4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Towards more Precise Rewriting Approximations (full version)

Yohan Boichut, Jacques Chabin and Pierre Réty
LIFO, Université d'Orléans

Rapport n^{o} **RR-2014-02**

Towards more Precise Rewriting Approximations

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

Abstract. To check a system, some verification techniques consider a set of terms I that represents the initial configurations of the system, and a rewrite system R that represents the system behavior. To check that no undesirable configuration is reached, they compute an over-approximation of the set of descendants (successors) issued from I by R, expressed by a tree language. Their success highly depends on the quality of the approximation. Some techniques have been presented using regular tree languages, and more recently using non-regular languages to get better approximations: using context-free tree languages [15] on the one hand, using synchronized tree languages [2] on the other hand. In this paper, we merge these two approaches to get even better approximations: we compute an over-approximation of the descendants, using synchronized-context-free tree languages expressed by logic programs. We give several examples for which our procedure computes the descendants in an exact way, whereas the former techniques compute a strict over-approximation.

Keywords: term rewriting, tree languages, logic programming, reachability.

1 Introduction

To check systems like cryptographic protocols or Java programs, some verification techniques consider a set of terms I that represents the initial configurations of the system, and a rewrite system R that represents the system behavior [1, 12, 13]. To check that no undesirable configuration is reached, they compute an over-approximation of the set of descendants[1] (successors) issued from I by R, expressed by a tree language. Let $R^*(I)$ denote the set of descendants of I, and consider a set Bad of *undesirable* terms. Thus, if a term of Bad is reached from I, i.e. $R^*(I) \cap Bad \neq \emptyset$, it means that the protocol or the program is flawed. In general, it is not possible to compute $R^*(I)$ exactly. Instead, one computes an over-approximation App of $R^*(I)$ (i.e. $App \supseteq R^*(I)$), and checks that $App \cap Bad = \emptyset$, which ensures that the protocol or the program is correct.

However, I, Bad and App have often been considered as regular tree languages, recognized by finite tree automata. In the general case, $R^*(I)$ is not

[1] I.e. terms obtained by applying arbitrarily many rewrite steps on the terms of I.

regular, even if I is. Moreover, the expressiveness of regular languages is poor. Then the over-approximation App may not be precise enough, and we may have $App \cap Bad \neq \emptyset$ whereas $R^*(I) \cap Bad = \emptyset$. In other words, the protocol is correct, but we cannot prove it. Some work has proposed CEGAR-techniques (Counter-Example Guided Approximation Refinement) to conclude as often as possible [1, 3, 5]. However, in some cases, no regular over-approximation works [4].

To overcome this theoretical limit, the idea is to use more expressive languages to express the over-approximation, i.e. non-regular ones. However, to be able to check that $App \cap Bad = \emptyset$, we need a class of languages closed under intersection and whose emptiness is decidable. Actually, if we assume that Bad is regular, closure under intersection with a regular language is enough. The class of context-free tree languages has these properties, and an approximation technique using context-free tree languages has been proposed in [15]. On the other hand, the class of synchronized tree languages [16] also has these properties, and an approximation technique using synchronized tree languages has been proposed in [2]. Both classes include regular languages, but they are incomparable. Context-free tree languages cannot express dependencies between different branches, except in some cases, whereas synchronized tree languages cannot express vertical dependencies.

We want to use a more powerful class of languages that can express the two kinds of dependencies together: the class of synchronized-context-free tree-(tuple) languages [20, 21], which has the same properties as context-free languages and as synchronized languages, i.e. closure under union, closure under intersection with a regular language, decidability of membership and emptiness.

In this paper, we propose a procedure that always terminates and that computes an over-approximation of the descendants obtained by a linear rewrite system, using synchronized-context-free tree-(tuple) languages expressed by logic programs. Compared to our previous work [2], we introduce "input arguments" in predicates, which is a major technical change that highly improves the quality of the approximation, and that requires new results and new proofs. This work is a first step towards a verification technique offering more than regular approximations. Some on-going work is discussed in Section 5 in order to make this technique be an accepted verification technique.

The paper is organized as follows: classical notations and notions manipulated throughout the paper are introduced in Section 2. Our main contribution, i.e. computing approximations, is explained in Section 3. Finally, in Section 4 our technique is applied on examples, in particular when $R^*(I)$ can be expressed in an exact way neither by a context-free language, nor by a synchronized language. For lack of space, all proofs are in the appendix.

Related Work: The class of tree-tuples whose overlapping coding is recognized by a tree automaton on the product alphabet [6] (called "regular tree relations" by some authors), is strictly included in the class of rational tree relations [18].

The latter is equivalent to the class of non-copying[2] synchronized languages [19], which is strictly included in the class of synchronized languages.

Context-free tree languages (i.e. without assuming a particular strategy for grammar derivations) [22] are equivalent to OI (outside-in strategy) context-free tree languages, but are incomparable with IO (inside-out strategy) context-free tree languages [10, 11]. The IO class (and not the OI one) is strictly included in the class of synchronized-context-free tree languages. The latter is equivalent to the "term languages of hyperedge replacement grammars", which are equivalent to the tree languages definable by attribute grammars [8, 9]. However, we prefer to use the synchronized-context-free tree languages, which use the well known formalism of pure logic programming, for its implementation ease.

Much other work computes the descendants in an exact way using regular tree languages (in particular the recent paper [7]). In general the set of descendants is not regular even if the initial set is. Consequently strong restrictions over the rewrite system are needed to get regular descendants, which are not suitable in the framework of protocol or program verification.

2 Preliminaries

Consider a *finite ranked alphabet* Σ and a set of variables Var. 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 σ and σ' two substitutions, $\sigma \circ \sigma'$ 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'. $PosVar(t) = \{p \in Pos(t) \mid t(p) \in Var\}$, $PosNonVar(t) = \{p \in Pos(t) \mid t(p) \notin Var\}$. Note that if $p \in PosNonVar(t)$, $t|_p = f(t_1, \ldots, t_n)$, and $i \in \{1, \ldots, n\}$, then $p.i$ is the position of t_i in t. For $p, p' \in Pos(t)$, $p < p'$ means that p occurs in t strictly above p'. Let t, t' be terms, t is *more general than* t' (denoted $t \leq t'$) if there exists a substitution ρ s.t. $\rho(t) = t'$. Let σ, σ' be substitutions, σ is *more general than* σ' (denoted $\sigma \leq \sigma'$) if there exists a substitution ρ s.t. $\rho \circ \sigma = \sigma'$.

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] Clause heads are assumed to be linear.

In the following, we consider the framework of *pure logic programming*, and the class of synchronized-context-free tree-tuple[3] languages [20, 21], which is presented as an extension of the class of synchronized tree-tuple languages defined by CS-clauses [16, 17]. 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. *The tuple of terms $(t_1, \ldots, t_n)$ is* flat *if $t_1, \ldots, t_n$ are variables. The sequence of atoms B is* flat *if for each atom $P(t_1, \ldots, t_n)$ of B, $t_1, \ldots, t_n$ 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 Horn clause $P(t_1, \ldots, t_n) \leftarrow B$ is:

- empty *if $P(t_1, \ldots, t_n)$ is flat, i.e. $\forall i \in \{1, \ldots, n\}$, 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. A program is* normalized *if all its clauses are normalized.*

Example 1. Let x, y, z be variables. The sequence of atoms $P_1(x, y), P_2(z)$ is flat, whereas $P_1(x, f(y)), P_2(z)$ is not flat. The clause $P(x, y) \leftarrow Q(x, y)$ is empty and normalized (x, y are variables). The clause $P(f(x), y) \leftarrow Q(x, y)$ is normalized whereas $P(f(f(x)), y) \leftarrow Q(x, y)$ is not.

Definition 2. A *logic program with modes* is a logic program such that a mode-tuple $\vec{m} \in \{I, O\}^n$ 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. Let B be a sequence of atoms (possibly containing only one atom). $In(B)$ is the input part of B, i.e. the tuple composed of the input arguments of B. $ArIn(B)$ is the arity of $In(B)$. $Var^{in}(B)$ is the set of variables that appear in $In(B)$. $Out(B)$, $ArOut(B)$, and $Var^{out}(B)$ are defined in a similar way. We also define $Var(B) = Var^{in}(B) \cup Var^{out}(B)$.

Example 2. Let $B = P(\widehat{t_1}, \widehat{t_2}, t_3), Q(\widehat{t_4}, t_5, t_6)$. Then, $Out(B) = (t_1, t_2, t_4)$ and $In(B) = (t_3, t_5, t_6)$.

Definition 3. Let $B = A_1, \ldots, A_n$ be a sequence of atoms. We say that $A_j \succ A_k$ (possibly $j = k$) if $\exists y \in Var^{in}(A_j) \cap Var^{out}(A_k)$. In other words an input of A_j depends on an output of A_k. We say that B has a *loop* if $A_j \succ^+ A_j$ for some A_j ($\succ^+$ is the transitive closure of $\succ$).

Example 3. $Q(\widehat{x}, s(y))$, $R(\widehat{y}, s(x))$ (where x, y are variables) has a loop because $Q(\widehat{x}, s(y)) \succ R(\widehat{y}, s(x)) \succ Q(\widehat{x}, s(y))$.

[3] For simplicity, "tree-tuple" is sometimes omitted.

Definition 4. A *Synchronized-Context-Free (S-CF) program* $Prog$ is a logic program with modes, whose clauses $H \leftarrow B$ satisfy:

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

A clause of an S-CF program is called *S-CF clause*.

Example 4. $Prog = \{P(\widehat{x}, y) \leftarrow P(\widehat{s(x)}, y)\}$ is not an S-CF program because $In(H).Out(B) = (y, s(x))$ is not a tuple of variables. $Prog' = \{P'(\widehat{s(x)}, y) \leftarrow P'(\widehat{x}, s(y))\}$ is an S-CF program because $In(H).Out(B) = (y, x)$ is a linear tuple of variables, and there is no loop in the clause body.

Definition 5. Let $Prog$ be an S-CF program. Given a predicate symbol P without input arguments, the tree-(tuple) language generated by P is $L_{Prog}(P) = \{\vec{t} \in (T_\Sigma)^{ArOut(P)} \mid P(\vec{t}) \in Mod(Prog)\}$, where T_Σ is the set of ground terms over the signature Σ and $Mod(Prog)$ is the least Herbrand model of $Prog$. $L_{Prog}(P)$ is called *Synchronized-Context-Free language (S-CF language)*.

Example 5. Let us consider the S-CF program without input arguments $Prog = \{P_1(\widehat{g(x,y)}) \leftarrow P_2(\widehat{x}, \widehat{y}).\ P_2(\widehat{a}, \widehat{a}) \leftarrow .\ P_2(\widehat{c(x,y)}, \widehat{c(x',y')}) \leftarrow P_2(\widehat{x}, \widehat{y'}), P_2(\widehat{y}, \widehat{x'}).\}$. The language generated by P_1 is $L_{Prog}(P_1) = \{g(t, t_{sym}) \mid t \in T_{\{c\backslash 2, a\backslash 0\}}\}$, where t_{sym} is the symmetric tree of t (for instance $c(c(a,a),\ a)$ is the symmetric of $c(a, c(a,a))$). This language is synchronized, but it is not context-free.

Example 6. $Prog = \{S(\widehat{c(x,y)}) \leftarrow P(\widehat{x}, \widehat{y}, a, b).$
$P(\widehat{f(x)}, \widehat{g(y)}, x', y') \leftarrow P(\widehat{x}, \widehat{y}, h(x'), i(y')).\ P(\widehat{x}, \widehat{y}, x, y) \leftarrow\}$ is an S-CF program. The language generated by S is $L_{Prog}(S) = \{c(f^n(h^n(a)), g^n(i^n(b))) \mid n \in \mathbb{N}\}$, which is not synchronized (there are vertical dependencies) nor context-free.

Definition 6. The S-CF clause $H \leftarrow B$ is *non-copying* if the tuple $Out(H).In(B)$ is linear. A program is *non-copying* if all its clauses are non-copying.

Example 7. The clause $P(\widehat{d(x,x)}, y) \leftarrow Q(\widehat{x}, p(y))$ is copying whereas $P(\widehat{c(x)}, y) \leftarrow Q(\widehat{x}, p(y))$ is non-copying.

Remark 1. An S-CF program without input arguments is actually a CS-program (composed of CS-clauses) [16], which generates a synchronized language[4]. A non-copying CS-program such that every predicate symbol has only one argument generates a regular tree language[5]. Conversely, every regular tree language can be generated by a non-copying CS-program.

Given an S-CF program, we focus on two kinds of derivations.

Definition 7. *Given an S-CF program $Prog$ and a sequence of atoms G,*

[4] Initially, synchronized languages were presented using constraint systems (sorts of grammars) [14], and later using logic programs. CS stands for “Constraint System”.

[5] In this case, the S-CF program can easily be transformed into a finite tree automaton.

- *G derives into G' by a* resolution *step if there exists a clause*[6] *$H \leftarrow B$ 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 $G \leadsto_\sigma G'$.*
  We consider the transitive closure $\leadsto^+$ and the reflexive-transitive closure $\leadsto^*$ of $\leadsto$. If $G_1 \leadsto_{\sigma_1} G_2$ and $G_2 \leadsto_{\sigma_2} G_3$, we write $G_1 \leadsto^*_{\sigma_2 \circ \sigma_1} G_3$.
- *G* rewrites *into G' (possibly in several steps) if $G \leadsto^*_\sigma G'$ s.t. σ does not instantiate the variables of G. It is written $G \to^*_\sigma G'$.*

Example 8. Let $Prog = \{P(\widehat{x_1}, \widehat{g(x_2)}) \leftarrow P'(\widehat{x_1}, \widehat{x_2}).\quad P(\widehat{f(x_1)}, \widehat{x_2}) \leftarrow P''(\widehat{x_1}, \widehat{x_2}).\}$, 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 $P(f(x), y)) \to_{\sigma_2} P''(x, y)$ with $\sigma_2 = [x_1/x, x_2/y]$.

In the remainder of the paper, given an S-CF program $Prog$ and two sequences of atoms G_1 and G_2, $G_1 \leadsto^*_{Prog} G_2$ (resp. $G_1 \to^*_{Prog} G_2$) also denotes that G_2 can be derived (resp. rewritten) from G_1 using clauses of $Prog$. Note that for any atom A, if $A \to B$ then $A \leadsto B$. On the other hand, $A \leadsto_\sigma B$ implies $\sigma(A) \to B$. Consequently, if A is ground, $A \leadsto B$ implies $A \to B$.

It is well known that resolution is complete.

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

3 Computing Descendants

To make the understanding easier, we first give the completion algorithm in Definition 8. Given a normalized S-CF program $Prog$ and a linear rewrite system R, we propose an algorithm to compute a normalized S-CF program $Prog'$ such that $R^*(Mod(Prog)) \subseteq Mod(Prog')$, and consequently $R^*(L_{Prog}(P)) \subseteq L_{Prog'}(P)$ for each predicate symbol P. Some notions will be explained later.

Definition 8 (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 S-CF program strongly coherent with R. 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 $Prog$

Let us explain this algorithm.

The notion of critical pair is at the heart of the technique. Given an S-CF program $Prog$, a predicate symbol P and a rewrite rule $l \to r$, a critical pair, explained in details in Section 3.1, is a way to detect a possible rewriting by $l \to r$ for a term t in $L(P)$. A convergent critical pair means that the rewrite step is

[6] We assume that the clause and G have distinct variables.

already handled i.e. if $t \rightarrow_{l \rightarrow r} s$ then $s \in L(P)$. Consequently, the language of a normalized CS-program involving only convergent critical pairs is closed by rewriting.

To summarize, a non-convergent critical pair gives rise to an S-CF clause. Adding the resulting S-CF clause to the current S-CF program makes the critical pair convergent. But, let us emphasize on the main problems arising from Definition 8, i.e. the computation may not terminate and the resulting S-CF clause may not be normalized. Concerning the non-termination, there are mainly two reasons. Given a normalized S-CF program $Prog$, 1) the number of critical pairs may be infinite and 2) even if the number of critical pairs is finite, adding the critical pairs to $Prog$ may create new non-convergent critical pairs, and so on.

Actually, as in [2], there is a function called $\mathsf{removeCycles}$ whose goal is to get finitely many critical pairs from a given finite S-CF program. For lack of space, many details on this function are given in Appendix E. Basically, given an S-CF program $Prog$ having infinitely many critical pairs, $\mathsf{removeCycles}(Prog)$ is another S-CF program that has finitely many critical pairs, and such that for any predicate symbol P, $L_{Prog}(P) \subseteq L_{\mathsf{removeCycles}(Prog)}(P)$. The normalization process presented in Section 3.2 not only preserves the normalized nature of the computed S-CF programs but also allows us to control the creation of new non-convergent critical pairs. Finally, in Section 3.3, our main contribution, i.e. the computation of an over-approximating S-CF program, is fully described.

3.1 Critical pairs

The notion of critical pair is the heart of our technique. Indeed, it allows us to add S-CF clauses into the current S-CF program in order to cover rewriting steps.

Definition 9. *Let $Prog$ be a non-copying S-CF program and $l \rightarrow r$ be a left-linear rewrite rule. Let $x_1, \ldots, x_n$ be distinct variables such that $\{x_1, \ldots, x_n\} \cap Var(l) = \emptyset$. If there are P and k s.t. the k^{th} argument of P is an output, and $P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n) \leadsto_\theta^+ G$ where*[7]

1. *resolution steps are applied only on atoms whose output is not flat,*
2. *$Out(G)$ is flat and*
3. *the clause $P(t_1, \ldots, t_n) \leftarrow B$ used in the first step of this derivation satisfies t_k is not a variable*[8]

then the clause $\theta(P(x_1, \ldots, x_{k-1}, r, x_{k+1}, \ldots, x_n)) \leftarrow G$ is called critical pair. *Moreover, if θ does not instantiate the variables of $In(P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n))$ then the critical pair is said* strict.

[7] Here, we do not use a hat to indicate output arguments because they may occur anywhere depending on P.

[8] In other words, the overlap of l on the clause head $P(t_1, \ldots, t_n)$ is done at a non-variable position.

Example 9. Let $Prog$ be the S-CF program defined by:
$Prog = \{P(\widehat{x}) \leftarrow Q(\widehat{x}, a).\ \ Q(\widehat{f(x)}, y) \leftarrow Q(\widehat{x}, g(y)).\ \ Q(\widehat{x}, x) \leftarrow .\}$ and consider the rewrite system: $R = \{f(x) \to x\}$. Note that $L(P) = \{f^n(g^n(a)) \mid n \in \mathbb{N}\}$.
We have $Q(\widehat{f(x)}, y) \rightsquigarrow_{Id} Q(\widehat{x}, g(y))$ where Id denotes the substitution that leaves every variable unchanged. Since $Out(Q(\widehat{x}, g(y)))$ is flat, this generates the strict critical pair $Q(\widehat{x}, y) \leftarrow Q(\widehat{x}, g(y))$.

Lemma 1. *A strict critical pair is an S-CF clause. In addition, if $l \to r$ is right-linear, a strict critical pair is a non-copying S-CF clause.*

Definition 10. *A critical pair $H \leftarrow B$ is said* convergent *if $H \to^*_{Prog} B$.*

The critical pair of Example 9 is not convergent.

Let us recall that the completion procedure is based on adding the non-convergent critical pairs into the program. In order to preserve the nature of the S-CF program, the computed non-convergent critical pairs are expected to be strict. So we define a sufficient condition on R and $Prog$ called *strong coherence*.

Definition 11. Let R be a rewrite system. We consider the smallest set of *consuming* symbols, recursively defined by: $f \in \Sigma$ is *consuming* if there exists a rewrite rule $f(t_1, \ldots, t_n) \to r$ in R s.t. some t_i is not a variable, or r contains at least one consuming symbol.
The S-CF program $Prog$ is *strongly coherent* with R if 1) for all $l \to r \in R$, the top-symbol of l does not occur in input arguments of $Prog$ and 2) no consuming symbol occurs in clause-heads having input arguments.

In $R = \{f(x) \to g(x),\ g(s(x)) \to h(x)\}$, g is consuming and so is f. Thus $Prog = \{P(\widehat{f(x)}, x) \leftarrow .\}$ is not strongly coherent with R. Note that a CS-program (no input arguments) is strongly coherent with any rewrite system.

Lemma 2. *If $Prog$ is a normalized S-CF program strongly coherent with R, then every critical pair is strict.*

So, we come to our main result that ensures to get the rewriting closure when every computable critical pair is convergent.

Theorem 2. *Let R be a linear rewrite system, and $Prog$ be a non-copying normalized S-CF program strongly coherent with R. If all strict critical pairs are convergent, then for every predicate symbol P without input arguments, $L(P)$ is closed under rewriting by R, i.e. $(\vec{t} \in L(P) \wedge \vec{t} \to^*_R \vec{t'}) \implies \vec{t'} \in L(P)$.*

3.2 Normalizing critical pairs – norm_{Prog}

If a critical pair is not convergent, we add it into $Prog$, and the critical pair becomes convergent. However, in the general case, a critical pair is not normalized, whereas all clauses in $Prog$ should be normalized. In the case of CS-clauses (i.e. without input arguments), a procedure that transforms a non-normalized clause into normalized ones has been presented [2]. For example, $P(\widehat{f(g(x))}, \widehat{b}) \leftarrow Q(\widehat{x})$ is normalized into $\{P(\widehat{f(x_1)}, \widehat{b}) \leftarrow P_1(\widehat{x_1}).\ \ P_1(\widehat{g(x_1)}) \leftarrow Q(\widehat{x_1}).\}$ (P_1 is a new predicate symbol). Since only output arguments should be normalized, this procedure still works even if there are also input arguments. As new predicate symbols are introduced, possibly with bigger arities, the procedure may not terminate. To make it terminate in every case, two positive integers are used: *predicate-limit* and *arity-limit*. If the number of predicate symbols having the same arity as P_1 (including P_1) exceeds *predicate-limit*, an existing predicate symbol (for example Q) must be used instead of the new predicate P_1. This may enlarge $Mod(Prog)$ in general and may lead to a strict over-approximation. If the arity of P_1 exceeds *arity-limit*, P_1 must be replaced in the clause body by several predicate symbols[9] whose arities are less than or equal to *arity-limit*. This may also enlarge $Mod(Prog)$. See [2] for more details.

In other words $\mathsf{norm}_{Prog}(H \leftarrow B)$ builds a set of normalized S-CF clauses N such that $Mod(Prog \cup \{H \leftarrow B\}) \subseteq Mod(Prog \cup N)$.

However, when starting from a CS-program (i.e. without input arguments), it could be interesting to normalize by introducing input arguments, in order to profit from the bigger expressiveness of S-CF programs, and consequently to get a better approximation of the set of descendants, or even an exact computation, like in Examples 10 and 11 presented in Section 4. The quality of the approximation depends on the way the normalization is achieved. Some heuristics concerning the choice of functional symbols occurring as inputs will be developed in further work. Anyway, these heuristics will have to preserve the strong coherence property.

3.3 Completion

At the very beginning of Section 3, we have presented in Definition 8 the completion algorithm i.e. comp_R. In Sections 3.1 and 3.2, we have described how to detect non-convergent critical pairs and how to convert them into normalized clauses using norm_{Prog}.

Theorem 3 illustrates that our technique leads to a finite S-CF program whose language over-approximates the descendants obtained by a linear rewrite system R.

Theorem 3. *Function* comp *always terminates, and all critical pairs are convergent in* $\mathsf{comp}_R(Prog)$. *Moreover, for each predicate symbol P without input arguments,* $R^*(L_{Prog}(P)) \subseteq L_{\mathsf{comp}_R(Prog)}(P)$.

[9] For instance, if P_1 is binary and *arity-limit* $= 1$, then $P_1(t_1, t_2)$ should be replaced by the sequence of atoms $P_2(t_1), P_3(t_2)$. Note that the dependency between t_1 and t_2 is lost, which may enlarge $Mod(Prog)$. Symbols P_2 and P_3 are new if it is compatible with *predicate-limit*. Otherwise former predicate symbols should be used instead of P_2 and P_3.

4 Examples

In this section, our technique is applied on several examples. I is the initial set of terms and R is the rewrite system. Initially, we define an S-CF program $Prog$ that generates I and that satisfies the assumptions of Definition 8. For lack of space, the examples should be as short as possible. To make the procedure terminate shortly, we suppose that *predicate-limit*=1, which means that for all i, there is at most one predicate symbol having i arguments, except for $i = 1$ we allow two predicate symbols having one argument.

When the following example is dealt with synchronized languages, i.e. with CS-programs [2, Example 42], we get a strict over-approximation of the descendants. Now, thanks to the bigger expressive power of S-CF programs, we compute the descendants in an exact way.

Example 10. Let $I = \{f(a,a)\}$ and $R = \{f(x,y) \to u(f(v(x),w(y)))\}$. Intuitively, the exact set of descendants is $R^*(I) = \{u^n(f(v^n(a),w^n(a))) \mid n \in \mathbb{N}\}$ where u^n means that u occurs n times. We define $Prog = \{P_f(\widehat{f(x,y)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}).,\ P_a(\widehat{a}) \leftarrow .\}$. Note that $L_{Prog}(P_f) = I$. The run of the completion is given in Fig 1. The reader can refer to Appendix G for a detailed explanation. In Fig 1, the left-most column reports the detected non-convergent critical pairs and the right-most column describes how they are normalized. Note that for the resulting program $Prog$, i.e. clauses appearing in the right-most column, $L_{Prog}(P_f) = R^*(I)$ indeed.

Detected non-convergent critical pairs	New clauses obtained by norm_{Prog}
	Starting S-CF program $P_f(\widehat{f(x,y)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}).$ $P_a(\widehat{a}) \leftarrow .$
$P_f(\widehat{u(f(v(x),w(y)))}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}).$	$P_f(\widehat{z}) \leftarrow P_1(\widehat{z},x,y), P_a(\widehat{x}), P_a(\widehat{y}).$ $P_1(\widehat{u(z)},x,y) \leftarrow P_1(\widehat{z},v(x),w(y)).$ $P_1(\widehat{f(x,y)},x,y) \leftarrow .$
$\emptyset$	

Fig. 1. Run of comp_R on Example 10

The previous example could probably be dealt in an exact way using the technique of [15] as well, since $R^*(I)$ is a context-free language. It is not the case for the following example, whose language of descendants $R^*(I)$ is not context-free (and not synchronized). It can be handled by S-CF programs in an exact way thanks to their bigger expressive power.

Example 11. Let $I = \{d_1(a,a,a)\}$ and

$$R = \left\{ \begin{array}{ll} d_1(x,y,z) \xrightarrow{1} d_1(h(x),i(y),s(z)), & d_1(x,y,z) \xrightarrow{2} d_2(x,y,z) \\ d_2(x,y,s(z)) \xrightarrow{3} d_2(f(x),g(y),z), & d_2(x,y,a) \xrightarrow{4} c(x,y) \end{array} \right\}$$

$R^*(I)$ is composed of all terms appearing in the following derivation:

$$d_1(a,a,a) \xrightarrow{1}{}^{n} d_1(h^n(a),i^n(a),s^n(a)) \xrightarrow{2} d_2(h^n(a),i^n(a),s^n(a))$$
$$\xrightarrow{3}{}^{k} d_2(f^k(h^n(a)),g^k(i^n(a)),s^{n-k}(a)) \xrightarrow{4} c(f^n(h^n(a)),g^n(i^n(a)))\,.$$

Note that the last rewrite step by rule 4 is possible only when $k = n$. The run of the completion on this example is given in Fig 2. Black arrows means that the non-convergent critical pair is directly added to *Prog* since it is already normalized. The reader can find a full explanation of this example in Appendix H.

Detected non-convergent critical pairs	New clauses obtained by norm_{Prog}
	Starting S-CF program $P_d(d_1(\widehat{x,y},z)) \leftarrow P_a(\widehat{x}),P_a(\widehat{y}),P_a(\widehat{z})$. $P_a(\widehat{a}) \leftarrow$.
$P_d(d_1(h(x)\widehat{,i(y)},s(z))) \leftarrow P_a(\widehat{x}),P_a(\widehat{y}),P_a(\widehat{z})$	$P_d(d_1(\widehat{x,y},z)) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$. $P_1(\widehat{h(x)},\widehat{i(y)},\widehat{s(z)}) \leftarrow P_a(\widehat{x}),P_a(\widehat{y}),P_a(\widehat{z})$.
$P_d(d_2(\widehat{x,y},z)) \leftarrow P_a(\widehat{x}),P_a(\widehat{y}),P_a(\widehat{z})$.	$\longrightarrow$
$P_d(d_1(h(x)\widehat{,i(y)},s(z))) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$	$P_1(\widehat{h(x)},\widehat{i(y)},\ \widehat{s(z)}) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$.
$P_d(d_2(\widehat{x,y},z)) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$.	$\longrightarrow$
$P_d(c(\widehat{x,y})) \leftarrow P_a(\widehat{x}),P_a(\widehat{y})$.	$\longrightarrow$
$P_d(d_2(f(h(x)\widehat{),g}(i(y)),z)) \leftarrow P_a(\widehat{x}),P_a(\widehat{y}),P_a(\widehat{z})$	$P_d(d_2(\widehat{x,y},z)) \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z},x',y',z'),P_a(\widehat{x'}),P_a(\widehat{y'}),P_a(\widehat{z'})$. $P_2(\widehat{f(x)},\widehat{g(y)},\widehat{z},x',y',z') \leftarrow P_2(\ \widehat{x},\widehat{y},\widehat{z},h(x'),i(y'),z')$ $P_2(\widehat{x},\widehat{y},\widehat{z},x,y,z) \leftarrow$.
A cycle is detected – $\mathsf{removeCycles}$ replaces the blue clause by the red one.	$P_2(\widehat{f(x)},\widehat{g(y)},\widehat{z},\ x',y',z') \leftarrow\ P_2(\widehat{x},\widehat{y},\widehat{z_1},h(x'),i(y'),z'_1),$ $P_2(\widehat{x_1},\widehat{y_1},\widehat{z},\ h(x'_1),i(y'_1),\ z')$
$P_d(d_2(f(h(x)\widehat{),g}(i(y)),z)) \leftarrow P_1(\widehat{x},\widehat{y},\widehat{z})$	$P_d(d_2(\widehat{x,y},z)) \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z},x',y',z'),\ P_1(\widehat{x'},\widehat{y'},\ \widehat{z'})$.
$P_d(c(f(\widehat{x),g}(y))) \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z},h(x'),i(y'),z'),$ $P_a(\widehat{x'}),P_a(\widehat{y'})$.	$P_3(\widehat{f(x)},\widehat{g(y)}) \leftarrow P_2(\widehat{x},\widehat{y},\widehat{z},\ h(x'),i(y'),z'),\ \ P_a(\widehat{x'}),P_a(\widehat{y'})$. $P_d(c(\widehat{x,y})) \leftarrow P_3(\widehat{x},\widehat{y})$.

Fig. 2. Run of comp_R on Example 11

Note that the subset of descendants $d_2(f^k(h^n(a)),g^k(i^n(a)),s^{n-k}(a))$ can be seen (with $p = n-k$) as $d_2(f^k(h^{k+p}(a)),g^k(i^{k+p}(a\ )),s^p(a))$. Let *Prog'* be the S-CF program composed of all the clauses except the blue one occurring in the right-most column in Fig 2. Thus, the reader can check by himself that $L_{Prog'}(P_d)$ is exactly $R^*(I)$.

5 Further Work

Computing approximations more precise than regular approximations is a first step towards a verification technique. However, there are at least two steps before claiming this technique as a verification technique: 1) automatically handling the choices done during the normalization process and 2) extending our technique to any rewrite system. The quality of the approximation is closely related to those choices. On one hand, it depends on the choice of the predicate symbol to be reused when *predicate-limit* is reached. On the other hand, the choice of generating function-symbols as output or as input is also crucial. According to the verification context, some automated heuristics will have to be designed in order to obtain well-customized approximations.

On-going work tends to show that the linear restriction concerning the rewrite system can be tackled. A non right-linear rewrite system makes the computed S-CF program copying. Consequently, Theorem 2 does not hold anymore. To get rid of the right-linearity restriction, we are studying the transformation of a copying S-CF clause into non-copying ones that will generate an over-approximation. On the other hand, to get rid of the left-linearity restriction, we are studying a technique based on the transformation of any Horn clause into CS-clauses [16]. However, the method of [16] does not always terminate. We want to ensure termination thanks to an additional over-approximation.

References

1. Y. Boichut, B. Boyer, Th. Genet, and A. Legay. Equational Abstraction Refinement for Certified Tree Regular Model Checking. In *ICFEM*, volume 7635 of *LNCS*, pages 299–315. Springer, 2012.
2. Y. Boichut, J. Chabin, and P. Réty. Over-approximating descendants by synchronized tree languages. In *RTA*, volume 21 of *LIPIcs*, pages 128–142, 2013.
3. Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Finer is Better: Abstraction Refinement for Rewriting Approximations. In *RTA*, volume 5117 of *LNCS*, pages 48–62. Springer, 2008.
4. Y. Boichut and P.-C. Héam. A Theoretical Limit for Safety Verification Techniques with Regular Fix-point Computations. *IPL*, 108(1):1–2, 2008.
5. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular (Tree) Model Checking. *STTT*, 14(2):167–191, 2012.
6. H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications* (TATA).
7. Irène Durand and Marc Sylvestre. Left-linear bounded trss are inverse recognizability preserving. In *RTA*, volume 10 of *LIPIcs*, pages 361–376, 2011.
8. J. Engelfriet and L. Heyker. Context-free Hypergraph Grammars have the same Term-generating Power as Attribute Grammars. *Acta Informatica*, 29, 1992.

9. J. Engelfriet and J.J. Vereijken. Context-free Grammars and Concatenation of Graphs. *Acta Informatica*, 34:773–803, 1997.
10. Joost Engelfriet and Erik Meineche Schmidt. IO and OI (I). *Journal of Computer and System Sciences*, 15(3):328 – 353, 1977.
11. Joost Engelfriet and Erik Meineche Schmidt. IO and OI (II). *Journal of Computer and System Sciences*, 16(1):67 – 99, 1978.
12. Th. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In *RTA*, volume 1379 of *LNCS*, pages 151–165. Springer-Verlag, 1998.
13. Th. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *CADE*, volume 1831 of *LNAI*, pages 271–290. Springer-Verlag, 2000.
14. V. Gouranton, P. Réty, and H. Seidl. Synchronized Tree Languages Revisited and New Applications. In *FoSSaCS*, volume 2030 of *LNCS*. Springer, 2001.
15. J. Kochems and C.-H. Luke Ong. Improved Functional Flow and Reachability Analyses Using Indexed Linear Tree Grammars. In *RTA*, volume 10 of *LIPIcs*, pages 187–202, 2011.
16. 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.
17. S. Limet and G. Salzer. Tree Tuple Languages from the Logic Programming Point of View. *Journal of Automated Reasoning*, 37(4):323–349, 2006.
18. J.C. Raoult. Rational Tree Relations. *Bulletin of the Belgian Mathematical Society Simon Stevin*, 4:149–176, 1997.
19. P. Réty. Langages synchronisés d'arbres et applications. Habilitation Thesis (in French). LIFO, Université d'Orléans. Technical report, June 2001.
20. P. Réty, J. Chabin, and J. Chen. R-Unification thanks to Synchronized-Contextfree Tree Languages. In *UNIF*, 2005.
21. P. Réty, J. Chabin, and J. Chen. Synchronized ContextFree Tree-tuple Languages. Technical Report RR-2006-13, LIFO, 2006.
22. William C. Rounds. Context-free grammars on trees. In Patrick C. Fischer, Seymour Ginsburg, and Michael A. Harrison, editors, *STOC*. ACM, 1969.

Appendix

A Intermediary Technical Results

The following technical lemmas are necessary for proving Lemma 1.

Lemma 3. *Let t and t' be two terms such that $Var(t) \cap Var(t') = \emptyset$. Suppose that t' is linear. Let σ be the most general unifier of t and t'. Then, one has: $\forall x, y : (x, y \in Var(t) \wedge x \neq y) \Rightarrow Var(\sigma(x)) \cap Var(\sigma(y)) = \emptyset$ and $\forall x : x \in Var(t) \Rightarrow \sigma(x)$ is linear.*

Proof. By Definition, $\sigma(t) = \sigma(t')$. Let us focus on t. Let $p_1, \ldots, p_n$ be positions of variables occurring in t. For any p_i,

- if $\sigma(t)|_{p_i}$ is a variable then $\sigma(t|_{p_i})$ is linear;
- Suppose that $\sigma(t)|_{p_i}$ is not a variable. Since σ is the most general unifier, there exists p_j such that $\sigma(t|_{p_i}) = t'|_{p_j}$. In particular p_j may be different from p_i if the variable occurring at position p_i in t occurs more than once in t. The term t' being linear, so is $\sigma(t|_{p_i})$.

For any p_i, p_j such that $t|_{p_i} \neq t|_{p_j}$, one has to study the different cases presented below:

- $\sigma(t)|_{p_i}$ and $\sigma(t)|_{p_j}$ are variables: Necessarily, $\sigma(t)|_{p_i} \neq \sigma(t)|_{p'_i}$ because t' is linear and $t|_{p_i} \neq t|_{p_j}$. Consequently, $Var(\sigma(t|_{p_i})) \cap Var(\sigma(t|_{p_i})) = \emptyset$.
- $\sigma(t)|_{p_i}$ is a variable and $\sigma(t)|_{p_j}$ is not: Consequently, there exists p_k such that $\sigma(t)|_{p_j} = t'|_{p_k}$. Indeed, if the variable $t|_{p_j}$ occurs only once in t then $p_k = p_j$. Otherwise, p_k is a position such that $t|_{p_j} = t|_{p_k}$. This position exists since the most general unifier exists.
  - $\sigma(t|_{p_i}) \in Var(t)$: necessarily, $Var(\sigma(t|_{p_i})) \cap Var(\sigma(t|_{p_j})) = \emptyset$ since $Var(\sigma(t|_{p_j})) = Var(t'|_{p_k}) \subseteq Var(t')$ and by hypothesis $Var(t) \cap Var(t') = \emptyset$
  - $\sigma(t|_{p_i}) \in Var(t')$: As for $t|_{p_j}$, since σ is the most general unifier, there exists p_l a position of $p_1, \ldots, p_n$ such that $t|_{p_l} = t|_{p_i}$ and $\sigma(t|_{p_i}) = t'|_{p_l}$. Moreover, $t|_{p_i} \neq t|_{p_j}$. Thus, $p_l \neq p_k$. Since t' is linear, $Var(t'|_{p_l}) \cap Var(t'|_{p_k}) = \emptyset$. Finally, one has that $Var(\sigma(t|_{p_i})) \cap Var(\sigma(t|_{p_j})) = \emptyset$.
- $\sigma(t)|_{p_j}$ is a variable and $\sigma(t)|_{p_i}$ is not: similar to the previous case.
- $\sigma(t|_{p_i})$ and $\sigma(t|_{p_j})$ are not variables: Once again, since σ is the mgu of t and t', there exists p_k and p_l such that $t|_{p_k} = t|_{p_i}$, $t|_{p_l} = t|_{p_j}$, $p_l \neq p_k$, $\sigma(t|_{p_i}) = t'|_{p_k}$ and $\sigma(t|_{p_j}) = t'|_{p_l}$. The term t' being linear, $\sigma(t|_{p_i}) \cap \sigma(t|_{p_j}) = \emptyset$.

Concluding the proof.

For the next lemmas, we introduce two notions allowing the extraction of variables occurring once in a sequence of atoms.

Definition 12. *Let G be a sequence of atoms. $Var^{in}_{Lin}(G)$ is a tuple of variables occurring in $In(G)$ and not in $Out(G)$, and $Var^{out}_{Lin}(G)$ is a tuple of variable occurring in $Out(G)$ and not in $In(G)$.*

Example 12. Let G be a sequence of atoms s.t. $G = P(g(\widehat{f(x', z')}), y'), Q(\widehat{v'}, g(z'))$. Consequently, $Var^{in}_{Lin}(G) = (y')$ and $Var^{out}_{Lin}(G) = (x', v')$.

Note that for a matter of simplicity, we denote by $x \in Var^{in}_{Lin}(G)$ (resp. $x \in Var^{out}_{Lin}(G)$) that x occurs in the tuple $Var^{in}_{Lin}(G)$ (resp. $Var^{out}_{Lin}(\ G)$). The following lemma focuses on a property of a sequence of atoms obtained after a resolution step.

Lemma 4. *Let Prog be a non-copying S-CF program, and G be a sequence of atoms such that $Out(G)$ is linear, $In(G)$ is linear and G does not contain loops. We assume*[10] *that variables occurring in Prog are different from those occurring in G. If $G \rightsquigarrow_\sigma G'$, then G' is loop free, $\sigma(Var^{in}_{Lin}(G)).Out(G')$ and $\sigma(Var^{out}_{Lin}(G)).In(G')$ are both linear.*

Example 13. Let $Prog = \{P(\widehat{g(x)}, y) \leftarrow Q(\widehat{x}, f(y))\}$ and $G = P(\widehat{g(f(x'))}, y')$. Then $G \rightsquigarrow_\sigma G'$ with $\sigma = (x/f(x'), y/y')$ and $G' = Q(\widehat{f(x')}, f(y'))$. Note that $\sigma(Var^{in}_{Lin}(G)).Out(G') = (y', f(x'))$ is linear.

Proof. First, we show that $\sigma(Var^{in}_{Lin}(G)).\ Out(G')$ and $\sigma(Var^{out}_{Lin}(G)).In(G')$ are linear. Thus, in a second time, we show that G' is loop free.

Suppose that $G \rightsquigarrow_\sigma G'$. Thus, there exist an atom A_x in $G = A_1, \ldots, A_x, \ldots, A_n$, a S-CF-clause $H \leftarrow B \in Prog$ and the mgu σ such that $\sigma(H) = \sigma(A_x)$ and $G' = \sigma(G)[\sigma(A_x) \leftarrow \sigma(B)]$.

Let $Var^{in}_{Lin}(G) = x_1, \ldots, x_k, \ldots, x_{k+n'}, \ldots, x_m$ built as follows:

- $x_1, \ldots, x_{k-1}$ are the variables occurring in the atoms $A_1, \ldots, A_{x-1}$;
- $x_k, \ldots, x_{k+n'}$ are the variables occurring in A_x;
- $x_{k+n'+1}, \ldots, x_m$ are the variable occurring the atoms $A_{x+1}, \ldots, A_n$.

Since $In(G)$ and $Out(G)$ are both linear and σ is the mgu of A_x and H, one has $\sigma(Var^{in}_{Lin}(G)) = x_1, \ldots, x_{k+1}, \sigma(x_k), \ldots, \sigma(x_{k+n'}), x_{k+n'+1}, \ldots x_m$. Note that the linearity of $In(G)$ involves the linearity of $Var^{in}_{Lin}(G)$. Moreover, one can deduce that $\sigma(Var^{in}_{Lin}(G))$ is linear iff the tuple $\sigma(x_k), \ldots, \sigma(\ x_{k+n'})$ is linear.

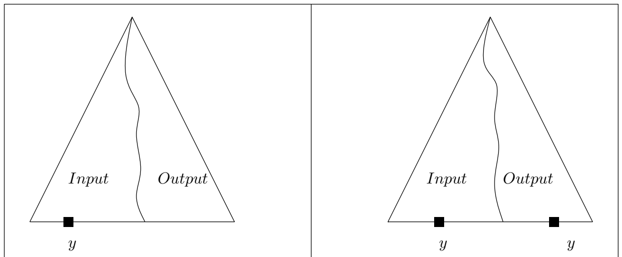
By hypothesis, $Out(H).In(B)$ and $Out(B).In(H)$ are both linear.

So, a variable occurring in $Var(H) \cap Var(B)$ is either

- a variable that is in $Out(H)$ and $Out(B)$ or
- a variable that is in $In(H)$ and $In(B)$.

A variable occurring in $Out(H)$ and in $In(H)$ does not occur in B. Symmetrically, a variable occurring in $Out(B)$ and in $In(B)$ does not occur in H. Moreover, a variable cannot occur twice in either $Out(H)$ or $In(H)$.

Let us focus on A_x. A_x is linear since it does not contain loop by hypothesis. Let us study the possible forms of H given in Fig. 3.



(a) Linear (b) non linear

Fig. 3. Possible forms of H

Each variable y occurring in B is:

- either a new variable or
- a variable occurring once in H and preserving its nature (input or output).

The relation $\rightsquigarrow_{Prog}$ ensures the nature stability of variables i.e.

$$Var(Out(\sigma(B))) \cap Var(In(\sigma(H))) = \emptyset \text{ and} \tag{1}$$
$$Var(In(\sigma(B))) \cap Var(Out(\sigma(H))) = \emptyset \tag{2}$$

Moreover, a consequence of Lemma 3 is that $Out(\sigma(B))$ and $In(\sigma(B))$ are both linear.

Let us study the two possible cases:

(a) since the variables of H and the variables of G are supposed to be disjointed and $Var^{in}_{Lin}(G)$ is linear, $\sigma(Var^{in}_{Lin}(G)) = x_1, \ldots, x_{k+1}, \sigma(x_k), \ldots, \sigma(x_{k+n'})$, $x_{k+n'+1}, \ldots, x_m$ is also linear. Moreover, considering H as linear and (1) and (2), a consequence is that

$$\bigcup_{x_i, i \in \{k,\ldots,k+n'\}} Var(\sigma(x_i)) \subseteq \{x_k, \ldots, x_{k+n'}\} \cup Var^{in}(A_x).$$

One can also deduce that $Var^{out}(G') \subseteq Var^{out}(G) \cup (Var^{out}(B))$. Consequently, $Var^{out}(G') \cap Var(\sigma(Var^{in}_{Lin}(G))) = \emptyset$ and the tuple $\sigma(Var^{in}_{Lin}(G)).Out(G')$ is linear iff $Out(G')$ is linear.

(b) A variable can occur at most twice in H but an occurrence of such a variable is necessarily an input variable and the other an output variable. Consequently the unification between A_x and H leads to a variable α of $\sigma(Var^{in}_{Lin}(G))$ occurring twice in $\sigma(H)$. But according to the form of H, these two occurrences of α do not occur in $\sigma(Var^{in}_{Lin}(G))$ since one of the two occurrences is necessarily at an output position. So, once again, the tuple $\sigma(Var^{in}_{Lin}(G)) = x_1, \ldots, x_{k+1}, \sigma(x_k), \ldots, \sigma(\ x_{k+n'}), x_{k+n'+1}, \ldots, x_m$ is linear. Moreover, $Prog$ being a non-copying S-CF program, for any variable x_i, with $i = k, \ldots k + n'$,

[10] If it is not the case then variables are relabelled.

- if $x_i \in Var(\sigma(x))$ with x a variable occurring twice in H then $Var(\sigma(x_i)) \cap Var^{out}(G') = \emptyset$;
- if there exists $z \in Var^{out}(A_x)$ such that $z \in Var(\sigma(x_i))$ and $z \in Var(\sigma(x))$ with x a variable occurring twice in H then $Var(\sigma(x_i)) \cap Var^{out}(G') = \emptyset$;
- if there exists $z \in Var^{out}(A_x)$ such that $x_i \in Var(\sigma(z))$ and $z \in Var(\sigma(x))$ with x a variable occurring twice in H then $Var(\sigma(x_i)) \cap Var^{out}(G') = \emptyset$;
- if there exists $x \in Var^{in}(H)$ such that $x \notin Var^{out}(H)$ then $Var(\sigma(x_i)) \subseteq \{x_k, \dots, x_{k+n'}\} \cup Var^{in}(A_x)$. Thus, $Var(\sigma(x_i)) \cap Var^{out}(G') = \emptyset$.

Consequently, $\sigma(Var^{in}_{Lin}(G)).Out(G')$ is linear iff $Out(G')$ is linear.

Let us now study the linearity of $Out(G')$. First, let us focus on the case $Out(\sigma(G - A_x))$ where $G - A_x$ is the sequence of atoms G for which the atom A_x has been removed. Note that $\sigma(G - A_x) = G' - \sigma(B)$.

Suppose that $Out(G - A_x)$ is not linear. So there exist two distinct variables x and y of G such that $Var(\sigma(x)) \cap Var(\sigma(y))$. Since these variables are concerned by the mgu σ, they are also variables of A_x at input positions as illustrated in Fig. 4. Since these variables are distinct and share the same variable by the application of σ, then there exist two subterms (red and green triangles in Fig. 4) at input positions in H sharing the same variable α. That is impossible since, by definition, for each $H \leftarrow B \in Prog$, one has $In(H).Out(B)$ and $Out(H).In(B)$ both linear.

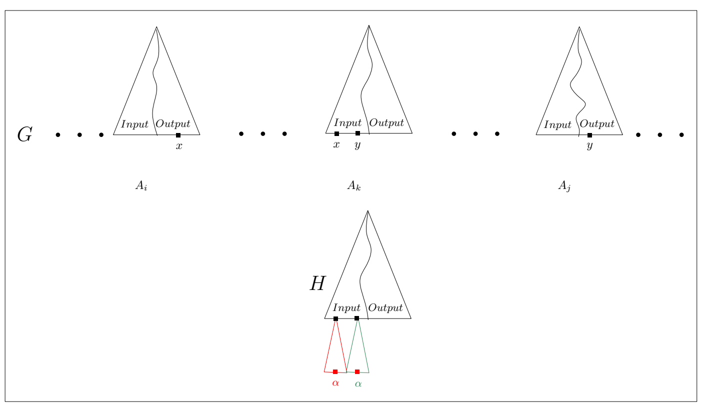


Fig. 4. $G - A_x$

So, the last possible case for breaking the linearity of $Out(G')$ is that there exist two distinct variables x and y such that x occurs in $Out(B)$, y occurs in $Out(G - A_x)$ and $Var(\sigma(x)) \cap Var(\sigma(y)) \neq \emptyset$. A variable α of $Var(\sigma(x)) \cap Var(\sigma(y))$ is necessarily a variable of H. Since a copy of α is done in the variable y and y necessarily occurs in A_x at an input position, there is a contradiction.

Indeed, it means that the variable α must occur both in $Out(H)$ and $In(H)$ but also in $Out(B)$. Thus, $H \leftarrow B$ is not a non-copying S-CF clause. Consequently, $Out(G')$ is linear.

To conclude, $\sigma(Var^{in}_{Lin}(G)).Out(G')$ **is linear**. Note that showing that $\sigma(Var^{out}_{Lin}(G)).In(G')$ is linear is very close.

The last remaining point to show is that G' does not contain any loops.

By construction, $G' = \sigma(G)[\sigma(A_x) \leftarrow \sigma(B)]$. There are three cases to study:

- Suppose there exists a loop occurring in $G' - \sigma(B)$: So, let us construct $G' - \sigma(B)$. By definition, $G' - \sigma(B) = \sigma(A_1), \dots \sigma(A_{x-1}), \sigma(A_{x+1}), \dots \sigma(A_m)$. Let us reason on the sequence of atoms G where $G = A_i, A_x, A_j$. Note that it can be easily generalized to a sequence of atoms of any size, but for a matter of simplicity, we focus on a significant sequence composed of three atoms. In that case, $G' - \sigma(B) = \sigma(A_i), \sigma(A_j)$. If there exist a loop in $G' - \sigma(B)$ but not in G then there are two possibilities (actually three but two of them are exactly symmetric):
  - $A_i \not\succ A_j$ and $A_j \not\succ A_i$: Consequently, σ has generated the loop. So, one can deduce that there exist two variables α and β such that $\alpha \in Var^{in}(\sigma(A_i)) \cap Var^{out}(\sigma(A_j))$ and $\beta \in Var^{out}(\sigma(A_i)) \cap Var^{in}(\sigma(A_j))$. Thus, there exist $y \in Var^{out}(A_i)$, $y' \in Var^{in}(A_i)$, $z \in Var^{out}(A_j)$ and $z' \in Var^{in}(A_j)$ such that $\alpha \in Var(\sigma(y')) \cap Var(\sigma(z))$ and $\beta \in Var(\sigma(y)) \cap Var(\sigma(z'))$. Since those four variables are concerned by the mgu, one can deduce that they also occur in A_x. More precisely, according to the linearity of $In(G)$ and $Out(G)$, $y' \in Var^{out}(A_x)$, $y \in Var^{in}(A_x)$, $z \in Var^{in}(A_x)$ and $z' \in Var^{out}(A_x)$. In that case, $A_i \succ A_x$ and $A_x \succ A_i$ because $y' \in Var^{out}(A_x) \cap Var^{in}(A_i)$ and $y \in Var^{out}(A_i) \cap Var^{in}(A_x)$. Consequently, a loop occurs in G. Contradiction.
  - $A_i \succ A_j$ and $A_j \not\succ A_i$: Consequently, σ has generated the loop. Since $A_i \succ A_j$, then there exists a variable y such that $y \in Var^{in}(A_i) \cap Var^{out}(A_j)$. Moreover, if there exists a loop in $G' - \sigma(B)$ then there exists a variable α such that $\alpha \in Var^{out}(\sigma(A_i)) \cap Var^{in}(\sigma(A_j))$. Thus, there exist two variables y' and z' with $y' \in Var^{out}(A_i)$ and $z' \in Var^{in}(A_j)$ such that $\alpha \in Var(\sigma(y')) \cap Var(\sigma(z'))$. Since those two variables are concerned by the mgu, one can deduce that they also occur in A_x. More precisely, according to the linearity of $In(G)$ and $Out(G)$, $y' \in Var^{in}(A_x)$ and $z' \in Var^{out}(A_x)$. In that case, one has $A_x \succ A_i$ and $A_j \succ A_x$ because $y' \in Var^{in}(A_x) \cap Var^{out}(A_i)$ and $z' \in Var^{out}(A_x) \cap Var^{in}(A_j)$. Moreover, by hypothesis, $A_i \succ A_j$. Consequently, a loop occurs in G because $A_j \succ A_x \succ A_i \succ A_x$. Contradiction.
- A loop cannot occur in $\sigma(B)$: This is a direct consequence of Lemma 3. Indeed, σ is the mgu of A_x which is linear and H. B is constructed from the variables occurring once in H and new variables. Moreover, $In(B)$ and $Out(B)$ are linear and the only variables allowed to appear in both $In(B)$ and $Out(B)$ are necessarily new and then not instantiated by σ. To create a loop in these conditions would require that two different variables α and β

instantiated by σ would share the same variable i.e. $Var(\sigma(\alpha)) \cap Var(\sigma(\beta)) \neq \emptyset$. Contradicting Lemma 3.

– Suppose that a loop occurs in G' but neither in $G' - \sigma(B)$ nor in $\sigma(B)$: Let G be the sequence of atoms such that $G = A_i, A_x$. In that case, $G' = \sigma(A_i), \sigma(B)$ with σ the mgu of A_x and H. One can extend the schema to any kind of sequence of atoms satisfying the hypothesis of this lemma without loss of generality. We consider B as follows: $B = B_1, \ldots, B_k$. If there exists a loop in G' but neither in $G' - \sigma(B)$ nor in $\sigma(B)$ then there exist $B_{k_1}, \ldots, B_{k_n}$ atoms occurring in B such that $\sigma(A_i) \succ \sigma(B_{k_1}) \succ \ldots \succ \sigma(B_{k_n}) \succ \sigma(A_i)$. So, one can deduce that there exists two variables α and β such that $\alpha \in Var^{in}(\sigma(A_i)) \cap Var^{out}(\sigma(B_{k_1}))$ and $\beta \in Var^{out}(\sigma(A_i)) \cap Var^{out}(\sigma(B_{k_n})))$. Consequently, there exists two variables y, z such that $y \in Var^{in}(A_i)$, $z \in Var^{out}(A_i)$, $\alpha \in Var(\sigma(y))$ and $\beta \in Var(\sigma(z))$. Both variables also occur in A_x. Suppose that y does not occur in A_x. Since σ is the mgu of A_x and H and y not in $Var(A_x)$, σ does not instantiate y. Consequently, $\alpha = y$. However, $Var(\sigma(B)) \subseteq Var(H) \cup Var(A_x) \cup Var(B)$. Moreover, the sets of variables occurring in $Prog$ and in G are supposed to be disjointed. So, y cannot occur in $\sigma(B)$ and then the loop in G' does not exist. Thus, y occurs in A_x as well as z. Furthermore, since $In(G)$ and $Out(G)$ are linear, $y \in Var^{out}(A_x)$ and $z \in Var^{in}(A_x)$. Consequently, G contains a cycle. Contradicting the hypothesis.
  To conclude, G' does not contain any loop.

Lemma 4 can be generalized to several steps.

Lemma 5. *The assumptions are those of Lemma 4. If* $G \rightsquigarrow^*_\sigma G'$*, then* G' *is loop free,* $\sigma(Var^{in}_{Lin}(G)).Out(G')$ *and* $\sigma(Var^{out}_{Lin}(G)).In(G')$ *are both linear.*

Proof. Let $G \rightsquigarrow^*_\sigma G'$ be rewritten as follows: $G_0 \rightsquigarrow_{\sigma_1} G_1 \ldots \rightsquigarrow_{\sigma_k} G_k$ with $G_0 = G$, $G' = G_k$ and $\sigma = \sigma_k \circ \ldots \circ \sigma_1$. Let P_k be the induction hypothesis defined such that: If $G_0 \rightsquigarrow^*_\sigma G_k$ then

– G_k does not contain any loop,
– $\sigma(Var^{in}_{Lin}(G_0)).Out(G_k)$ is linear and
– $\sigma(Var^{out}_{Lin}(G_0)).In(G_k)$ is linear.

Let us proceed by induction.

– P_0 is trivially true. Indeed, $In(G_0)$ and $Out(G_0)$ are linear. Moreover, for any $x \in Var^{in}_{Lin}(G_0)$ (resp. $x \in Var^{out}_{Lin}(G_0)$), one has $x \notin Var(Out(G_0))$ (resp. $x \notin Var(In(G_0))$). Thus, $Var^{in}_{Lin}(G_0).Out(G_0)$ is linear (resp. $Var^{out}_{Lin}(G_0).In(G_0)$).
– Suppose that P_k is true and $G_k \rightsquigarrow_{\sigma_{k+1}} G_{k+1}$. Since one has $G_k \rightsquigarrow_{\sigma_{k+1}} G_{k+1}$, there exist $H \leftarrow B \in Prog$ and an atom A_x occurring in $G_k$such that σ_{k+1} is the mgu of A_x and H, and $G_{k+1} = \sigma_{k+1}(G_k)[\sigma_{k+1}(H) \leftarrow \sigma_{k+1}(B)]$. By hypothesis, one has $Out(G_k)$ and $In(G_k)$ linear. Consequently, Lemma 4 can be applied and one obtains that
  - $\sigma(Var^{in}_{Lin}(G_k)).Out(G_{k+1})$ is linear,
  - $\sigma(Var^{out}_{Lin}(G_k).In(G_{k+1})$ is linear and

- G_{k+1} does not contain any loop.

Moreover, for $Prog$ a non-copying S-CF program, if $G_i \leadsto_{\sigma_{i+1}} G_{i+1}$ then one has: For any variable x, y, if $x \in Var^{in}_{Lin}(G_i)$ and $y \in Var(\sigma_{i+1}(x))$ then $y \in Var^{in}_{Lin}(G_{i+1})$ or $y \notin Var(G_{i+1})$. So, one can conclude that given $\sigma_k \circ \ldots \circ \sigma_1(Var^{in}_{Lin}(G_0))$, for any variable $x \in Var^{in}_{Lin}(G_0)$, for any $y \in Var(\sigma_k \circ \ldots \circ \sigma_1(x))$, either $y \in Var^{in}_{Lin}(G_k)$ or $y \notin Var(G_k)$.
Let us study the variables of $\bigcup_{y \in Var^{in}_{Lin}(G_0)}(Var(\sigma_k \circ \ldots \circ \sigma_1(y))$.

- For any variable x such that $x \in \bigcup_{y \in Var^{in}_{Lin}(G_0)}(Var(\sigma_k \circ \ldots \circ \sigma_1(y))) \setminus Var(G_k)$, $x \notin Var(G_{k+1})$. Indeed, an already-used variable cannot be reused for relabelling variables of $Prog$ while the reduction process. Moreover such variables are not instantiated by σ_{k+1} since the mgu σ_{k+1} of A_x and H only concerns variables of $Var(H) \cup Var(A_x)$. So, for any variable y in $Var(\sigma_k \circ \ldots \circ \sigma_1(y)) \setminus Var(G_k)$, one has $\sigma_{k+1}(y) = y$ and $y \notin Var(G_{k+1})$. Consequently, for any variable y in $Var(\sigma_{k+1} \circ \sigma_k \circ \ldots \circ \sigma_1(y))) \setminus Var(G_k)$, $y \notin Var(G_{k+1})$.
- For any variable x such that $x \in \bigcup_{y \in Var^{in}_{Lin}(G_0)}(Var(\sigma_k \circ \ldots \circ \sigma_1(y))) \cap Var(G_k)$, one can deduce that $x \in Var^{in}_{Lin}(G_k)$. Since $\sigma_{k+1}(Var^{in}_{Lin}(G_k)).Out(G_{k+1})$ is linear, one can deduce that for any $y \in \bigcup_{y \in Var^{in}_{Lin}(G_0)}(Var(\sigma_k \circ \ldots \circ \sigma_1(y))) \cap Var(G_k)$, $Var(\sigma_{k+1} \circ \sigma_k \circ \ldots \circ \sigma_1(y)) \cap Var(Out(G_{k+1})) = \emptyset$.

So, one has $\sigma_{k+1} \circ \sigma_k \circ \ldots \circ \sigma_1(Var^{in}_{Lin}(G_k)).Out(G_{k+1})$ is linear. The proof of $\sigma(Var^{out}_{Lin}(G_k).In(G_{k+1})$ is in some sense symmetric. To conclude, considering the hypothesis of Lemma 4, one has: If $G \leadsto^*_\sigma G'$, then

- G' is loop free;
- $\sigma(Var^{in}_{Lin}(G)).Out(G')$ is linear;
- $\sigma(Var^{out}_{Lin}(G)).In(G')$ is linear.

B Proof of Lemma 1

Proof. Let $G_0 = P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n)$. Since l is linear, G_0 is linear and $Var^{in}_{Lin}(G_0) = In(G_0)$. From Lemma 5, $\theta(In(G_0)).Out(G)$ is linear and G is loop-free. Note that $In(G_0)$ and $Out(G)$ are tuples of variables. Since the critical pair is strict,we deduce that θ does not instantiate the variables of $In(G_0)$, then $\theta(In(G_0)).Out(G)$ is a linear tuple of variables. Consequently, a strict critical pair is a S-CF clause.

Since G_0 is linear, $Var^{out}_{Lin}(G_0) = Var^{out}(G_0)$. Thus, from Lemma 5, $\theta(Out(G_0)).In(G)$ is linear. And since r is linear, the critical pair is a non-copying clause.

Let $Prog = \{P(\widehat{f(x)}, x) \leftarrow .\}$ and consider the rewrite rule $f(a) \to b$. Thus $P(\widehat{f(a)}, y) \leadsto_{(x/a,\, y/a)} \emptyset$, which gives rise to the extended critical pair $P(\widehat{b}, a) \leftarrow .$, which is not a S-CF clause.

C Proof of Lemma 2

Proof. Consider $f(\vec{s}) \to r \in R$ ($\vec{s}$ is a tuple of terms), and assume that

$$P(\widehat{\vec{x_1}}, \widehat{f(\vec{s})}, \widehat{\vec{x_2}}, \vec{z}) \leadsto_{[P(\widehat{\vec{t_1}}, \widehat{f(\vec{u})}, \widehat{\vec{t_2}}, \vec{v}) \leftarrow B,\, \theta]} G \leadsto^*_\sigma G'$$

such that $Out(G')$ is flat, $\vec{x_1}$, $\vec{x_2}$, $\vec{z}$, $\vec{u}$, $\vec{v}$ are tuples of distinct variables and $\vec{t_1}$, $\vec{t_2}$ are tuples of terms (however $\vec{v}$ may share some variables with $\vec{t_1}.\vec{u}.\vec{t_2}$). This derivation generates the critical pair $(\sigma \circ \theta)(P(\widehat{\vec{x_1}}, \widehat{r}, \widehat{\vec{x_2}}, \vec{z})) \leftarrow G'$.

If $l \to r$ is consuming then P has no input arguments, i.e. $\vec{z}$ and $\vec{v}$ do not exist. Therefore $\sigma \circ \theta$ cannot instantiate the input variables of P, hence the critical pair is strict.

Otherwise $\vec{s}$ is a linear tuple of variables, and (x/t means that the variable x is replaced by t) $\theta = (\vec{v}/\vec{z}) \circ (\vec{x_1}/\vec{t_1},\ \vec{s}/\vec{u},\ \vec{x_2}/\vec{t_2})$, which does not instantiate $\vec{z}$ nor the output variables of B. Moreover $Out(B)$ is flat, then $Out(G) = Out(\theta B)$ is flat. Thus $G' = G$ and the critical pair is $P(\widehat{\theta\vec{x_1}}, \widehat{\theta r}, \widehat{\theta\vec{x_2}}, \vec{z}) \leftarrow G$, which is strict.

D Proof of Theorem 2

Proof. Let $A \in Mod(Prog)$ s.t. $A \to_{l \to r} A'$. 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 of C one by one. Since $Prog$ is coherent with R, the top symbol of l cannot be generated as an input: it is either consumed in an output argument, or the whole $\sigma(l)$ disappears thanks to an output argument. Consequently $G_0{=}A \leadsto^* G_k \leadsto^* \emptyset$ and there exists an atom $A'' = P(t_1, \dots, t_n)$ in G_k and an output argument j s.t. $t_j = \sigma(l)$, and along the step $G_k \leadsto G_{k+1}$ the top symbol of t_j is consumed or t_j disappears entirely. On the other hand, since $Prog$ is non-copying, $A' \leadsto^* G_k[A'' \leftarrow P(t_1, \dots, \sigma(r), \dots, t_n)]$.

If $t_j = \sigma(l)$ disappears entirely, it can be replaced by any term, then $A' \leadsto^* G_k[A'' \leftarrow P(t_1, \dots, \sigma(r), \dots, t_n)] \leadsto^* \emptyset$, hence $A' \in Mod(Prog)$. Otherwise the top symbol of $\sigma(l)$ is consumed along $G_k \leadsto G_{k+1}$. Consider new variables $x_1, \dots, x_n$ such that $\{x_1, \dots, x_n\} \cap Var(l) = \emptyset$, and let us define the substitution σ' by $\forall i \in \{1, \dots, n\}$, $\sigma'(x_i) = t_i$ and $\forall x \in Var(l)$, $\sigma'(x) = \sigma(x)$. Then $\sigma'(P(x_1, \dots, x_{j-1}, l, x_{j+1}, \dots, x_n)) = A''$, and according to resolution (or narrowing) properties $P(x_1, \dots, l, \dots, x_n) \leadsto^*_\theta \emptyset$ and $\theta \le \sigma'$. This derivation can be decomposed into: $P(x_1, \dots, l, \dots, x_n) \leadsto^*_{\theta_1} G' \leadsto_{\theta_2} G \leadsto^*_{\theta_3} \emptyset$ where $\theta = \theta_3 \circ \theta_2 \circ \theta_1$, and s.t. $Out(G')$ is not flat and $Out(G)$ is flat[11].
The derivation $P(x_1, \dots, l, \dots, x_n) \leadsto^*_{\theta_1} G' \leadsto_{\theta_2} G$ can be commuted into: $P(x_1, \dots, l, \dots, x_n) \leadsto^*_{\gamma_1} B' \leadsto_{\gamma_2} B \leadsto^*_{\gamma_3} G$ s.t. $Out(B)$ is flat, $Out(B')$ is not flat, and within $P(x_1, \dots, l, \dots, x_n) \leadsto^*_{\gamma_1} B' \leadsto_{\gamma_2} B$ resolution is applied only on atoms whose output is not flat, and we have $\gamma_3 \circ \gamma_2 \circ \gamma_1 = \theta_2 \circ \theta_1$. Then

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

$\gamma_2 \circ \gamma_1(P(x_1, \ldots, r, \ldots, x_n)) \leftarrow B$ is a critical pair. By hypothesis, it is convergent, then $\gamma_2 \circ \gamma_1(P(x_1, \ldots, r, \ldots, x_n)) \to^* B$. Note that $\gamma_3(B) \to^* G$ and recall that $\theta_3 \circ \gamma_3 \circ \gamma_2 \circ \gamma_1 = \theta_3 \circ \theta_2 \circ \theta_1 = \theta$. Then $\theta(P(x_1, \ldots, r, \ldots, x_n)) \to^* \theta_3(G) \to^* \emptyset$, and since $\theta \leq \sigma'$ we get $P(t_1, \ldots, \sigma(r), \ldots, t_n) = \sigma'(P(x_1, \ldots, r, \ldots, x_n)) \to^* \emptyset$. Therefore $A' \rightsquigarrow^* G_k[A'' \leftarrow P(t_1, \ldots, \sigma(r), \ldots, t_n)] \rightsquigarrow^* \emptyset$, hence $A' \in Mod(Prog)$. By trivial induction, the proof can be extended to the case of several rewrite steps.

E Ensuring finitely many critical pairs

The following example illustrates a situation where the number of critical pairs is infinite.

Example 14. Let $\Sigma = \{f^{\backslash 2}, c^{\backslash 1}, d^{\backslash 1}, s^{\backslash 1}, a^{\backslash 0}\}$, $f(c(x), y) \to d(y)$ be a rewrite rule, and $\{P_0(\widehat{f(x,y)}) \leftarrow P_1(\widehat{x}, \widehat{y}).\ P_1(\widehat{x}, \widehat{s(y)}) \leftarrow P_1(\widehat{x}, \widehat{y}).\ P_1(\widehat{c(x)}, \widehat{y}) \leftarrow P_2(\widehat{x}, \widehat{y}).\ P_2(\widehat{a}, \widehat{a}) \leftarrow .\}$ a S-CF programs. Then $P_0(\widehat{f(c(x), y)}) \to P_1(\widehat{c(x)}, \widehat{y}) \rightsquigarrow_{y/s(y)} P_1(\widehat{c(x)}, \widehat{y}) \rightsquigarrow_{y/s(y)} \cdots P_1(\widehat{c(x)}, \widehat{y}) \to P_2(\widehat{x}, \widehat{y})$. Resolution is applied only on non-flat atoms and the last atom obtained by this derivation is flat. The composition of substitutions along this derivation gives $y/s^n(y)$ for some $n \in \mathbb{N}$. There are infinitely many such derivations, which generates infinitely many critical pairs of the form $P_0(d(\widehat{s^n(y)})) \leftarrow P_2(\widehat{x}, \widehat{y})$.

This is annoying since the completion process presented in the following needs to compute all critical pairs. This is why we define sufficient conditions to ensure that a given finite S-CF program has finitely many critical pairs.

Definition 13.
Prog is empty-recursive *if there exist a predicate symbol P and two tuples $\vec{x} = (x_1, \ldots, x_n)$, $\vec{y} = (y_1, \ldots, y_k)$ composed of distinct variables s.t. $P(\widehat{\vec{x}.\vec{y}}) \rightsquigarrow^+_\sigma A_1, \ldots, P(\widehat{\vec{x'}.\vec{t'}}), \ldots, A_k$ where $\vec{x'} = (x'_1, \ldots, x'_n)$ is a tuple of variables and there exist i, j s.t. $x'_i = \sigma(x_i)$ and $\sigma(x_j)$ is not a variable and $x'_j \in Var(\sigma(x_j))$.*

Example 15. Let $Prog$ be the S-CF program defined as follows:
$Prog = \{P(\widehat{x'}, \widehat{s(y')}) \leftarrow P(\widehat{x'}, \widehat{y'}).\ P(\widehat{a}, \widehat{b}) \leftarrow .\}$ From $P(\widehat{x}, \widehat{y})$, one can obtained the following derivation: $P(\widehat{x}, \widehat{y}) \rightsquigarrow_{[x/x',\, y/s(y')]} P(\widehat{x'}, \widehat{y'})$. Consequently, $Prog$ is empty-recursive since $\sigma = [x/x',\, y/s(y')]$, $x' = \sigma(x)$ and y' is a variable of $\sigma(y) = s(y')$.

The following lemma shows that the non empty-recursiveness of a S-CF program is sufficient to ensure the finiteness of the number of critical pairs.

Lemma 6. *Let $Prog$ be a normalized S-CF program.*
If $Prog$ is not empty-recursive, then the number of critical pairs is finite.

Remark 2. Note that the S-CF program of Example 14 is normalized and has infinitely many critical pairs. However it is empty-recursive because
$P_1(\widehat{x}, \widehat{y}) \rightsquigarrow_{[x/x',\, y/s(y')]} P_1(\widehat{x'}, \widehat{y'})$.

Proof. By contrapositive. Let us suppose there exist infinitely many critical pairs. So there exist P_1 and infinitely many derivations of the form
(i): $P_1(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n) \leadsto^*_\alpha G' \leadsto_\theta G$ (the number of steps is not bounded). As the number of predicates is finite and every predicate has a fixed arity, there exists a predicate P_2 and a derivation of the form
(ii): $P_2(t_1, \ldots, t_p) \leadsto^k_\sigma G''_1, P_2(t'_1, \ldots, t'_p), G''_2$ (with $k > 0$) included in some derivation of (i), strictly before the last step, such that:

1. $Out(G''_1)$ and $Out(G''_2)$ are flat and the derivation from $P_2(t_1, \ldots, t_p)$ can be applied on $P_2(t'_1, \ldots, t'_p)$ again, which gives rise to an infinite derivation.
2. σ is not empty and there exists a variable x in $P_2(t_1, \ldots, t_p)$ such that $\sigma(x) = t$ and t is not a variable and contains a variable y that occurs in $P_2(t'_1, \ldots, t'_p)$. Otherwise $\sigma \circ \ldots \circ \sigma$ would always be a variable renaming and there would be finitely many critical pairs.
3. There is at least one non-variable term (let t_j) in output arguments of $P_2(t_1, \ldots, t_p)$ (due to the definition of critical pairs) such that $t'_j = t_j$[12]. As we use a S-CF clause in each derivation step, the output argument t'_j matches a variable (output argument) in the body of the last clause used in (ii). As $t'_j = t_j$, the output argument t_j matches a variable (output argument) in head of the first clause used in (ii). So, for each variable x occurring in the non-variable output terms of P_2, we have $\sigma(x) = x$.
4. From the previous item, we deduce that the variable x found in item 2 is one of the terms $t_1, \ldots, t_p$, say t_k. We can assume that y is t'_k. t_k is an output argument of P_2 because it matches a non-variable and only output arguments are non-variable in the head of S-CF clause.

If in derivation (ii) we replace all non-variable output terms by new variables, we obtain a new derivation[13]
(iii): $P_2(x_1, \ldots, x_n, t_{n+1}, \ldots, t_p) \leadsto^k_{\sigma'} G'''_1, P_2(x'_1, \ldots, x'_n, t'_{n+1}, \ldots, t'_p), G'''_2$ and there exists i, k (in $\{1, \ldots n\}$) such that $\sigma'(x_i) = x'_i$ (at least one non-variable term (in output arguments) in the (ii) derivation), and $\sigma'(x_k) = t_k$, x'_k is a variable of t_k. We conclude that Prog is empty-recursive.

Deciding the empty-recursiveness of a S-CF program seems to be a difficult problem (undecidable ?). Nevertheless, we propose a sufficient syntactic condition to ensure that a S-CF program is not empty-recursive.

Definition 14.
The S-CF clause $P(\widehat{t_1}, \ldots, \widehat{t_n}, x_1, \ldots, x_k) \leftarrow A_1, \ldots, Q(\ldots), \ldots, A_m$ *is* pseudo-empty over Q *if there exist* i, j *s.t.*

– t_i *is a variable,*

[12] This property does not necessarily hold as soon as P_2 is reached within (ii). We may have to consider further occurrences of P_2 so that each required term occurs in the required argument, which will necessarily happen because there are only finitely many permutations.

[13] Without loss of generality, we can consider that the output arguments (at least two) are the first arguments of P_2.

– *and* t_j *is not a variable,*
– *and* $\exists x \in Var(t_j)$*,* $x \neq t_i \wedge \{x, t_i\} \subseteq VarOut(Q(\ldots))$*.*

Roughly speaking, when making a resolution step issued from the following flat atom $P(\widehat{y_1}, \ldots, \widehat{y_n}, z_1, \ldots, z_k)$*, the variable* y_i *is not instantiated, and* y_j *is instantiated by something that is synchronized with* y_i *(in* $Q(\ldots)$*).*

The S-CF clause $H \leftarrow B$ *is* pseudo-empty *if there exists some* Q *s.t.* $H \leftarrow B$ *is pseudo-empty over* Q*.*

The S-CF clause $P(\widehat{t_1}, \ldots, \widehat{t_n}, x_1, \ldots, x_{n'}) \leftarrow A_1, \ldots, Q(\widehat{y_1}, \ldots, \widehat{y_k}, s_1, \ldots, s_{k'}), \ldots, A_m$ *is* empty over Q *if for all* y_i*,* $(\exists j,\ t_j = y_i$ *or* $y_i \notin Var(P(\widehat{t_1}, \ldots, \widehat{t_n}, x_1, \ldots, x_{n'})))$*.*

Example 16. The S-CF clause $P(\widehat{x}, \widehat{f(x)}, \widehat{z}) \leftarrow Q(\widehat{x}, \widehat{z})$ is both pseudo-empty (thanks to the second and the third argument of P) and empty over Q (thanks to the first and the third argument of P).

Definition 15. *Using Definition 14, let us define two relations over predicate symbols.*

– $P_1 \trianglerighteq_{Prog} P_2$ *if there exists in* $Prog$ *a clause empty over* P_2 *of the form* $P_1(\ldots) \leftarrow A_1, \ldots, P_2(\ldots), \ldots, A_n$*. The reflexive-transitive closure of* $\trianglerighteq_{Prog}$ *is denoted by* $\trianglerighteq^*_{Prog}$*.*
– $P_1 >_{Prog} P_2$ *if there exist in* $Prog$ *predicates* P'_1*,* P'_2 *s.t.* $P_1 \trianglerighteq^*_{Prog} P'_1$ *and* $P'_2 \trianglerighteq^*_{Prog} P_2$*, and a clause pseudo-empty over* P'_2 *of the form* $P'_1(\ldots) \leftarrow A_1, \ldots, P'_2(\ldots), \ldots, A_n$*. The transitive closure of* $>_{Prog}$ *is denoted by* $>^+_{Prog}$*.*

$Prog$ *is* cyclic *if there exists a predicate* P *s.t.* $P >^+_{Prog} P$*.*

Example 17. Let $\Sigma = \{f^{\backslash 1}, h^{\backslash 1}, a^{\backslash 0}\}$. Let $Prog$ be the following S-CF program $\{P(\widehat{x}, \widehat{h(y)}, \widehat{f(z)}) \leftarrow Q(\widehat{x}, \widehat{z}), R(\widehat{y}).\ Q(\widehat{x}, \widehat{g(y,z)}) \leftarrow P(\widehat{x}, \widehat{y}, \widehat{z}).\ R(\widehat{a}) \leftarrow .\ Q(\widehat{a}, \widehat{a}) \leftarrow .\}$. One has $P >_{Prog} Q$ and $Q >_{Prog} P$. Thus, $Prog$ is cyclic.

The lack of cycles is the key point of our technique since it ensures the finiteness of the number of critical pairs.

Lemma 7. *If* $Prog$ *is not cyclic, then* $Prog$ *is not empty-recursive, consequently the number of critical pairs is finite.*

Proof. By contrapositive. Let us suppose that $Prog$ is empty recursive. So it exists P s.t. $P(\widehat{x_1}, \ldots, \widehat{x_n}, y_1, \ldots, y_{n'}) \leadsto^+_\sigma A_1, \ldots, P(\widehat{x'_1}, \ldots, \widehat{x'_n}, t'_1, \ldots, t'_{n'}), \ldots, A_k$ where $x'_1, \ldots, x'_n$ are variables and there exist i, j s.t. $x'_i = \sigma(x_i)$ and $\sigma(x_j)$ is not a variable and $x'_j \in Var(\sigma(x_j))$. We can extract from the previous derivation the following derivation which has p steps ($p \geq 1$). $P(\widehat{x_1}, \ldots, \widehat{x_n}, y_1, \ldots, y_{n'}) =$
$Q^0(\widehat{x_1}, \ldots, \widehat{x_n}, y_1, \ldots, y_{n'}) \leadsto_{\alpha_1}$
$B^1_1 \ldots Q^1(\widehat{x^1_1}, \ldots, \widehat{x^1_{n_1}}, t^1_1, \ldots, t^1_{n'_1}) \ldots B^1_{k_1} \leadsto_{\alpha_2}$
$B^1_1 \ldots B^2_1 \ldots Q^2(\widehat{x^2_1}, \ldots, \widehat{x^2_{n_2}}, t^2_1, \ldots, t^2_{n'_2}) \ldots B^2_{k_2} \ldots B^1_{k_1} \leadsto_{\alpha_3} \ldots \leadsto_{\alpha_p}$

$B_1^1 \dots B_1^p \dots Q^p(\widehat{x_1^p}, \dots, \widehat{x_{n_p}^p}, t_1^p, \dots, t_{n'_p}^p) \dots B_{k_p}^p \dots B_{k_1}^1$
where $Q^p(\widehat{x_1^p}, \dots, \widehat{x_{n_p}^p}, t_1^p, \dots, t_{n'_p}^p) = P(\widehat{x'_1}, \dots, \widehat{x'_n}, t'_1, \dots, t'_{n'})$.
For each k (after k steps in the previous derivation), $\alpha_k \circ \alpha_{k-1} \dots \circ \alpha_1(x_i)$ is a variable of $Out(Q^k(\widehat{x_1^k}, \dots, \widehat{x_{n_k}^k}, t_1^k, \dots, t_{n'_k}^k))$ and $\alpha_k \circ \alpha_{k-1} \dots \circ \alpha_1(x_j)$ is either a variable of $Out(Q^k(\widehat{x_1^k}, \dots, \widehat{x_{n_k}^k}, t_1^k, \dots, t_{n'_k}^k))$ or a non-variable term containing a variable of $Out(Q^k(\widehat{x_1^k}, \dots, \widehat{x_{n_k}^k}, t_1^k, \dots, t_{n'_k}^k))$.

Each derivation step issued from Q^k uses either a clause pseudo-empty over Q^{k+1} and we deduce $Q^k >_{Prog} Q^{k+1}$, or an empty clause over Q^{k+1} and we deduce $Q^k \trianglerighteq_{Prog} Q^{k+1}$. At least one step uses a pseudo-empty clause otherwise no variable from $x_1, \dots, x_n$ would be instantiated by a non-variable term containing at least one variable in $x'_1, \dots, x'_n$.
We conclude that $P = Q^0\ op_1\ Q^1\ op_2\ Q^2 \dots Q^{p-1}\ op_p\ Q^p = P$ with each op_i is $>_{Prog}$ or $\trianglerighteq_{Prog}$ and there exists k such that op_k is $>_{Prog}$. Therefore $P >^+_{Prog} P$, so $Prog$ is cyclic.

Thus, if $Prog$ is not cyclic, then all is fine. Otherwise, we have to transform $Prog$ into $Prog'$ such as $Prog'$ is not cyclic and $Mod(Prog) \subseteq Mod(Prog')$.

The transformation is based on the following observation. If $Prog$ is cyclic, there is at least one pseudo-empty clause that participates in a cycle. In Example 17, $P(\widehat{x}, \widehat{h(y)}, \widehat{f(z)}) \leftarrow Q(\widehat{x}, \widehat{z}), R(\widehat{y})$ is a pseudo-empty clause over Q involved in the cycle. To remove the cycle, we transform it into $P(\widehat{x}, \widehat{h(y)}, \widehat{f(z)}) \leftarrow Q(\widehat{x}, \widehat{x_2}), R(\widehat{x_1}), Q(\widehat{x_3}, \widehat{z}), R(\widehat{y})$ (x_1, x_2, x_3 are new variables), which is not pseudo-empty anymore. The main process is described in Definition 19. Definitions 16, 17 and 18 are preliminary definitions used in Definition 19. Example 18 illustrates the definitions. If there are input arguments then some variables occurring in the input arguments of the body must also be renamed in order to get a non-copying S-CF clause.

Definition 16. P *is* unproductive *(in Prog) if*
$\neg(\exists t_1, \dots, t_n, t'_1, \dots, t'_k \in T_\Sigma,\ P(\widehat{t_1}, \dots, \widehat{t_n}, t'_1, \dots, t'_k) \in Mod(Prog))$.

Definition 17 (simplify)**.** *Let $H \leftarrow A_1, \dots, A_n$ be a S-CF clause, and for each i, let us write $A_i = P_i(\dots)$.*
If there exists P_i s.t. P_i is unproductive then simplify$(H \leftarrow A_1, \dots, A_n)$ *is the empty set, otherwise it is the set that contains only the clause $H \leftarrow B_1, \dots, B_m$ such that*

- $\{B_i \mid 0 \le i \le m\} \subseteq \{A_i \mid 0 \le i \le n\}$ *and*
- $\forall i \in \{1, \dots, n\}$, $(\neg(\exists j, B_j = A_i) \Leftrightarrow (Var(A_i) \cap Var(H) = \emptyset \wedge \forall k \neq i, Var(A_i) \cap Var(A_k) = \emptyset))$.

In other words, simplify *deletes unproductive clauses, or it removes the atoms of the body that contain only free variables.*

Remark 3. Let $H \leftarrow B$ be a non-copying S-CF clause. If the variable x occurs several times in B then $x \notin Var(H)$.

Definition 18 (unSync)**.** *Let $H \leftarrow B$ be a non-copying S-CF clause. Let us write $Out(H) = (t_1, \dots, t_n)$ and $In(B) = (s_1, \dots, s_k)$.*
$\mathsf{unSync}(H \leftarrow B) = \mathsf{simplify}(H \leftarrow \sigma_0(B), \sigma_1(B))$ *where σ_0, σ_1 are substitutions built as follows. $\forall x \in Var(B)$:*

$$\sigma_0(x) = \begin{cases} x \text{ if } x \in VarOut(B) \wedge \exists i,\, t_i = x \\ x \text{ if } x \in VarIn(B) \cap VarIn(H) \,\wedge\, \exists j,\, (s_j = x) \\ \text{a fresh variable otherwise} \end{cases}$$

$$\sigma_1(x) = \begin{cases} x \text{ if } x \in VarOut(B) \wedge \exists i,\, (t_i \notin Var \wedge x \in Var(t_i)) \\ x \text{ if } x \in VarIn(B) \cap VarIn(H) \,\wedge\, \exists j,\, (s_j \notin Var \wedge x \in Var(s_j)) \\ \text{a fresh variable otherwise} \end{cases}$$

Definition 19 (removeCycles)**.** *Let $Prog$ be a S-CF program. If $Prog$ is not cyclic,* $\mathsf{removeCycles}(Prog) = Prog$ *otherwise* $\mathsf{removeCycles}(Prog) = \mathsf{removeCycles}(\{\mathsf{unSync}(H \leftarrow B)\} \cup Prog')$ *where $H \leftarrow B$ is a pseudo-empty clause involved in a cycle and $Prog' = Prog \setminus \{H \leftarrow B\}$.*

Example 18. Let $Prog$ be the S-CF program of Example 17. Since $Prog$ is cyclic, let us compute $\mathsf{removeCycles}(Prog)$. The pseudo-empty S-CF clause $P(\widehat{x}, \widehat{h(y)}, \widehat{f(z)}) \leftarrow Q(\widehat{x}, \widehat{z}), R(\widehat{y})$ is involved in the cycle. Consequently, unSync is applied on it. According to Definition 18, one obtains σ_0 and σ_1 where $\sigma_0 = [x/x, y/x_1, z/x_2]$ and $\sigma_1 = [x/x_3, y/y, z/z]$. Thus, one obtains the S-CF clause $P(\widehat{x}, \widehat{h(y)}, \widehat{f(z)}) \leftarrow Q(\widehat{x}, \widehat{x_2}), R(\widehat{x_1}), Q(\widehat{x_3}, \widehat{z}), R(\widehat{y})$. Note that according to Definition 18, simplify is applied and removes $R(\widehat{x_1})$ from the body. Following Definitions 17 and 19, one has to remove $P(\widehat{x}, \widehat{h(y)}, \widehat{f(z)}) \leftarrow Q(\widehat{x}, \widehat{z}), R(\widehat{y})$ from $Prog$ and to add $P(\widehat{x}, \widehat{h(y)}, \widehat{f(z)}) \leftarrow Q(\widehat{x}, \widehat{x_2}), Q(\widehat{x_3}, \widehat{z}), R(\widehat{y})$ instead. Note that the atom $R(\widehat{x_1})$ has been removed using simplify. Note also that there is no cycle anymore.

Lemma 8 describes that our transformation preserves at least and may extend the initial least Herbrand Model.

Lemma 8. *Let $Prog$ be a non-copying S-CF program and $Prog' =$* $\mathsf{removeCycles}(Prog)$*. Then $Prog'$ is a non-copying and non-cyclic S-CF program, and $Mod(Prog) \subseteq Mod(Prog')$. Moreover, if $Prog$ is normalized, then so is $Prog'$.*

Consequently, there are finitely many critical pairs in $Prog'$.

F Proof of Theorem 3

Proof. The proof is straightforward.

G Example 10 in Detail

Let $I = \{f(a,a)\}$ and $R = \{f(x,y) \to u(f(v(x), w(y)))\}$. Intuitively, the exact set of descendants is $R^*(I) = \{u^n(f(v^n(a), w^n(a))) \mid n \in \mathbb{N}\}$ where u^n means that u occurs n times. We define $Prog = \{P_f(\widehat{f(x,y)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y})$. **(1)**, $P_a(\widehat{a}) \leftarrow .$ **(2)**$\}$. Note that $L_{Prog}(P_f) = I$.

Using clause (1) we have $P_f(\widehat{f(x,y)}) \to_{(1)} P_a(\widehat{x}), P_a(\widehat{y})$ generating the critical pair: $P_f(u(f(\widehat{v(x), w(y)}))) \leftarrow P_a(\widehat{x}), P_a(\widehat{y})$. In order to normalize this critical pair, we choose to generate symbols u, f as output, v, w as input. Moreover only one predicate symbol of arity 3 is allowed. It produces three new S-CF clauses : $P_f(\widehat{z}) \leftarrow P_1(\widehat{z}, x, y), P_a(\widehat{x}), P_a(\widehat{y})$. **(3)**, $P_1(\widehat{u(z)}, x, y) \leftarrow P_1(\widehat{z}, v(x), w(y))$. **(4)** and $P_1(\widehat{f(x,y)}, x, y) \leftarrow .$ **(5)**.

Now $P_f(\widehat{f(x',y')}) \to_{(3)} P_1(\widehat{f(x',y')}, x, y), P_a(\widehat{x}), P_a(\widehat{y}) \rightsquigarrow_{(5),\sigma} P_a(\widehat{x}), P_a(\widehat{y})$ where $\sigma = (x'/x,\, y'/y)$. It generates the critical pair $P_f(u(f(\widehat{v(x), w(y)})))\ ) \leftarrow P_a(\widehat{x}), P_a(\widehat{y})$ again, which is convergent. Since one has $P_1(\widehat{f(x',y')}, x, y) \rightsquigarrow_{(5),(x'/x,\, y'/y)} \emptyset$, the critical pair $P_1(u(f(\widehat{v(x), w(y)})), x, y) \leftarrow .$ can be computed, but it is already convergent.

No other critical pair is detected. Then, we get the S-CF program $Prog'$ composed of clauses (1) to (5), and note that $L_{Prog'}(P_f) = R^*(I)$ indeed.

H Example 11 in Detail

Let $I = \{d_1(a,a,a)\}$ and

$$R = \left\{ \begin{array}{ll} d_1(x,y,z) \xrightarrow{1} d_1(h(x), i(y), s(z)), & d_1(x,y,z) \xrightarrow{2} d_2(x,y,z) \\ d_2(x,y,s(z)) \xrightarrow{3} d_2(f(x), g(y), z), & d_2(x,y,a) \xrightarrow{4} c(x,y) \end{array} \right\}$$

$R^*(I)$ is composed of all terms appearing in the following derivation:

$$d_1(a,a,a) \xrightarrow{1}{}^{n} d_1(h^n(a), i^n(a), s^n(a)) \xrightarrow{2} d_2(h^n(a), i^n(a), s^n(a))$$
$$\xrightarrow{3}{}^{k} d_2(f^k(h^n(a)), g^k(i^n(a)), s^{n-k}(a)) \xrightarrow{4} c(f^n(h^n(a)), g^n(i^n(a)))\,.$$

Note that the last rewrite step by rule 4 is possible only when $k = n$. Let $Prog$ be an S-CF program such that $Prog = \{P_d(d_1\widehat{(x,y,z)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$. **(1)**, $P_a(\widehat{a}) \leftarrow .$ **(2)**$\}$. Thus $L_{Prog}(P_d) = I$.

By applying clause (1) and using rule 1, we get the critical pair: $P_d(d_1(\widehat{h(x), i(y)}, s(z))) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$. To normalize it, we choose to generate all symbols as output. Then the following clauses (3) and (4) are added into $Prog$: $P_d(d_1\widehat{(x,y,z)}) \leftarrow P_1(\widehat{x}, \widehat{y}, \widehat{z})$. **(3)** and $P_1(\widehat{h(x)}, \widehat{i(y)}, \widehat{s(z)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$. **(4)**. By applying clause (1) and using rule 2, we obtain the critical pair $P_d(d_2\widehat{(x,y,z)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$. **(5)**. This critical pair being already normalized, it is directly added into $Prog$.

We obtain the critical pair $P_d(\widehat{d_1(h(x), i(y), s(z))}) \leftarrow P_1(\widehat{x}, \widehat{y}, \widehat{z})$ by applying clause (3) and rule 1. To normalize it, we choose to generate all symbols as output. It produces clause (3) again, as well as $P_1(\widehat{h(x)}, \widehat{i(y)}, \widehat{s(z)}) \leftarrow P_1(\widehat{x}, \widehat{y}, \widehat{z})$. **(6)**.

Applying clause (3) and using rule 2, we get the critical pair: $P_d(\widehat{d_2(x, y, z)}) \leftarrow P_1(\widehat{x}, \widehat{y}, \widehat{z})$. **(7)** which is already normalized. Thus, it is directly added into $Prog$. Applying clause (5) and using rule 4, we get the critical pair $P_d(\widehat{c(x, y)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y})$. **(8)** which is already normalized. Consequently, it is directly added into $Prog$.

By applying clauses (7) and (4), and using rule 3, we get the critical pair: $P_d(\widehat{d_2(f(h(x)), g(i(y)), z)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$. To normalize it, we choose to generate d_2, f, g as output, and h, i as input. It produces:

$P_d(\widehat{d_2(x, y, z)}) \leftarrow P_2(\widehat{x}, \widehat{y}, \widehat{z}, x', y', z'), P_a(\widehat{x'}), P_a(\widehat{y'}), P_a(\widehat{z'})$. **(9)**

$P_2(\widehat{f(x)}, \widehat{g(y)}, \widehat{z}, x', y', z') \leftarrow P_2(\widehat{x}, \widehat{y}, \widehat{z}, h(x'), i(y'), z')$ **(10')**

$P_2(\widehat{x}, \widehat{y}, \widehat{z}, x, y, z) \leftarrow .$ **(11)**

Now, clause $(10')$ may provide an infinite number of critical pairs. Applying removeCycles makes clause (10') be substituted by $P_2(\widehat{f(x)}, \widehat{g(y)}, \widehat{z}, x', y', z') \leftarrow P_2(\widehat{x}, \widehat{y}, \widehat{z_1}, h(x'), i(y'), z'_1), P_2(\widehat{x_1}, \widehat{y_1}, \widehat{z}, h(x'_1), i(y'_1), z')$ **(10)**.

By applying clauses (7) and (6), and using rule 3, we get the critical pair: $P_d(\widehat{d_2(f(h(x)), g(i(y)), z)}) \leftarrow P_1(\widehat{x}, \widehat{y}, \widehat{z})$. We normalize it as previously. We get $P_d(\widehat{d_2(x, y, z)}) \leftarrow P_2(\widehat{x}, \widehat{y}, \widehat{z}, x', y', z'), P_1(\widehat{x'}, \widehat{y'}, \widehat{z'})$. **(12)** as well as (10), (11) again.

With clauses (9 or 12), (10), and rule 3, we get the convergent critical pairs $P_d(\widehat{d_2(f(f(x)), g(g(y)), z)}) \leftarrow P_2(\widehat{x}, \widehat{y}, \widehat{z_1}, h(h(x')), i(i(y')), z'_1), P_a(\widehat{x'}), P_a(\widehat{y'}), P_a(\widehat{z})$ and $P_d(\widehat{d_2(f(f(x)), g(g(y)), z)}) \leftarrow P_2(\widehat{x}, \widehat{y}, \widehat{z_1}, h(h(x')), i(i(y')), z'_1), P_1(\widehat{x'}, \widehat{y'}, \widehat{z})$.

By applying clauses (9 or 12) and (11), and using rule 3, we get the convergent critical pairs $P_d(\widehat{d_2(f(h(x)), g(i(y)), z)}) \leftarrow P_1(\widehat{x}, \widehat{y}, \widehat{z})$. and $P_d(\widehat{d_2(f(h(x)), g(i(y)), z)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y}), P_a(\widehat{z})$. By applying clauses 9 and 11, and using rule 4, we get the convergent critical pair $P_d(\widehat{c(x, y)}) \leftarrow P_a(\widehat{x}), P_a(\widehat{y})$. Applying clauses 9 and 10, and using rule 4, we get the critical pair: $P_d(\widehat{c(f(x), g(y))}) \leftarrow P_2(\widehat{x}, \widehat{y}, \widehat{z}, h(x'), i(y'), z'), P_a(\widehat{x'}), P_a(\widehat{y'})$. Its normalization gives the clauses: $P_3(\widehat{f(x)}, \widehat{g(y)}) \leftarrow P_2(\widehat{x}, \widehat{y}, \widehat{z}, h(x'), i(y'), z'), P_a(\widehat{x'}), P_a(\widehat{y'})$. **(13)** and $P_d(\widehat{c(x, y)}) \leftarrow P_3(\widehat{x}, \widehat{y})$. **(14)**. Note that the symbols c, f and g have been considered as output parameters.

No more critical pairs are detected and the procedure stops. The resulting program $Prog'$ is composed of clauses (1) to (14). Note that the subset of descendants $d_2(f^k(h^n(a)), g^k(i^n(a)), s^{n-k}(a))$ can be seen (with $p = n - k$) as $d_2(f^k(h^{k+p}(a)), g^k(i^{k+p}(a)), s^p(a))$. The reader can check by himself that $L_{Prog'}(P_d)$ is exactly $R^*(I)$.