of counter-models like sets of atoms, and tries to compute with these presentations (see e.g. [8, 16]). Our algorithm transforms logic programs to cs-programs, which are presentations of the minimal Herbrand model; operations like testing for membership correspond to operations on models like checking the validity of an atom. Regarding program transformations, our algorithm is an instance of the 'rules+strategy' approach [21], and the termination results constitute a successful application of the principles of logic program transformation to the field of tree tuple languages.

References

1. F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
2. B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In A. Finkel and M. Jantzen, editors, *Ninth Anual Symbosium on Theoretical Aspects of Computer Science*, volume 577 of *LNCS*, pages 161–171. Springer Verlag, 1992.

[11] A prototype implementation in Prolog is available from `www.logic.at/css`.

3. Witold Charatonik and Andreas Podelski. Set constraints with intersection. *Information and Computation*, 179(2):151–385, 2002.
4. 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.
5. M. Dauchet and S. Tison. Structural complexity of classes of tree languages. In M. Nivat and A. Podelski, editors, *Tree Automata and Languages*, pages 327–353. North-Holland, Amsterdam, 1992.
6. C. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 25, pages 1791–1849. Elsevier Science, 2001.
7. C. G. Fermüller and G. Salzer. Ordered paramodulation and resolution as decision procedure. In A. Voronkov, editor, *Logic Programming and Automated Reasoning (LPAR'93)*, LNCS 698 (LNAI), pages 122–133. Springer, 1993.
8. C.G. Fermüller and A. Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 2(6):173–203, 1996.
9. Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Logic in Computer Science*, pages 300–309, 1991.
10. John P. Gallagher and Germán Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In *4th International Symposium, PADL 2002*, volume 2257 of *LNCS*, pages 243–261. Springer, 2002.
11. R. Gilleron, S. Tison, and M. Tommasi. Set constraints and automata. *Information and Computation*, 1(149):1–41, 1999.
12. J. Goubault-Larrecq. Higher-order positive set constraints. In *Proc. 16th Int. Workshop Computer Science Logic (CSL'2002), Edinburgh, Scotland, Sep. 2002*, volume 2471 of *LNCS*, pages 473–489. Springer, 2002.
13. J. Goubault-Larrecq and K. N. Verma. Alternating two-way AC-tree automata. Research Report LSV-02-11, Lab. Specification and Verification, ENS de Cachan, Cachan, France, September 2002. 21 pages.
14. V. Gouranton, P. Réty, and H. Seidl. Synchronized tree languages revisited and new applications. In *Proceedings of 6th Conference on Foundations of Software Science and Computation Structures, Genova (Italy)*, volume 2030 of *LNCS*, pages 214–229. Springer Verlag, 2001.
15. N.C. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proc. of the 17th ACM Symp. on Principles of Programming Languages*, pages 197–209, 1990.
16. A. Leitsch. Decision procedures and model building, or how to improve logical information in automated deduction. In R. Caferra and G. Salzer, editors, *Automated Deduction in Classical and Non-Classical Logics*, volume 1761 of *LNCS*, pages 62–79. Springer, 2000.
17. S. Limet and P. Réty. E-unification by means of tree tuple synchronized grammars. *Discrete Mathematics and Theoretical Computer Science*, 1:69–98, 1997.
18. J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1984.
19. R. Matzinger. *Computational Representations of Models in First-Order Logic.* Dissertation, Technische Universität Wien, Austria, 2000.
20. 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.

21. A. Pettorossi and M. Proietti. Transformation of logic programs. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1998.
22. J.-C. Raoult. Rational tree relations. *Bulletin of the Belgian Mathematical Society* `www.ulb.ac.be/assoc/bms/bms.bull.html`, 4(1):149–176, 1997.
23. P. Réty. *Langages synchronisés d'arbres et applications*. Habilitation thesis (in French), LIFO, Université d'Orléans, 2001.
24. Frédéric Saubion and Igor Stéphan. A unified framework to compute over tree synchronized grammars and primal grammars. *Discrete Mathematics and Theoretical Computer Science*, 5:227–262, 2002.
25. F. Seynhaeve, S. Tison, M. Tommasi, and R. Treinen. Grid structures and undecidable constraint theories. *Theoretical Computer Science*, 258(1/2):453–490, May 2001.