$\boldsymbol{R}$-Unification thanks to Synchronized Context-Free Tree Languages Extended Abstract

Pierre Réty Jacques Chabin Jing Chen

LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France
E-mail : {rety, chabin, chen}@lifo.univ-orleans.fr
http://www.univ-orleans.fr/SCIENCES/LIFO/Members/rety

Abstract. Expressing descendants or the rewrite relation by tree-(tuple) languages allows to deal with R-matching and R-unification problems. We present in addition a new class of tree-tuple languages, more expressive than former ones, but having the same properties. Thanks to this new class, we hope to solve more interesting R-matching and R-unification problems.

1 The General Method

Given a confluent term rewrite system R, computing descendants (i.e. successors by rewriting) can be applied to R-matching and R-unification. Indeed the R-matching problem $t \triangleleft_R t'$ has at least one solution θ iff $\exists s$, $t\theta \rightarrow_R^* s \; {}_R^*\!\leftarrow t'$, which is equivalent to check that

$$R^*(\{t\sigma \mid \sigma \in Subst\}) \cap R^*(\{t'\}) \neq \emptyset$$

where $Subst$ is the set of all substitutions and $R^*(E) \stackrel{def}{=} \{s \mid \exists e \in E,\, e \rightarrow_R^* s\}$ are the descendants of the set of terms E. In the same way, and if we assume that t and t' do not share variables, the R-unification problem $t \doteq_R t'$ has at least one solution iff

$$R^*(\{t\sigma \mid \sigma \in Subst\}) \cap R^*(\{t'\sigma' \mid \sigma' \in Subst\}) \neq \emptyset$$

The above remarks can be used to decide R-matchability or R-unifiability, but do not allow to express the possible R-matches or R-unifiers. However if we can compute the binary relation $\rightarrow_R^*$ by expressing a set of pairs of terms, we can in addition express the matched terms, i.e. $\{t\theta,\, t\theta =_R t'\}$ (or the unified terms in a similar way), by computing

$$M \;=\; \Pi_1((\{t\sigma,\, \sigma \in Subst\}\;{}_1\!\bowtie_1 \rightarrow_R^*)\;{}_2\!\bowtie_1 ({}_R^*\!\leftarrow \;{}_2\!\bowtie_1 \{t'\}))$$

where Π_1 is the projection onto the first component, and $T\;{}_i\!\bowtie_j T'$ denotes the natural join (like in a relational database) between the i-th component of T and the j-th component of T' [1]. Now, if we can solve the "set syntactic-matching

[1] Note that computing $\rightarrow_R^*$ allows to compute the descendants because $R^*(E) = \Pi_2(E\;{}_1\!\bowtie_1 \rightarrow_R^*)$.

problem", i.e. compute $\{\theta \in \mathit{Subst}, t\theta \in M\}$, we get the R-matches. We can get R-unifiers in a similar way.

2 Using Existing Tree-(tuple) Languages

Some tree-(tuple) languages allow to achieve the general method presented in the previous section.

2.1 By computing descendants

- Using regular tree languages.
  If t and t' are linear, then $\{t\sigma \mid \sigma \in \mathit{Subst}\}$ and $\{t'\sigma' \mid \sigma' \in \mathit{Subst}\}$ are regular languages. Moreover regular tree languages are closed under intersection, and emptiness is decidable. Therefore for every rewrite system R that preserves recognizability[2], R-matchability of problems $t \lhd_R t'$ s.t. t is linear is decidable, and R-unifiability of linear equations is decidable. Strong restrictions ensuring recognizability preservation have been studied [1, 4, 3, 9, 12, 8, 11].
- Using context-free tree languages.
  Assume that the rewrite system does not preserve recognizability, but satisfies the following property: E is regular implies $R^*(E)$ is context-free [10]. However context-free languages are not closed under intersection, but are closed under intersection with a regular language. Therefore, we need to assume in addition that t' is irreducible in the matching problem $t \lhd_R t'$ so that $R^*(\{t'\}) = \{t'\}$ is regular. In the linear unification problem $t \doteq_R t'$, we need to assume that R is left-linear and constructor-based, and t' contains only constructors and variables, so that $E = \{t'\sigma' \mid \sigma' \in \mathit{Subst} \wedge \sigma' \text{ is normalized}\}$ is regular and contains only normalized terms, and consequently $R^*(E) = E$ is regular.

2.2 By expressing $\rightarrow_R^*$

Here, tree-tuple languages are needed. The class of synchronized tree-tuple languages, first presented thanks to Tree-Tuple Synchronized Grammars (TTSG) [5], then extended thanks to the so-called Constraint Systems (CS) [2], and finally extended using a logic-programming framework [7] called CS-programs, allows to express $\rightarrow_R^*$ [6] in a number of cases. Moreover, thanks to the logic-programming formalism, the "set syntactic-matching problem" is trivially solvable. Therefore we can express the R-matches (or R-unifiers) and (more interesting) handle them.

[2] I.e. E is regular implies $R^*(E)$ is regular.

3 Merging Context-free and Synchronized Languages

Context-free tree languages and synchronized tree-tuple languages have the same properties : they are closed under union, closed under intersection (of one tuple-component) with a regular tree language, membership and emptiness are decidable.

In order to increase expressivity, we have defined a new class of tree-tuple languages : the synchronized context-free languages, obtained by mixing the previous ones, and we are proving that the properties recalled above still hold. This new class should allow to express both descendants and $\rightarrow_R^*$ in more cases than before, thanks to its bigger expressivity.

To give an intuitive idea, let us give a few examples. The following language is context-free ($n \in \mathbb{N}$) :

$$\begin{array}{c} s^n \\ | \\ p^n \\ | \\ a \end{array}$$

The following language is synchronized :

$$\begin{array}{ccc} & f & \\ / & & \backslash \\ s^n & & g^n \\ | & & | \\ a & & a \end{array}$$

The following language is in the new class (synchronized context-free), but it is neither a context-free language, nor a synchronized language :

$$\begin{array}{ccc} & f & \\ / & & \backslash \\ s^n & & g^n \\ | & & | \\ p^n & & h^n \\ | & & | \\ a & & a \end{array}$$

Actually, the new class can express an unbounded counting both in several independent branches (like in a synchronized language), and in two nested positions (like in a context-free language). Moreover, this class allows to recursively copy subterms, which enables to express (among others) the set of binary balanced trees.

We use logic programs, instead of grammars or automata.

Definition 1. *A* logic program with modes *is a logic program such that a mode-tuple* $\boldsymbol{m} \in \{I, O\}^n$ *is associated to each predicate symbol* P *(*n *is the arity of* P*). In other words, each predicate argument has mode* I *(Input) or* O *(Output).*
To distinguish them, **output arguments will be covered by a hat**.

Notation : Let P be a predicate symbol. $ArIn(P)$ is the number of input arguments of P, and $ArOut(P)$ is the number of output arguments.

Notation : Let B be a list of atoms (possibly containing only one atom). $Input(B)$ is the input part of B, i.e. the tuple composed of the input arguments of B. $ArIn(B)$ is the arity of $Input(B)$. $VarIn(B)$ is the set of variables that appear in $Input(B)$.
$Output(B)$, $ArOut(B)$, and $VarOut(B)$ are defined in a similar way.
We also define $Var(B) = VarIn(B) \cup VarOut(B)$.

Definition 2. *Let $B = A_1, \ldots, A_n$ be a list of atoms. We say that $A_j > A_k$ (possibly $j = k$) if $\exists y \in VarIn(A_j) \cap VarOut(A_k)$. In other words an input of A_j depends on an output of A_k.*
We say that B has a cycle *if $A_j >^+ A_j$ for some A_j ($>^+$ is the transitive closure of $>$).*
We say that the clause $H \leftarrow B$ has a cycle *if B has a cycle.*

Example 1. $Q(\widehat{x}, s(y))$, $R(\widehat{y}, s(x))$ (x, y are variables) has a cycle because $Q > R > Q$.

Definition 3. *A* Synchronized Context-Free program (S-CF program) *Prog is a logic program with modes, whose clauses $H \leftarrow B$ both satisfy :*

- *$Input(H).Output(B)$ (. is the tuple concatenation) is a linear tuple of variables, i.e. each tuple-component is a variable, and each variable occurs only once.*
- *B does not have a cycle.*

Given a predicate symbol P without input arguments, the tree-(tuple) language generated by P is $L(P) = \{\boldsymbol{t} \in (T_\Sigma)^{ArOut(P)} \mid P(\boldsymbol{t}) \in Mod(Prog)\}$, where $Mod(Prog)$ is the least Herbrand model of Prog. $L(P)$ is called Synchronized Context-Free language *(S-CF language).*

If in addition every clause $H \leftarrow B$ of Prog satisfies :

- *$Output(H).Input(B)$ (. is the tuple concatenation) is a linear tuple, i.e. each variable of $Output(H).Input(B)$ occurs only once,*

then Prog is said non-copying.

Remark 1.
- If a predicate symbol P has no input argument, this means that the outputs of P do not depend on a given input value (like in a regular language). Therefore, a S-CF program whose predicate symbols do not have input arguments, is called *synchronized program*. It generates a synchronized tree-(tuple) language. This class of languages has already been studied in [7, 6]. In these papers, synchronized programs are called cs-programs.
- If a predicate symbol P has several output arguments, this means that these output arguments are linked (synchronized) together. Therefore, a S-CF program whose predicate symbols have only one output argument, is called *context-free program* (without synchronization). It generates a context-free tree language.
- A non-copying context-free program whose predicate symbols have no input arguments, is called *regular program*. It generates a regular tree language.

Example 2. (x, y are variables)

$$Prog = \{P(\widehat{s(x)}, y) \leftarrow P(\widehat{s(x)}, y)\}$$

is not a S-CF program because $Input(H).Output(B) = (y, s(x))$ is not a tuple of variables.

$$Prog' = \{P'(\widehat{s(x)}, y) \leftarrow P'(\widehat{x}, s(y))\}$$

is a S-CF program because $Input(H).Output(B) = (y, x)$ is a linear tuple of variables, and there is no cycle. $Prog'$ is not copying because $Output(H).Input(B) = (s(x), s(y))$ is a linear tuple.

Example 3. (for $n \in \mathbb{N}$, $f^n(a)$ is the term $f(f(...f(a)))$ where f occurs n times)

$$Prog = \{S(\widehat{c(x, y)}) \leftarrow P(\widehat{x}, \widehat{y}, a, b).\ \ P(\widehat{f(x)}, \widehat{g(y)}, x', y') \leftarrow P(\widehat{x}, \widehat{y}, h(x'), i(y')).\ \ P(\widehat{x}, \widehat{y}, x, y) \leftarrow\}$$

is a non-copying S-CF program. Note that $Prog$ is neither a synchronized program, nor a context-free program. The language generated by S is:

$$L(S) = \{\ c(f^n(h^n(a)), g^n(i^n(b))) \mid n \in \mathbb{N}\}$$

Actually, this language is neither a synchronized language, nor a context-free language.

Example 4.

$$Prog = \{S(\widehat{f(x, x)}) \leftarrow S(\widehat{x}).\ \ S(\widehat{a}) \leftarrow\}$$

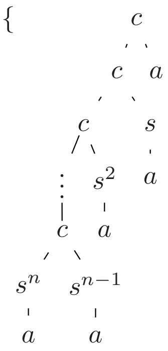
is a copying context-free program. The language $L(S)$ generated by S is the set of binary balanced terms (all leaves occur at the same depth) over the signature $\Sigma = \{f, a\}$. Actually, $L(S)$ cannot be generated by a non-copying S-CF program.

Example 5.

$$Prog = \{P(\widehat{x}) \leftarrow Q(\widehat{x}, y), R(\widehat{y}).\ \ Q(\widehat{c(y, x)}, x) \leftarrow Q(\widehat{y}, s(x)).\ \ Q(\widehat{x}, x) \leftarrow .\ \ R(\widehat{a}) \leftarrow\}$$

is a copying S-CF program.

$$L(P) = \{ \quad \text{(tree)} \quad \mid n \in N\}$$



References

1. M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Fifth Annual IEEE Symposium on Logic Computer Science*, pages 242–248. IEEE Computer Society Press, Philadelphia, Pennsylvania, 1990.
2. 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)*, LNCS. Springer, 2001.
3. P. Gyenizse and S. Vagvolgyi. Linear Generalized Semi-monadic Rewrite Systems Effectively Preserve Recognizability. *Theoretical Computer Science*, 194, pp 87-122, 1998.
4. F. Jacquemard. Decidable approximations of term rewrite systems. In editor H. Ganzinger, editor, *Proceedings 7th Conference RTA, New Brunswick (USA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, 1996.
5. S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. *Discrete Mathematics and Theoritical Computer Science (http://www.dmtcs.loria.fr)*, 1:69–98, 1997.
6. S. Limet and G. Salzer. Proving properties of term rewrite systems via logic programs. In *proceedings of RTA 2004*, volume 3091 of *LNCS*, pages 170–184. Springer Verlag, 2004.
7. Sébastien Limet and Gernot Salzer. Manipulating tree tuple languages by transforming logic programs. In Ingo Dahn and Laurent Vigneron, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.
8. C. Loding. Model-checking Infinite Systems Generated by Ground Tree Rewriting. In *Proceedings of the 7th Conference on Foundations of Software Science and Computation Structures*. LNCS, Springer, 2002.
9. P. Réty. Regular Sets of Descendants for Constructor-based Rewrite Systems. In *Proceedings of the 6th international conference LPAR*, number 1705 in Lecture Notes in Artificial Intelligence (LNAI), Tbilisi, Republic of Georgia, 1999. Springer Verlag.
10. P. Réty and J. Vuotto. Context-free tree languages for descendants. In *5th Workshop on Rule-Based Programming (RULE'2004). Proceedings in ENTCS vol 124(1)*, 2005.

11. H. Seki, T. Takai, F. Youhei, and Y. Kaji. Layered Transducing Term Rewriting System and its Recognizability Preserving Property. In *13th International Conference RTA*, volume 2378 of *Lectures Notes in computer Science*. Springer-Verlag, 2002.
12. T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In L. Bachmair, editor, *Proceedings 11th Conference RTA, Norwich (UK)*, volume 1833 of *LNCS*, pages 246–260. Springer-Verlag, 2000.