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/

Manipulating Tree Tuple Languages by Transforming Logic Programs

Sébastien Limet
Université d'Orléans, LIFO

Gernot Salzer
TU Wien

Rapport N^o **2004-01**

Manipulating Tree Tuple Languages by Transforming 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 introduce inductive definitions over language expressions as a framework for specifying tree tuple languages. Inductive definitions and their sub-classes correspond naturally to classes of logic programs, and operations on tree tuple languages correspond to the transformation of logic programs. We present an algorithm based on unfolding and definition introduction that is able to deal with several classes of tuple languages in a uniform way. Termination proofs for clause classes translate directly to closure properties of tuple languages, leading to new decidability and computability results for the latter.

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 [1, 13]. The drawback of regular tree languages is their weak expressiveness, which has lead to many extensions: tree automata with constraints [2], tree set automata [6], regular relations [1], or synchronized languages [9, 11]. Most extensions lose either closure under some operations or the nice complexity of the operations, or both.

The work presented in this paper grew out of the study of synchronized languages. They were introduced in [11] to obtain finite representations of the solutions of certain R-unification problems, and were subsequently applied to R-disunification and one-step-rewriting [15]. Moreover, synchronized languages are able to represent the transitive closure of relations induced by certain process algebras with communications and so can be used for model checking in this context [9].

Originally, synchronized languages were specified by means of Tree Tuple Synchronized Grammars (TTSGs), which are regular tree grammars with packs of productions that have to be applied at the same time. TTSGs are able to specify non-regular tree languages, but are a bit unwieldy; therefore grammar productions were subsequently replaced by set constraints [9]. In general, synchronized languages do not possess the same nice properties as regular tree language do, but they are considerably more expressive, and some relevant subclasses 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 to deal with constraint systems in a uniform framework using notions and notations 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.

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. Any subset of ${T_\Sigma}^n$ is called an *n-ary tree tuple language over Σ* (n-tuple language for short). Union, intersection, and Cartesian product are denoted by $\cup$, $\cap$, and $\times$, respectively. The notation $A_1 \times \cdots \times A_k$ is shorthand for $\{()\}$ (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.[1] The integers are called *indices* and are used to refer to particular components of tuple languages. As usual we denote substitutions as sets, i.e., $A\{s_1 \mapsto t_1, \ldots, s_l \mapsto t_l\}$ is obtained from A by replacing simultaneously all occurrences of s_i by t_i. The arity of any object O will be denoted by $\mathrm{ar}(O)$, where O may be a function or predicate symbol, a language variable, or a language expression.

Let A be an n-tuple language, and let $\Box$ be a k-tuple of templates with l denoting the largest index occurring in $\Box$. The operations *construction* and *filtering* are defined as:

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

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 $(\Box \circ A)/\Box = A$.

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

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

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

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

for language variables A and template tuples $\Box$. An *inductive definition* is a set of constraints of form $A \supseteq e$ such that the arities of A and e coincide.[2] The languages defined by an inductive definition $\mathcal{D}$ are the least fixed point of the constraints. The least fixed point solution of variable A, i.e. the language associated with A, is denoted by $\mathcal{L}_\mathcal{D}(A)$.

Example 2. Operations on term tuples like forming sets of term tuples, computing union, intersection, joins, projections, or testing for membership can be

[1] 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.

[2] 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.

expressed by proper language expressions:

$$
\begin{aligned}
\{\boldsymbol{t}_1, \ldots, \boldsymbol{t}_n\} &= \boldsymbol{t}_1 \circ \{()\} \cup \cdots \cup \boldsymbol{t}_n \circ \{()\} \\
e \cup f &= A \qquad \text{with the constraints } A \supseteq e \text{ and } A \supseteq f \\
e \cap f &= (e \times f)/[1, \ldots, m, 1, \ldots, n] \qquad \text{for } m = n \\
e \ltimes_{i,j} f &= (e \times f)/[1, \ldots, m+j-1, i, m+j, \ldots, m+n-1] \\
e \ltimes_{i,n} f &= (e \times f)/[1, \ldots, m+n-1, i] \\
\Pi_{i_1, \ldots, i_k} e &= [i_1, \ldots, i_k] \circ e \\
\boldsymbol{t} \in A &\equiv (A/\boldsymbol{t} \neq \emptyset) \equiv (A/\boldsymbol{t} = \{()\})
\end{aligned}
$$

where $\boldsymbol{t}, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_n$ are term tuples, e and f are m- and n-ary language expressions, respectively, and $1 \leq i, i_1, \ldots, i_k \leq m$, $1 \leq j < n$.

Example 3. Let $\Sigma = \{a/0, f/2\}$ and $\mathcal{X} = \{\mathit{Id}_2/2, \mathit{Sym}/2\}$. The constraints

$$
\begin{aligned}
\mathit{Id}_2 &\supseteq \{(a,a)\} \cup [f(1,3), f(2,4)] \circ (\mathit{Id}_2 \times \mathit{Id}_2) \\
\mathit{Sym} &\supseteq \{(a,a)\} \cup [f(1,3), f(4,2)] \circ (\mathit{Sym} \times \mathit{Sym})
\end{aligned}
$$

define the languages

$$
\begin{aligned}
\mathcal{L}(\mathit{Id}_2) &= \{\, (t,t) \mid t \in T_\Sigma \,\} \\
\mathcal{L}(\mathit{Sym}) &= \{\, (t,t') \mid t \in T_\Sigma,\ t' \text{ is the symmetric tree of } t \,\}
\end{aligned}
$$

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 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 [9] 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. A constraint $A \supseteq \square \circ (A_1 \times \cdots \times A_k)$ is called

- *linear* iff no index occurs twice in $\square$;[3]
- *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;
- *regular* iff it is linear and for some mapping $\pi\colon \omega \mapsto \omega$, each component of $\square$ is of the form $f(i_1.j_1, \ldots, i_{\mathrm{ar}(f)}.j_{\mathrm{ar}(f)})$, where $\pi(i_l) = l$ for $l = 1, \ldots, \mathrm{ar}(f)$.
- *pseudo-regular* iff for some mapping $\pi\colon \omega \mapsto \omega$, each component of $\square$ is of the form $f(i_1.j_1, \ldots, i_{\mathrm{ar}(f)}.j_{\mathrm{ar}(f)})$, where $\pi(i_l) = l$ for $l = 1, \ldots, \mathrm{ar}(f)$.

A constraint system is called linear (horizontal, regular, pseudo-regular) iff all constraints have the respective property. Obviously regularity implies horizontality and, by definition, linearity. Pseudo-regularity extends regularity by allowing duplicate indices if they occur at the same positions of some components.

[3] In [9], this property is called *non-copying*.

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

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

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

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

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. [12]). We give a few definitions to fix notation.

Let $T_{\Sigma,V}$ denote the set of first-order terms over signature Σ and 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). A term, an atom or a set of atoms is called *linear* if no variable occurs more than once in it.

For atoms $H, B_1, \ldots, B_k$, the expression $H \leftarrow B_1, \ldots, B_k$ is called *program clause*, with H being the *head* and $B_1, \ldots, B_k$ the *body* of the clause; for $k = 0$ the clause is called a *fact*. A *logic program* is a finite set of program clauses. A *query* is a program clause without head. 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

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

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

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

Relation $\rightsquigarrow$ is specified in Figure 1; the first-order variables x_i introduced by the rules are assumed to be fresh variables occurring nowhere else. For an inductive definition $\mathcal{D}$, i.e., for a set of constraints, we define the corresponding logic program as $\mathrm{horn}(\mathcal{D}) \stackrel{\mathrm{def}}{=} \bigcup_{C\in\mathcal{D}} \mathrm{horn}(C)$.

$$\frac{\top}{\{()\} \rightsquigarrow \langle (),\, \{\}\rangle} \qquad \frac{\top}{A \rightsquigarrow \langle (x_1,\ldots,x_{\mathrm{ar}(A)}),\, \{P_A(x_1,\ldots,x_{\mathrm{ar}(A)})\}\rangle}$$

$$\frac{e \rightsquigarrow \langle (s_1,\ldots,s_m),\, \mathcal{G}\rangle \qquad f \rightsquigarrow \langle (t_1,\ldots,t_n),\, \mathcal{H}\rangle}{(e \times f) \rightsquigarrow \langle (s_1,\ldots,s_m,t_1,\ldots,t_n),\, \mathcal{G}\cup\mathcal{H}\rangle}$$

$$\frac{e \rightsquigarrow \langle (s_1,\ldots,s_m),\, \mathcal{G}\rangle}{(\Box \circ e) \rightsquigarrow \langle \Box\{1\mapsto s_1,\ldots,m\mapsto s_m, m{+}1\mapsto x_1, m{+}2\mapsto x_2,\ldots\},\, \mathcal{G}\rangle}$$

$$\frac{e \rightsquigarrow \langle (s_1,\ldots,s_m),\, \mathcal{G}\rangle}{(e/\Box) \rightsquigarrow \langle (x_1,\ldots,x_l)\mu,\, \mathcal{G}\mu\rangle}$$

provided μ, the m.g.u. of $(s_1,\ldots,s_m)$ and $\Box\{1\mapsto x_1,\ldots,l\mapsto x_l\}$, exists; l denotes the maximal index occurring in $\Box$.

Fig. 1. 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 2.

$$\dfrac{\dfrac{\top}{\{()\} \rightsquigarrow \langle (),\ \{\} \rangle}}{(a,1,1) \circ \{()\} \rightsquigarrow \langle (a,x_1,x_1),\ \{\} \rangle} \qquad \dfrac{\dfrac{\top}{\{()\} \rightsquigarrow \langle (),\ \{\} \rangle}}{(a,1,a) \circ \{()\} \rightsquigarrow \langle (a,x_1,a),\ \{\} \rangle}$$

$$\dfrac{\dfrac{\top}{X \rightsquigarrow \langle (x_1,x_2,x_3),\ \{P_X(x_1,x_2,x_3)\} \rangle}}{(s(1),2,s(3)) \circ X \rightsquigarrow \langle (s(x_1),x_2,s(x_3)),\ \{P_X(x_1,x_2,x_3)\} \rangle}$$

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

Fig. 2. 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

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

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

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

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) $\mathcal{L}_{\mathcal{D}}(A) = \mathcal{M}(\mathrm{horn}(\mathcal{D}))|_{P_A}$ for every language variable A.
(b) $\mathcal{M}(\mathcal{P})|_P = \mathcal{L}_{\mathrm{indef}(\mathcal{P})}(A_P)$ 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;
- *regular* iff the head is linear and for some mapping $\pi\colon \omega \mapsto \omega$, each argument of the head is of the form $f(x_1, \ldots, x_{\mathrm{ar}(f)})$, where x_i occurs in a body atom B_j such that $\pi(j) = i$.

– *pseudo-regular* iff for some mapping $\pi\colon \omega \mapsto \omega$, each argument of the head is of the form $f(x_1, \ldots, x_{\mathrm{ar}(f)})$, where x_i occurs in a body atom B_j such that $\pi(j) = i$.

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 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 [14]. 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. A set of definitions, $\mathcal{D}$, is *compatible with* $\mathcal{P}$, if all predicate symbols occurring in the heads of the definitions occur in exactly one head and nowhere else in $\mathcal{D}$ and $\mathcal{P}$; 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 $\overset{*}{\Rightarrow}$. An *initial state* is of the form $\langle \mathcal{P}, \mathcal{D}, \emptyset, \emptyset, \emptyset \rangle$ where $\mathcal{D}$ is compatible with $\mathcal{P}$; a *final state* is of the form $\langle \mathcal{P}, \emptyset, \mathcal{D}', \emptyset, \mathcal{P}' \rangle$. $\mathcal{P}$ and $\mathcal{D}$ are called the input of a derivation, $\mathcal{P}'$ its output. A derivation is 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:

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

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:

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

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 $1 \le i \le k$ 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.[4]

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 $\mathcal{D} = \{R(x,z) \leftarrow M(x,y,z), E(y)\}$. Predicate M defines subtraction (minus), E evenness, and R defines all pairs of numbers with an even difference. Note that the definition is no cs-clause since its body atoms share variables.

Figure 3 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.

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

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

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

Fig. 3. Transforming the program of Example 6 to a cs-program

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. [14]). □

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 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.*

Proof. We show that for quasi-cs-clauses all definitions occurring in a derivation are of the form $L \leftarrow A$, where A is a single linear atom whose depth is bounded by the maximal depth, τ, of any atom occurring in $\mathcal{P}$. 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 $\mathcal{D}_\mathcal{P}$ are of this particular form.

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

It remains to show that the depth of $\mathcal{B}$ and therefore of the new definitions is bounded by τ. Let x be a variable occurring both in head H and body $\mathcal{B}$ of the clause, and let $\tau_{x,H}$ denote the minimal depth of an occurrence of x in the head and $\tau_{x,\mathcal{B}}$ denote the depth of the occurrence of x in the body. Furthermore, let τ_A be the term depth of A and $\tau_{x\mu}$ be the depth of the term substituted for x in the unfolding step. By definition of quasi-cs-ness we have $\tau_{x,\mathcal{B}} \leq \tau_{x,H}$. Moreover, since A is a linear atom, we have $\tau_{x,H} + \tau_{x\mu} \leq \tau_A$ and therefore also $\tau_{x,\mathcal{B}} + \tau_{x\mu} \leq \tau_A$. By assumption, $\tau_A \leq \tau$. Therefore the depth of $\mathcal{B}\mu$, 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. $\square$

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 1. *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. Consider the constraint $A' \supseteq A/\Box \circ \{()\}$ for a new nullary variable A'. We have $\mathcal{L}(A') = \emptyset$ if $L/\Box$ is empty, and $\mathcal{L}(A') = \{()\}$ 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 $\mathcal{L}(A') = \emptyset$, or clauses equivalent to the singleton $P_{A'} \leftarrow$, indicating $\mathcal{L}(A') = \{()\}$. $\Box$

Example 7. Consider the cs-program constructed in example 6. In order to test whether the tuple $(s(a), s(a), a)$ occurs in the language associated with predicate R we add the definition $P \leftarrow R(s(a), s(a), a)$ 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 4 shows the relevant steps of the $\Rightarrow$-derivation; each line corresponds to the application of one unfolding and one definition introduction step. The last

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

Fig. 4. Membership test of Example 7

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

Now suppose we want to know whether there is any ground instance of the template $(s(x), s(y), a)$ that belongs to the language associated with R. We add the definition $P \leftarrow R(s(x), s(y), a)$ and compute again a $\Rightarrow$-derivation (see Figure 5). 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
$P \leftarrow R(s(x), s(y), a)$			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. 5. 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 constraint systems defining pseudo-regular relations on terms. According to its definition, pseudo-regularity extends regularity by admitting a restricted form of duplicate variables. We will show that this class is closed under intersection and joins.

This is an interesting result since it is the first class with duplication that has both a decidable emptiness test and is closed under intersection. [17] 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 emptiness is undecidable for all super-classes: adding just one equality constraint is sufficient for the membership test to become undecidable.

A clause (or definition) $H \leftarrow B_1, \ldots, B_k$ is called a *join-clause* (or *join-definition*) if it has the following properties:

- The atoms $H, B_1, \ldots, B_k$ are linear and do not contain function symbols.
- The variables of H are among those of the body.

In general, join-clauses are no cs-clauses since the B_i's may share variables, i.e., the body of a join-clause need not be linear though each atom is. This property allows to represent the intersection or join of two or more tuple languages as join-clauses.

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.*

Proof. Let *shared* variables be variables that occur in more than one body atom, and let the *degree* of a clause be the maximal number of occurrences of any body variable. Let sv be the number of shared variables in the original join-definition D, and d be its degree. We show that all definitions introduced by the derivation are join-definitions of degree d or less and containing at most

sv shared variables. Moreover the definitions contain at most $d \cdot sv$ atoms. Since join-definitions do not contain function symbols the number of definitions is finite up to renaming.

Let $L \leftarrow A_1, \ldots, A_k$ be a join-definition of degree d and with sv shared variables. For all $1 \leq i \leq k$, let $H_i \leftarrow B_{i,1}, B_{i,2}, \ldots$ be pseudo-regular cs-clauses from $\mathcal{P}$ such that μ, the most general simultaneous unifier of $(A_1, \ldots, A_k)$ and $(H_1, \ldots, H_k)$ exists. We make the following observations:

- Let $A_i = p_i(t_{i1}, t_{i2}, \ldots)$ and $H_i = p_i(s_{i1}, s_{i2}, \ldots)$. Then the computation of μ amounts to unifying simultaneously all pairs (s_{ij}, t_{ij}). All terms involved in unification are linear, hence μ replaces every variable in the join-definition by a linear term $f(\boldsymbol{x})$.
  - Each argument of $L\mu$ is a linear term of the form $f(\boldsymbol{x})$.
  - The maximal number of occurrences of any variable in $H_1\mu, \ldots, H_k\mu$ is at most the same as in $A_1, \ldots, A_k$, i.e. d.
  - The number of shared variables that occur at a position p in the arguments of $H_1\mu, \ldots, H_k\mu$ is bounded by sv.
- If $H_i\mu$ and $H_{i'}\mu$ share a variable then the relative position of this variable in the argument of $H_i\mu$ is the same as in the argument of $H_{i'}\mu$. Indeed, since the two clauses have to be renamed before unification, $H_i\mu$ and $H_{i'}\mu$ share variables iff A_i and $A_{i'}$ share at least one variable, say x. Since $A_i\mu = H_i\mu$ and $A_{i'}\mu = H_{i'}\mu$, one argument of $H_i\mu$ and one argument of $H_{i'}\mu$ is $x\mu$, so the variables of $x\mu$ appear at the same position in the argument of $H_i\mu$ and in the argument of $H_{i'}\mu$.
- If a body atom $B_{i,j}\mu$ of a clause headed by $H_i\mu$ shares variables with a body atom $B_{i',j'}\mu$ of a clause headed by $H_{i'}\mu$, then we have $\pi_i(j) = \pi_{i'}(j')$. If $B_{i,j}\mu$ and $B_{i',j'}\mu$ share the variable x, x occurs in $H_i\mu$ and in $H_{i'}\mu$. Hence, for the same reason as above, x occurs at the same position p in the arguments of $H_i\mu$ and $H_j\mu$. So $\pi_i(j) = \pi_{i'}(j') = p$.

Hence, when unfolding is applied to $L \leftarrow A_1, \ldots, A_k$, the most general unifier μ is of depth one. If $\mathcal{B}_1 \dot{\cup} \cdots \dot{\cup} \mathcal{B}_n$ is the maximal variable-disjoint decomposition of the body of the resulting clause, then each atom $B_{i,j}$ of component $\mathcal{B}_m$ satisfies $\pi_i(j) = k_m$. Let π be the mapping $\pi(m) = k_m$ for $1 \leq m \leq n$. Since all variables of $L\mu$ appear in the body, the image of π is $\{1, \ldots, \mathrm{ar}(f_{\max})\}$ where $f_{\max}$ is the symbol of maximal arity of $L\mu$. So the new clauses added to $\mathcal{C}_{\mathrm{out}}$ are of the form $L\mu \leftarrow L_1, \ldots, L_n$ where the variables of each L_i are those of the corresponding component $\mathcal{B}_i$. Therefore for any argument $f(x_1, \ldots, x_k)$ of $L\mu$, x_i occurs in one L_j such that $\pi(j) = i$ which means that this clause is pseudo-regular.

Moreover, for each component $\mathcal{B}_m$, the number of shared variables is at most the number of shared variables occurring at position $\pi(m)$, i.e. sv, and the maximal number of occurrences of a variable is bounded by d (see the first argument above). Hence the new definitions are join-definitions of degree not greater than d and with no more than sv shared variables. Note that this implies that the maximal number of atoms in $\mathcal{B}_m$ is $d \cdot sv$. □

Corollary 2. *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 we can compute a regular cs-program for X which can be rewritten as pseudo-regular constraint system. $\square$

Remark 5. If $\mathcal{P}$ is a regular cs-program, then 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: $A_i\mu = H_i\mu$ is a linear term which implies that for $x \neq y$, $x\mu$ and $y\mu$ do not share variables and therefore $L\mu$ is a linear atom. Hence the result of the derivation is again a regular cs-program. Translated to tree tuple languages this observation reproves the well-known fact that tuple languages specified by regular constraints are closed under intersection.

Example 8. Consider the pseudo-regular constraints

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

$\mathcal{L}(P)$ 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:

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

The constraint for the intersection corresponds to the join-definition

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

Figure 6 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 $\mathcal{L}(Q) = \{(0,0),\ (c(0,0), c(0,0))\}$.

$\mathcal{D}_{\text{new}}$	$\mathcal{C}_{\text{new}}$	$\mathcal{C}_{\text{out}}$	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. 6. Computing the intersection in Example 8

The results on pseudo-regular constraint systems generalize those for regular relations [1]. Our approach allows 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 one way or another (see e.g. [5, 16] or [1, 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. [4] show that the types for a logic program can be constructed 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 [7] and AC tree automata [8].

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.

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,[5] 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.

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 to compute with these presentations (see e.g. [3, 10]). 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 [14], and the termination results constitute a successful application of the principles of logic program transformation to the field of tree tuple languages.

Properties beyond those in the last sections can be proved in a similar way, like closure properties of weakly regular relations [15]. 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 to mix different classes of tree tuple languages in a single scheme.

References

[1] 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.

[2] 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.

[5] A prototype implementation of the algorithm in Prolog is available from `http://www.logic.at/css/`.

[3] C.G. Fermüller and A. Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 2(6):173–203, 1996.

[4] 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.

[5] 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.

[6] R. Gilleron, S. Tison, and M. Tommasi. Set constraints and automata. *Information and Computation*, 1(149):1–41, 1999.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] S. Limet and P. Réty. E-unification by means of tree tuple synchronized grammars. *Discrete Mathematics and Theoretical Computer Science*, 1:69–98, 1997.

[12] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1984.

[13] R. Matzinger. *Computational Representations of Models in First-Order Logic.* Dissertation, Technische Universität Wien, Austria, 2000.

[14] 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.

[15] P. Réty. *Langages synchronisés d'arbres et applications.* Habilitation thesis (in French), LIFO, Université d'Orléans, 2001.

[16] 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.

[17] F. Seynhaeve, S. Tison, Tommasi M., and R. Treinen. Grid structures and undecidable constraint theories. *Theoretical Computer Science*, 258(1/2):453–490, May 2001.