Erratum of [1]

Yohan Boichut, Jacques Chabin and Pierre Réty

LIFO - Université d'Orléans, B.P. 6759, 45067 Orléans cedex 2, France
{yohan.boichut, jacques.chabin, pierre.rety}@univ-orleans.fr

1 Introduction

In [1], we have described a technique for computing non-regular approximations using synchronized tree languages. This technique can handle the reachability problem of [2]. These synchronized tree languages [4, 3] are recognized using CS-programs [5], i.e. a particular class of Horn clauses. From an initial CS-program *Prog* and a left-linear term rewrite system (TRS) R, another CS-program $Prog'$ is computed in such a way that its *language* represents an over-approximation of the set of terms (called descendants) reachable by rewriting using R, from the terms of the language of *Prog*. This algorithm is called completion. However, the assumptions of the result showing that all the descendants are obtained, i.e. Theorem 14 in [1], are not correct. Actually, *preserving* should be replaced by *non-copying* (a variable cannot occur several times in the head of a clause). However, the non-copying nature of a CS-program is not preserved by completion as soon as the given TRS is not right-linear. Consequently, the final result presented in [1] holds for completely linear TRS, and not for just left-linear TRS.

In this paper, we propose a correction of [1], assuming that the initial CS-program is non-copying, and the TRS is completely linear (see Section 3).

2 Preliminaries

Consider two disjoint sets, Σ a *finite ranked alphabet* and *Var* a set of variables. Each symbol $f \in \Sigma$ has a unique arity, denoted by $ar(f)$. The notions of *first-order term*, *position* and *substitution* are defined as usual. Given two substitutions σ and σ', $\sigma \circ \sigma'$ denotes the substitution such that for any variable x, $\sigma \circ \sigma'(x) = \sigma(\sigma'(x))$. T_Σ denotes the set of ground terms (without variables) over Σ. For a term t, $Var(t)$ is the set of variables of t, $Pos(t)$ is the set of positions of t. For $p \in Pos(t)$, $t(p)$ is the symbol of $\Sigma \cup Var$ occurring at position p in t, and $t|_p$ is the subterm of t at position p. The term t is *linear* if each variable of t occurs only once in t. The term $t[t']_p$ is obtained from t by replacing the subterm at position p by t'. $PosVar(t) = \{p \in Pos(t) \mid t(p) \in Var\}$, $PosNonVar(t) = \{p \in Pos(t) \mid t(p) \notin Var\}$.

A *rewrite rule* is an oriented pair of terms, written $l \to r$. We always assume that l is not a variable, and $Var(r) \subseteq Var(l)$. A *rewrite system* R is a finite set of rewrite rules. *lhs* stands for left-hand-side, *rhs* for right-hand-side. The rewrite relation $\to_R$ is defined as follows: $t \to_R t'$ if there exist a position $p \in$

$PosNonVar(t)$, a rule $l \to r \in R$, and a substitution θ s.t. $t|_p = \theta(l)$ and $t' = t[\theta(r)]_p$. $\to_R^*$ denotes the reflexive-transitive closure of $\to_R$. t' is a *descendant* of t if $t \to_R^* t'$. If E is a set of ground terms, $R^*(E)$ denotes the set of descendants of elements of E. The rewrite rule $l \to r$ is *left (resp. right) linear* if l (resp. r) is linear. R is *left (resp. right) linear* if all its rewrite rules are left (resp. right) linear. R is *linear* if R is both left and right linear.

2.1 CS-Program

In the following, we consider the framework of *pure logic programming*, and the class of synchronized tree-tuple languages defined by CS-clauses [5, 6]. Given a set $Pred$ of *predicate* symbols; *atoms*, *goals*, *bodies* and *Horn-clauses* are defined as usual. Note that both *goals* and *bodies* are sequences of atoms. We will use letters G or B for sequences of atoms, and A for atoms. Given a goal $G = A_1, \ldots, A_k$ and positive integers i, j, we define $G|_i = A_i$ and $G|_{i.j} = (A_i)|_j = t_j$ where $A_i = P(t_1, \ldots, t_n)$.

Definition 1. *Let B be a sequence of atoms. B is* flat *if for each atom $P(t_1, \ldots, t_n)$ of B, all terms $t_1, \ldots, t_n$ are variables. B is* linear *if each variable occurring in B (possibly at sub-term position) occurs only once in B. Note that the empty sequence of atoms (denoted by $\emptyset$) is flat and linear.*
A CS-clause[1] *is a Horn-clause $H \leftarrow B$ s.t. B is flat and linear. A* CS-program *$Prog$ is a logic program composed of CS-clauses. $Pred(Prog)$ denotes the set of predicate symbols of $Prog$. Given a predicate symbol P of arity n, the tree-(tuple) language generated by P is $L(P) = \{\boldsymbol{t} \in (T_\Sigma)^n \mid P(\boldsymbol{t}) \in Mod(Prog)\}$, where T_Σ is the set of ground terms over the signature Σ and $Mod(Prog)$ is the least Herbrand model of $Prog$. $L(P)$ is called* Synchronized language*.*

The following definition describes syntactic properties over CS-clauses.

Definition 2. *A CS-clause $P(t_1, \ldots, t_n) \leftarrow B$ is :*

- empty *if $\forall i \in \{1, \ldots, n\}$, t_i is a variable.*
- normalized *if $\forall i \in \{1, \ldots, n\}$, t_i is a variable or contains only one occurrence of function-symbol.*
- preserving *if $Var(P(t_1, \ldots, t_n)) \subseteq Var(B)$.*
- non-copying *if $P(t_1, \ldots, t_n)$ is linear.*

A CS-program is normalized, preserving, non-copying *if all its clauses are.*

Example 1. The CS-clause $P(x, y, z) \leftarrow Q(x, y, z)$ is empty, normalized, preserving, and non-copying (x, y, z are variables).
The CS-clause $P(f(x), y, g(x, z)) \leftarrow Q_1(x), Q_2(y, z)$ is normalized, preserving, and copying. $P(f(g(x)), y) \leftarrow Q(x)$ is not normalized and not preserving.

Given a CS-program, we focus on two kinds of derivations.

[1] In former papers, synchronized tree-tuple languages were defined thanks to sorts of grammars, called constraint systems. Thus "CS" stands for Constraint System.

Definition 3. *Given a logic program Prog and a sequence of atoms G,*

- *G derives into G' by a* resolution *step if there exist a clause*[2] $H \leftarrow B$ *in Prog and an atom $A \in G$ such that A and H are unifiable by the most general unifier σ (then $\sigma(A) = \sigma(H)$) and $G' = \sigma(G)[\sigma(A) \leftarrow \sigma(B)]$. It is written $G \leadsto_\sigma G'$.*
- *G* rewrites *into G' if there exist a clause $H \leftarrow B$ in Prog, an atom $A \in G$, and a substitution σ, such that $A = \sigma(H)$ (A is not instantiated by σ) and $G' = G[A \leftarrow \sigma(B)]$. It is written $G \rightarrow_\sigma G'$.*

Sometimes, we will write $G \leadsto_{[H \leftarrow B,\sigma]} G'$ or $G \rightarrow_{[H \leftarrow B,\sigma]} G'$ to indicate the clause used by the step.

Example 2. Let $Prog = \{P(x_1, g(x_2)) \leftarrow P'(x_1, x_2).\ P(f(x_1), x_2) \leftarrow P''(x_1, x_2).\}$, and consider $G = P(f(x), y)$. Thus, $P(f(x), y) \leadsto_{\sigma_1} P'(f(x), x_2)$ with $\sigma_1 = [x_1/f(x), y/g(x_2)]$ and $P(f(x), y) \rightarrow_{\sigma_2} P''(x, y)$ with $\sigma_2 = [x_1/x, x_2/y]$.

Note that for any atom A, if $A \rightarrow B$ then $A \leadsto B$. If in addition $Prog$ is preserving, then $Var(A) \subseteq Var(B)$. On the other hand, $A \leadsto_\sigma B$ implies $\sigma(A) \rightarrow B$. Consequently, if A is ground, $A \leadsto B$ implies $A \rightarrow B$.

We consider the transitive closure $\leadsto^+$ and the reflexive-transitive closure $\leadsto^*$ of $\leadsto$.

For both derivations, given a logic program $Prog$ and three sequences of atoms G_1, G_2 and G_3 :

- if $G_1 \leadsto_{\sigma_1} G_2$ and $G_2 \leadsto_{\sigma_2} G_3$ then one has $G_1 \leadsto^*_{\sigma_2 \circ \sigma_1} G_3$;
- if $G_1 \rightarrow_{\sigma_1} G_2$ and $G_2 \rightarrow_{\sigma_2} G_3$ then one has $G_1 \rightarrow^*_{\sigma_2 \circ \sigma_1} G_3$.

In the remainder of the paper, given a set of CS-clauses $Prog$ and two sequences of atoms G_1 and G_2, $G_1 \leadsto^*_{Prog} G_2$ (resp. $G_1 \rightarrow^*_{Prog} G_2$) also denotes that G_2 can be derived (resp. rewritten) from G_1 using clauses of $Prog$.

It is well known that resolution is complete.

Theorem 1. *Let A be a ground atom. $A \in Mod(Prog)$ iff $A \leadsto^*_{Prog} \emptyset$.*

2.2 Computing descendants

We just give the main ideas using an example. See [1] for a formal description.

Example 3. Let $R = \{f(x) \rightarrow g(h(x))\}$ and let $I = \{f(a)\}$ generated by the CS-program $Prog = \{P(f(x)) \leftarrow Q(x).\ Q(a) \leftarrow\}$.
Note that $R^*(I) = \{f(a), g(h(a))\}$.

To simulate the rewrite step $f(a) \rightarrow g(h(a))$, we consider the rewrite-rule's left-hand side $f(x)$. We can see that $P(f(x)) \rightarrow_{Prog} Q(x)$ and $P(f(x)) \rightarrow_R P(g(h(x)))$. Then the clause $P(g(h(x))) \leftarrow Q(x)$ is called *critical pair*[3]. This

[2] We assume that the clause and G have distinct variables.
[3] In former work, a critical pair was a pair. Here it is a clause since we use logic programs.

critical pair is not *convergent* (in $Prog$) because $\neg(P(g(h(x))) \rightarrow^*_{Prog} Q(x))$. To get the descendants, the critical pairs should be convergent. Let $Prog' = Prog \cup \{P(g(h(x))) \leftarrow Q(x)\}$. Now the critical pair is convergent in $Prog'$, and note that the predicate P of $Prog'$ generates $R^*(I)$.

For technical reasons[4], we consider only normalized CS-programs, and $Prog'$ is not normalized. The critical pair can be normalized using a new predicate symbol, and replaced by normalized clauses $P(g(y)) \leftarrow Q_1(y)$. $Q_1(h(x)) \leftarrow Q(x)$. This is the role of Function norm in the completion algorithm below.

In general, adding a critical pair (after normalizing it) into the CS-program may create new critical pairs, and the completion process may not terminate. To force termination, two bounds *predicate-limit* and *arity-limit* are fixed. If *predicate-limit* is reached, Function norm should re-use existing predicates instead of creating new ones. If a new predicate symbol is created whose arity[5] is greater than *arity-limit*, then this predicate has to be cut by Function norm into several predicates whose arities do not exceed *arity-limit*. On the other hand, for a fixed[6] CS-program, the number of critical pairs may be infinite. Function removeCycles modifies some clauses so that the number of critical pairs is finite.

Definition 4 (comp as in [1]). *Let arity-limit and predicate-limit be positive integers. Let R be a left-linear rewrite system, and $Prog$ be a finite, normalized and preserving CS-program. The completion process is defined by:*
Function $\mathsf{comp}_R(Prog)$

```
Prog = removeCycles(Prog)
while there exists a non-convergent critical pair H ← B in Prog do
  Prog = removeCycles(Prog ∪ norm_Prog(H ← B))
end while
return Prog
```

The following results show that an over-approximation of the descendants is computed.

Theorem 2 ([1]). *Let $Prog$ be a normalized and preserving CS-program and R be a left-linear rewrite system.*
If all critical pairs are convergent, then $Mod(Prog)$ is closed under rewriting by R, i.e. $(A \in Mod(Prog) \wedge A \rightarrow^*_R A') \implies A' \in Mod(Prog)$.

Theorem 3 ([1]). *Let R be a left-linear TRS and $Prog$ be a normalized preserving CS-program. Function* comp *always terminates, and all critical pairs are convergent in $\mathsf{comp}_R(Prog)$. Moreover, $R^*(Mod(Prog)) \subseteq Mod(\mathsf{comp}_R(Prog))$.*

3 Fixing [1]

The hypotheses mentioned in Definition 4 and Theorems 2 and 3 are not sufficient to ensure the computation of an over-approximation.

---

[4] Critical pairs are computed only at root positions.
[5] The number of arguments.
[6] i.e. without adding new clauses in the CS-program.

Indeed, let us describe two counter-examples that are strongly connected.

Example 4. Let $Prog = \{P(f(x), f(x)) \leftarrow Q(x).\quad Q(a) \leftarrow .\quad Q(b) \leftarrow .\}$ and $R = \{a \rightarrow b\}$. $Prog$ is preserving and normalized as required in Theorem 2. R is ground and consequently, left-linear. There is only one critical pair $Q(b) \leftarrow .$, which is convergent. $P(f(a), f(a)) \in Mod(Prog)$ and $P(f(a), f(a)) \rightarrow_R P(f(b), f(a))$. However $P(f(b), f(a)) \notin Mod(Prog)$.

The copying nature of $Prog$ is problematic in Example 4 in the sense that it prevents the predicate symbol P from having two different terms right under. Another problem is that a non-right-linear rewrite rule (but left-linear) may generate a copying critical pair, i.e. copying CS-clauses. Consequently, even if the starting program is not copying, it may become copying during the completion algorithm. Example 5 illustrates this problem.

Example 5. $Prog = \{P(f(x)) \leftarrow Q(x).\quad Q(a) \leftarrow .\}$ and $R = \{f(x) \rightarrow g(x, x)\}$. There is one critical pair $P(g(x, x)) \leftarrow Q(x)$, which is copying. Thus $\mathsf{comp}_R(Prog)$ is copying.

Actually, the hypotheses of Definition 4 have to be stronger i.e. the TRS must be linear in order to prevent the introduction of copying clauses by the completion process, and the starting program must be non-copying. On the other hand, the *preserving* assumption is not needed anymore.

Definition 5 (New comp). *Let arity-limit and predicate-limit be positive integers. Let R be a linear rewrite system, and $Prog$ be a finite, normalized and non-copying CS-program. The completion process is defined by:*
Function $\mathsf{comp}_R(Prog)$
  $Prog = \mathsf{removeCycles}(Prog)$
  while *there exists a non-convergent critical pair $H \leftarrow B$ in $Prog$* ***do***
    $Prog = \mathsf{removeCycles}(Prog \cup \mathsf{norm}_{Prog}(H \leftarrow B))$
  end while
  return $Prog$

Thus, a new version of Theorem 2 is given below:

Theorem 4. *Let R be a left-linear*[7] *rewrite system and $Prog$ be a normalized non-copying CS-program.*
If all critical pairs are convergent, then $Mod(Prog)$ is closed under rewriting by R, i.e. $(A \in Mod(Prog) \wedge A \rightarrow_R^* A') \implies A' \in Mod(Prog)$.

Proof. Let $A \in Mod(Prog)$ s.t. $A \rightarrow_{l \rightarrow r} A'$. Then $A|_i = C[\sigma(l)]$ for some $i \in \mathbb{N}$ and $A' = A[i \leftarrow C[\sigma(r)]$.
Since resolution is complete, $A \leadsto^* \emptyset$. Since $Prog$ is normalized, resolution consumes symbols in C one by one, thus $G_0 = A \leadsto^* G_k \leadsto^* \emptyset$ and there exists

[7] From a theoretical point of view, left-linearity is sufficient when every critical pair is convergent. However, to make every critical pair convergent by completion, full linearity is necessary (see Theorem 5).

an atom $A'' = P(t_1, \ldots, t_n)$ in G_k and j s.t. $t_j = \sigma(l)$ and the top symbol of t_j is consumed (or t_j disappears) during the step $G_k \leadsto G_{k+1}$. Since $Prog$ is non-copying, $t_j = \sigma(l)$ admits only one antecedent in A.

Consider new variables $x_1, \ldots, x_n$ s.t. $\{x_1, \ldots, x_n\} \cap Var(l) = \emptyset$, and let us define the substitution σ' by $\forall i$, $\sigma'(x_i) = t_i$ and $\forall x \in Var(l)$, $\sigma'(x) = \sigma(x)$. Then $\sigma'(P(x_1, \ldots, x_{j-1}, l, x_{j+1}, \ldots, x_n)) = A''$, and according to resolution (or narrowing) properties $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_\theta \emptyset$ and $\theta \leq \sigma'$.

This derivation can be decomposed into : $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_{\theta_1} G' \leadsto_{\theta_2} G \leadsto^*_{\theta_3} \emptyset$ where $\theta = \theta_3 \circ \theta_2 \circ \theta_1$, and s.t. G' is not flat and G is flat[8]. $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_{\theta_1} G' \leadsto_{\theta_2} G$ can be commuted into $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_{\gamma_1} B' \leadsto_{\gamma_2} B \leadsto^*_{\gamma_3} G$ s.t. B is flat, B' is not flat, and within $P(x_1, \ldots, l, \ldots, x_n) \leadsto^*_{\gamma_1} B' \leadsto_{\gamma_2} B$ resolution is applied only on non-flat atoms, and we have $\gamma_3 \circ \gamma_2 \circ \gamma_1 = \theta_2 \circ \theta_1$. Then $\gamma_2 \circ \gamma_1(P(x_1, \ldots, r, \ldots, x_n)) \leftarrow B$ is a critical pair. By hypothesis, it is convergent, then $\gamma_2 \circ \gamma_1(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* B$. Note that $\gamma_3(B) \rightarrow^* G$ and recall that $\theta_3 \circ \gamma_3 \circ \gamma_2 \circ \gamma_1 = \theta_3 \circ \theta_2 \circ \theta_1 = \theta$. Then $\theta(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* \theta_3(G) \rightarrow^* \emptyset$, and since $\theta \leq \sigma'$ we get $P(t_1, \ldots, \sigma(r), \ldots, t_n) = \sigma'(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* \emptyset$. Recall that $\sigma(l)$ has only one antecedent in A. Therefore $A' \leadsto^* G_k[A'' \leftarrow P(t_1, \ldots, \sigma(r), \ldots, t_n)] \leadsto^* \emptyset$, hence $A' \in Mod(Prog)$.

By trivial induction, the proof can be extended to the case of several rewrite steps.

Consequently, one can update Theorem 3 as follows:

Theorem 5. *Let R be a linear rewrite system and $Prog$ be a normalized non-copying CS-program. Function* comp *always terminates, and all critical pairs are convergent in* $\mathsf{comp}_R(Prog)$*. Moreover,* $R^*(Mod(Prog)) \subseteq Mod(\mathsf{comp}_R(Prog))$*.*

References

1. Y. Boichut, J. Chabin, and P. Réty. Over-approximating descendants by synchronized tree languages. In *RTA*, volume 21 of *LIPIcs*, pages 128–142, 2013.
2. Y. Boichut and P.-C. Héam. A Theoretical Limit for Safety Verification Techniques with Regular Fix-point Computations. *Information Processing Letters*, 108(1):1–2, 2008.
3. V. Gouranton, P. Réty, and H. Seidl. Synchronized Tree Languages Revisited and New Applications. In *FoSSaCS*, volume 2030 of *LNCS*, pages 214–229. Springer, 2001.
4. S. Limet and P. Réty. E-Unification by Means of Tree Tuple Synchronized Grammars. *Discrete Mathematics and Theoritical Computer Science*, 1(1):69–98, 1997.
5. S. Limet and G. Salzer. Proving Properties of Term Rewrite Systems via Logic Programs. In *RTA*, volume 3091 of *LNCS*, pages 170–184. Springer, 2004.
6. Sébastien Limet and Gernot Salzer. Tree Tuple Languages from the Logic Programming Point of View. *Journal of Automated Reasoning*, 37(4):323–349, 2006.

[8] Since $\emptyset$ is flat, a flat goal can always be reached, i.e. in some cases $G = \emptyset$.