Over-approximating Descendants by Synchronized Tree Languages

Yohan Boichut  Jacques Chabin  Pierre Réty

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

Abstract

Over-approximating the descendants (successors) of an initial set of terms by a rewrite system is used in verification. The success of such verification methods depends on the quality of the approximation. To get better approximations, we are going to use non-regular languages. We present a procedure that always terminates and that computes an over-approximation of descendants, using synchronized tree-(tuple) languages expressed by logic programs.

Keywords: rewriting, descendants, tree languages, logic programming.

1 Introduction

Given an initial set of terms $I$, computing the descendants (successors) of $I$ by a rewrite system $R$ is used in the verification domain, for example to check cryptographic protocols or Java programs [2, 7, 8, 9]. 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, we compute an over-approximation $App$ of $R^*(I)$ (i.e. $App \supseteq R^*(I)$), and check that $App \cap Bad = \emptyset$, which ensures that the protocol or the program is correct.

Most often, $I$, $App$ and $Bad$ have been considered as regular tree languages, recognized by finite tree automata. In the general case, $R^*(I)$ is not regular, even if $I$ is. Moreover, the expressiveness of regular languages is poor, and 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) in order to conclude as often as possible [2, 3, 5]. However, in some cases, no regular over-approximation works, whatever the quality of the approximation is [4].

To overcome this theoretical limit, we want 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, since we still 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 over-approximation of descendants using context-free tree languages has been proposed in [13]. This class of languages is quite interesting, however it cannot express relations (or countings) in terms between independent branches, except if there are only unary symbols and constants. For example, let $R = \{ f(x) \rightarrow c(x, x) \}$ and the infinite set $I = \{ f(t) \}$ where $t$ denotes any term composed with the binary symbol $g$ and constant $b$. Then $R^*(I) = I \cup \{ c(t, t) \}$, which is not a context-free language [1, 12].

We want to use another class of languages that has the needed properties, and that can express relations between independent branches: the synchronized tree-(tuple) languages [14, 11], which were finally expressed thanks to logic programs (Horn clauses) [15, 16]. This class has the same properties as context-free tree languages: closure under union, closure under intersection with a regular language, decidability of membership and emptiness. Both
Over-approximating Descendants by Synchronized Tree Languages

include regular languages, however they are different. The example given above is not context-free, but synchronized. The language \( \{s^n(p^n(a))\} \) (where \( s^a \) means that \( s \) occurs \( n \) times vertically) is context-free, but it is not synchronized. \( \{c(s^n(a), p^n(a))\} \) belongs to both classes (note that \( s \) and \( p \) are unary).

In this paper, we propose a procedure that always terminates and that computes an over-approximation of the descendants obtained by a left-linear rewrite system, using synchronized tree-(tuple) languages expressed by logic programs. Note that the left-linearity of rewrite systems (or transducers) is a usual restriction, see \([2, 5, 7, 8, 9]\). Nevertheless, such rewrite systems are still Turing complete \([6]\).

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 using synchronized languages, is explained in Section 3. Finally, in Section 4 our technique is applied on two pertinent examples: an example illustrating a non-regular approximation of a non-regular set of terms, and another one that cannot be handled by any regular approximation.

2 Preliminaries

Consider a finite ranked alphabet \( \Sigma \) and a set of variables \( \text{Var} \). Each symbol \( f \in \Sigma \) has a unique arity, denoted by \( \text{ar}(f) \). The notions of first-order term, position, substitution, are defined as usual. \( T_\Sigma \) denotes the set of ground terms (without variables) over \( \Sigma \). For a term \( t \), \( \text{Var}(t) \) is the set of variables of \( t \), \( \text{Pos}(t) \) is the set of positions of \( t \). For \( p \in \text{Pos}(t) \), \( t(p) \) is the symbol of \( \Sigma \cup \text{Var} \) occurring at position \( p \) in \( t \), and \( t_p \) is the subterm of \( t \) at position \( p \). The term \( t[t']_p \) is obtained from \( t \) by replacing the subterm at position \( p \) by \( t' \). \( \text{PosVar}(t) = \{p \in \text{Pos}(t) \mid t(p) \in \text{Var}\} \), \( \text{PosNonVar}(t) = \{p \in \text{Pos}(t) \mid t(p) \notin \text{Var}\} \). Note that if \( p \in \text{PosNonVar}(t) \), \( t_p = f(t_1, \ldots , t_n) \), and \( i \in \{1, \ldots , n\} \), then \( p.i \) is the position of \( t_i \) in \( t \). For \( p, p' \in \text{Pos}(t) \), \( p < p' \) 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 \( \rho \) s.t. \( \rho(t) = t' \). Let \( \sigma, \sigma' \) be substitutions, \( \sigma \) is more general than \( \sigma' \) (denoted \( \sigma \leq \sigma' \)) if there exists a substitution \( \rho \) s.t. \( \rho.\sigma = \sigma' \).

A rewrite rule is an oriented pair of terms, written \( l \rightarrow r \). We always assume that \( l \) is not a variable, and \( \text{Var}(r) \subseteq \text{Var}(l) \). 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 \( \rightarrow_R \) is defined as follows: \( t \rightarrow_R t' \) if there exist a position \( p \in \text{PosNonVar}(t) \), a rule \( l \rightarrow r \in R \), and a substitution \( \theta \) s.t. \( t_p = \theta(l) \) and \( t' = t[\theta(r)]_p \). \( \rightarrow_R^* \) denotes the reflexive-transitive closure of \( \rightarrow_R \). \( t' \) is a descendant of \( t \) if \( t \rightarrow_R^n t' \). If \( E \) is a set of ground terms, \( R^*(E) \) denotes the set of descendants of elements of \( E \).

In the following, we consider the framework of pure logic programming, and the class of synchronized tree-tuple languages defined by CS-clauses \([15, 16]\). Given a set \textit{Pred} of predicate symbols; \textit{atoms}, \textit{goals}, \textit{bodies} and \textit{Horn-clauses} are defined as usual. Note that both \textit{goals} and \textit{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 \( A_i = P(t_1, \ldots , t_n) \).

\textbf{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 \textit{CS-clause} is a Horn-clause \( H \leftarrow B \) s.t. \( B \) is flat and linear. A \textit{CS-program} \( \text{Prog} \) is a logic
program composed of CS-clauses.

Given a predicate symbol $P$ of arity $n$, the tree-(tuple) language generated by $P$ is $L(P) = \{ t \in (T_T)^n \mid P(t) \in \text{Mod}(\text{Prog}) \}$, where $T_T$ is the set of ground terms over the signature $\Sigma$ and $\text{Mod}(\text{Prog})$ is the least Herbrand model of $\text{Prog}$. $L(P)$ is called Synchronized language.

The following definition describes the different kinds of CS-clauses that can occur.

► **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. A CS-program is normalized if all its clauses are normalized.
- preserving if $\text{Var}(P(t_1, \ldots, t_n)) \subseteq \text{Var}(B)$. A CS-program is preserving if all its clauses are preserving.
- synchronizing if $B$ is composed of only one atom.

► **Example 3.** The CS-clause $P(x, y, z) \leftarrow G(x, y, z)$ is empty, normalized, and preserving ($x$, $y$, $z$ are variables). The CS-clause $P(f(x), y, g(x, z)) \leftarrow G(x, y)$ is normalized and non-preserving. Both clauses are synchronizing.

Given a CS-program, we focus on two kinds of derivations: a classical one based on unification and a rewriting one based on matching and a rewriting process.

► **Definition 4.** Given a logic program $\text{Prog}$ and a sequence of atoms $G$,

- $G$ derives into $G'$ by a resolution step if there exist a clause\(^1\) $H \leftarrow B$ in $\text{Prog}$ and an atom $A \in G$ such that $A$ and $H$ are unifiable by the most general unifier $\sigma$ (then $\sigma(A) = \sigma(H)$) and $G' = \sigma(G)[\sigma(A) \leftarrow \sigma(B)]$. It is written $G \sim_\sigma G'$.
- $G$ rewrites into $G'$ if there exist a clause $H \leftarrow B$ in $\text{Prog}$, an atom $A \in G$, and a substitution $\sigma$, such that $A \equiv \sigma(H)$ ($A$ is not instantiated by $\sigma$) and $G' = G[A \leftarrow \sigma(B)]$. It is written $G \rightarrow_\sigma G'$.

► **Example 5.** Let $\text{Prog} = \{ P(x_1, g(x_2)) \leftarrow P'(x_1, x_2). P(f(x_1), x_2) \leftarrow P''(x_1, x_2) \}$. and consider $G = P(f(x), y)$. Thus, $P(f(x), y) \sim_{\sigma_1} P'(f(x), x_2)$ with $\sigma_1 = [x_1/f(x), y/g(x_2)]$ and $P(f(x), y) \rightarrow_{\sigma_2} P''(x, y)$ with $\sigma_2 = [x_1/x, x_2/y]$.

We consider the transitive closure $\sim^+$ and the reflexive-transitive closure $\rightarrow^*$ of $\sim$. It is well known that resolution is complete.

► **Theorem 6.** Let $A$ be a ground atom. $A \in \text{Mod}(\text{Prog})$ iff $A \sim^*_\text{Prog} \emptyset$.

► **Example 7.** Let $A = P(f(g(a)), g(a), c)$ and $A' = P'(f(g(a)), h(c))$ be two ground atoms. Let $\text{Prog}$ be the CS-program defined by:

$\text{Prog} = \{ P(f(g(x)), y, c) \leftarrow P_1(x), P_2(y). P_1(a) \leftarrow . P_2(g(x)) \leftarrow P_1(x). P'(f(x), u(z)) \leftarrow . \}$

Thus, $A \in \text{Mod}(\text{Prog})$ and $A' \notin \text{Mod}(\text{Prog})$.

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

The following lemma focuses on a preserving property of the relation $\sim_{\sigma}$.

\(^1\) We assume that the clause and $G$ have distinct variables.
Lemma 8. Let $Prog$ be a CS-program, and $G$ be a sequence of atoms. Let $|G'|_\Sigma$ denote the number of occurrences of function-symbols in $G$. If $G$ is linear and $G \sim^* G'$, then $G'$ is also linear and $|G'|_\Sigma \leq |G|_\Sigma$. Consequently, if $G$ is flat and linear, then $G'$ is also flat and linear.

Proof. Let $G = A_1 \ldots A_k$ be a linear sequence of atoms and suppose that $G \sim_\sigma G'$. Then there exist an atom $A'(s_1, \ldots, s_n)$ of $G$ and a CS-clause $A'(t_1, \ldots, t_n) \leftarrow B$ in $Prog$ such that $G' = \sigma(G)[\sigma(A') \leftarrow \sigma(B)]$. As $G$ is linear and $\sigma$ is the most general unifier between $A'(s_1, \ldots, s_n)$ and $A'(t_1, \ldots, t_n)$, $\sigma$ does not instantiate variables from $A^1, \ldots, A^{i-1}, A^{i+1}, \ldots, A^k$. So $G' = A_1^{i-1}, A_i^{i+1}, \ldots, A_k$. $G'$ is not linear only if $\sigma(B)$ is not linear. As $B$ is linear, $\sigma(B)$ is not linear would require that two distinct variables $x_{j_1}, x_{j_2}$ from $B$ are instantiated by two terms containing a same variable $y \in \text{Var}(\sigma(x_{j_1}) \cap \text{Var}(\sigma(x_{j_2})))$. Since $\sigma$ is the most general unifier, $x_{j_1}, x_{j_2}$ are also in $\text{Var}(A'(t_1, \ldots, t_n))$ ($\sigma$ does not instantiate extra variables). Then $y$ occurs at least twice in $A'(s_1, \ldots, s_n)$ (the atom of goal $G$), which is impossible since $G$ is linear. Consequently $G'$ is linear.

By contradiction: to obtain $|G'|_\Sigma > |G|_\Sigma$, we must have in $\sigma(B)$ a duplication of a non-variable subterm of $\sigma((A'(s_1, \ldots, s_n))$ (because $B$ is flat), which is not possible because $B$ and $A'(s_1, \ldots, s_n)$ are linear and $\sigma$ is the most general unifier.

The result trivially extends to the case of several steps $G \sim^* G'$.

Example 9. Let $Prog = \{ P(g(x), f(x)) \leftarrow P_1(x) \}$ and $G = P(g(f(y)), z)$. Then $G \sim G'$ with $G' = P_1(f(y))$, and $G'$ is linear. Moreover, $|G'|_\Sigma \leq |G|_\Sigma$ with $\Sigma = \{ f^1, a^0 \}$.

3 Computing Descendants

Given a CS-program $Prog$ and a left-linear rewrite system $R$, we propose a technique allowing us to compute a CS-program $Prog'$ such that $R^*(\text{Mod}(Prog)) \subseteq \text{Mod}(Prog')$. First of all, a notion of critical pairs is introduced in Section 3.1. Roughly speaking, this notion makes the detection of uncovered rewriting steps possible. Critical pair detection is at the heart of the technique. Thus, in Section 3.2 some restrictions are underlined on CS-programs in order to make the number of critical pairs finite. Moreover, when a CS-program does not fit these restrictions, we have proposed a technique in order to transform such a CS-program into another one of the expected form (REMOVE CYCLES in Fig.1). The detected critical pairs lead to a set of CS-clauses to be added in the current CS-program. However, they may not be in the expected form i.e. normalized CS-clauses. Indeed, one of the restrictions set in Section 3.2 is that the CS-program has to be normalized. So, we propose in Section 3.3 an algorithm providing normalized CS-clauses from non-normalized ones. Finally, in Section 3.4, our main contribution, i.e. the computation of an over-approximating CS-program, is fully described.

3.1 Critical pairs

The notion of critical pair is at the heart of our technique. Indeed, it allows us to add CS-clauses into the current CS-program in order to cover rewriting steps. This notion is described in Definition 10.

Definition 10. Let $Prog$ be a CS-program and $l \rightarrow r$ be a left-linear rewrite rule. Let $x_1, \ldots, x_n$ be distinct variables s.t. $\{x_1, \ldots, x_n\} \cap \text{Var}(l) = \emptyset$. If there are $P$ and $k$ s.t.
Example 11. Let $Prog$ be the normalized and preserving CS-program defined by:

$$Prog = \{ P(c(x), c(x), y) \leftarrow Q(x, y). \quad Q(a, b) \leftarrow. \quad Q(c(x), y) \leftarrow Q(x, y) \}.$$ 

and consider the left-linear rewrite rule: $c(c(x')) \rightarrow h(h(x'))$. Recall that for all goals $G, G'$, the step $G \rightarrow G'$ means that $G \sim_{\sigma} G'$ where $\sigma$ does not instantiate the variables of $G$. Thus $P(c(c(x')), y', z') \sim_{\theta} Q(c(x'), y) \rightarrow Q(x', y)$ where $\theta = [x/c(x'), y'/c(c(x')), z'/y]$. It generates the critical pair $P(h(h(x')), c(c(x'))), y) \leftarrow Q(x', y)$. There are also two other critical pairs: $P(c(c(x'))), h(h(x')), y) \leftarrow Q(x', y)$ and $Q(h(h(x')), y) \leftarrow Q(x', y)$.

However, some of the detected critical pairs are not so critical since they are already covered by the current CS-program. These critical pairs are said to be convergent.

Definition 12. A critical pair $H \leftarrow B$ is said convergent if $H \Rightarrow_{Prog}^{*} B$.

Example 13. The three critical pairs detected in Example 11 are not convergent in $Prog$.

So, here we come to Theorem 14, i.e. the corner stone making our approach sound. Indeed, given a rewrite system $R$ and CS-program $Prog$, if every critical pair that can be detected is convergent, then for any set of terms $I$ such that $I \subseteq Mod(Prog)$, $Mod(Prog)$ is an over-approximation of the set of terms reachable by $R$ from $I$.

Theorem 14. 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. $(A \in Mod(Prog) \land A \Rightarrow_{R}^{*} A') \implies A' \in Mod(Prog)$.

---

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

3 We have $\theta(P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n)) \rightarrow G$, and since $Prog$ is preserving $\text{Var}(\theta(P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n))) \subseteq \text{Var}(G)$. Since $\text{Var}(l) \subseteq \text{Var}(l)$ we have $\text{Var}(\theta(P(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n))) \subseteq \text{Var}(G)$. 

---

Figure 1 An overview of our contribution
Proof. Let $A \in Mod(Prog)$ s.t. $A \rightarrow_{t \rightarrow r} A'$. Then $A|_i = C[\sigma(l)]$ for some $i \in \mathbb{N}$ and $A' = A|i \leftarrow C[\sigma(r)]$.

Since resolution is complete, $A \rightarrow^* \emptyset$. Since $Prog$ is normalized and preserving, resolution consumes symbols in $C$ one by one, thus $G_0 = A \rightarrow^* G_k \rightarrow^* \emptyset$ and there exists 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 during the step $G_k \rightarrow G_{k+1}$. 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 $\sigma'$ 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 $P(x_1, \ldots, l, \ldots, x_n) \sim^* \emptyset$ and $\theta \leq \sigma'$. This derivation can be decomposed into: $P(x_1, \ldots, l, \ldots, x_n) \sim^* B' \sim^* B$ resolution is applied only on non-flat atoms, and we have $\gamma_2 \gamma_1(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* B$. Note that $\gamma_3(B) \rightarrow^* G$ and recall that $\gamma_3 \gamma_2 \gamma_1 = \theta_2 \eta_2 \theta_1 = \theta$. Then $\theta(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* \emptyset$. Since $\theta \leq \sigma'$ we get $P(t_1, \ldots, \sigma(r), \ldots, t_n) = \sigma'(P(x_1, \ldots, r, \ldots, x_n)) \rightarrow^* \emptyset$. Therefore $A' \rightarrow^* G_k[A'' \leftarrow P(t_1, \ldots, \sigma(r), \ldots, t_n)] \rightarrow^* \emptyset$, hence $A' \in Mod(Prog)$.

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

If $Progs$ is not normalized, Theorem 14 does not hold.

Example 15. Let $Progs = \{P(c(f(a))) \leftarrow \}$ and $R = \{f(a) \rightarrow b\}$. All critical pairs are convergent since there is no critical pair. $P(c(f(a))) \in Mod(Prog)$ and $P(c(f(a))) \rightarrow_R P(c(b))$. However there is no resolution step issued from $P(c(b))$, then $P(c(b)) \notin Mod(Prog)$.

If $Progs$ is not preserving, Theorem 14 does not hold.

Example 16. Let $Progs = \{P(c(x), c(x), y) \leftarrow Q(y). (a) \leftarrow \}$, and $R = \{f(b) \rightarrow b\}$. All critical pairs are convergent since there is no critical pair.

$P(c(f(b)), c(f(b)), a) \rightarrow_{progs} Q(a) \rightarrow_{progs} \emptyset$, then $P(c(f(b)), c(f(b)), a) \notin Mod(Prog)$. On the other hand, $P(c(f(b)), c(f(b)), a) \rightarrow_R P(c(b), c(f(b)), a)$. However there is no resolution step issued from $P(c(b), c(f(b)), a)$, then $P(c(b), c(f(b)), a) \notin Mod(Prog)$.

Unfortunately, for a given finite CS-program, there may be infinitely many critical pairs. In the following section, this problem is illustrated and some syntactical conditions on CS-programs is underlined in order to avoid this critical situation.

3.2 Ensuring finitely many critical pairs

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

Example 17. Let $\Sigma = \{f^{1,2}, c^{1,4}, d^{1,4}, s^{1,4}, a^{0}\}$ and $f(c(x), y) \rightarrow d(y)$ be a rewrite rule, and $Progs = \{P_d(f(x, y)) \leftarrow P_1(x, y). P_1(x, s(y)) \leftarrow P_1(x, y). P_1(c(x), y) \leftarrow P_2(x, y). P_2(a, a) \leftarrow \}$. Then $P_0(f(c(x), y)) \rightarrow P_1(c(x), y) \sim^* c(y/s) P_1(c(x), y) \sim^* d(y/s) \cdots P_1(c(x), y) \rightarrow P_2(x, y)$. 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 $P_0(d(s^n(y))) \leftarrow P_2(x, y)$.

---

4 Since $\emptyset$ is flat, a flat goal can always be reached, i.e. in some cases $G = \emptyset$. 
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 CS-program has finitely many critical pairs.

**Definition 18.** $Prog$ is empty-recursive if there exist a predicate $P$ and distinct variables $x_1, \ldots, x_n$ s.t. $P(x_1, \ldots, x_n) \iff \ast \ A_1, \ldots, P(x'_1, \ldots, x'_n), \ldots, A_k$ where $x'_1, \ldots, x'_n$ are variables and there exist $i, j$ s.t. $x'_i = \sigma(x_j)$ and $\sigma(x_j)$ is not a variable and $x'_j \in \text{Var}(\sigma(x_j))$.

**Example 19.** Let $Prog$ be the CS-program defined as follows:

$$Prog = \{ P(x', s(y')) \leftarrow P(x', y'). P(a, b) \leftarrow \}. $$

From $P(x, y)$, one can obtained the following derivation: $P(x, y) \iff s(x, y') \iff P(x', y')$. 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 CS-program is sufficient to ensure the finiteness of the number of critical pairs.

**Lemma 20.** Let $Prog$ be a normalized CS-program.

If $Prog$ is not empty-recursive, then the number of critical pairs is finite.

**Remark.** Note that the CS-program of Example 17 is normalized and has infinitely many critical pairs, however it is empty-recursive because $P_1(x, y) \iff s(x, y') \iff P_1(x', y')$.

**Proof.** By contrapositive. Let us suppose there exist infinitely many critical pairs. So there exist $P_1$ and infinitely many derivations of the form $(i) : P_1(x_1, \ldots, x_{k-1}, l, x_{k+1}, \ldots, x_n) \iff P_1(x', y')$. $G' \iff G$ (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 $(ii) : P_2(t_1, \ldots, t_p) \iff G'' \iff G''$ (with $k > 0$) included in some derivation of $(i)$, strictly before the last step, such that:

1. $G''$ and $G''$ are flat.
2. $\sigma$ 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 $P_2(t'_1, \ldots, t'_n)$. Otherwise we could not have an infinite number of $\sigma$ necessary to obtain infinitely many critical pairs.
3. At least one term $t'_j$ ($j \in \{1, \ldots, p\}$) is not a variable (only the last step of the initial derivation produces a flat goal $G$). As we use a CS-clause in each derivation step, we can assume that $t'_j$ is a term among $t_1, \ldots, t_n$ and moreover that $t'_j = t_j$. 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. So, for each variable $x$ occurring in the non-variable terms, 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 $y = t_k$.

If in the (ii) derivation we replace all non-variable terms by new variables, we obtain a new derivation $(iii) : P_2(x_1, \ldots, x_p) \iff G''_1, P_2(x'_1, \ldots, x'_p), G''_2$ and there exists $i, k$ such that $\sigma(x_i) = x'_i$ (at least one non-variable term in the (ii) derivation), $\sigma(x_k) = t_k$, and $x'_i$ is a variable of $t_k$. We conclude that $Prog$ is empty-recursive.

Deciding the empty-recursiveness of a CS-program seems to be a difficult problem (undecidable?). Nevertheless, we propose a sufficient syntactic condition to ensure that a CS-program is not empty-recursive.
Definition 21. The clause \( P(t_1, \ldots, t_n) \leftarrow A_1, \ldots, Q(\ldots), \ldots, A_m \) is pseudo-empty over \( Q \) if there exist \( i, j \) s.t.
\( t_i \) is a variable,
\( t_j \) is not a variable,
and \( \exists x \in \text{Var}(t_j), x \neq t_i \wedge \{x, t_i\} \subseteq \text{Var}(Q(\ldots)) \).

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

The clause \( H \leftarrow B \) is pseudo-empty if there exists some \( Q \) s.t. \( H \leftarrow B \) is pseudo-empty
over \( Q \).

The CS-clause \( P(t_1, \ldots, t_n) \leftarrow A_1, \ldots, Q(x_1, \ldots, x_k), \ldots, A_m \) is empty over \( Q \) if for all \( x_i \) \( (\exists j, t_j = x_i \) or \( x_i \not\in \text{Var}(P(t_1, \ldots, t_n)) \)).

Example 22. The CS-clause \( P(x, f(x), z) \leftarrow Q(x, z) \) 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 23. Using Definition 21, let us define two relations over predicate symbols.
\( P_1 \geq_{\text{Prog}} P_2 \) if there exists in \( \text{Prog} \) 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 \( \geq_{\text{Prog}} \) is denoted by \( \geq^*_{\text{Prog}} \).
\( P_1 >_{\text{Prog}} P_2 \) if there exist in \( \text{Prog} \) predicates \( P_1, P_2 \) s.t. \( P_1 \cong_{\text{Prog}} P_1' \) and \( P_2 \cong_{\text{Prog}} P_2' \),
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 \( >_{\text{Prog}} \) is denoted by \( >^*_{\text{Prog}} \).

Example 24. Let \( \Sigma = \{f^1, h^1, a^0\} \) Let \( \text{Prog} \) be the following CS-program:
\[ \text{Prog} = \{ P(x, h(y), f(z)) \leftarrow Q(x, z), R(y). \quad Q(x, g(y, z)) \leftarrow P(x, y, z). \quad R(a) \leftarrow. \quad Q(a, a) \leftarrow. \} \]
One has \( P >^*_{\text{Prog}} Q \) and \( Q >^*_{\text{Prog}} P \). Thus, \( >_{\text{Prog}} \) 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 25. If \( >_{\text{Prog}} \) is not cyclic, then \( \text{Prog} \) is not empty-recursive, consequently the number of critical pairs is finite.

Proof. By contrapositive. Let us suppose that \( \text{Prog} \) is empty recursive. So there exist \( P \)
and distinct variables \( x_1, \ldots, x_n \) s.t. \( P(x_1, \ldots, x_n) \sim_{\sigma} A_1, \ldots, P(x'_1, \ldots, x'_n), \ldots, A_k \) 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 \( x'_j \in \text{Var}(\sigma(x_j)). \) We can extract from the previous derivation the following derivation which has \( p \) steps (\( p \geq 1 \)).
\[ P(x_1, \ldots, x_n) = Q^0(x_1, \ldots, x_n) \sim_{\alpha_1} B_1^1 \ldots Q^1(x_1, \ldots, x_n) \sim_{\alpha_2} \ldots B_{k_1}^1 \ldots Q^k(x_1', \ldots, x_n') = P(x'_1, \ldots, x'_n). \]

For each \( k, \alpha_k(\alpha_{k-1}(\ldots(\alpha_1(x_1)))) \) is a variable of \( Q^k(x_1', \ldots, x_n') \) and \( \alpha_k(\alpha_{k-1}(\ldots(\alpha_1(x_1)))) \) is either a variable of \( Q^k(x_1', \ldots, x_n') \) or a non-variable term containing a variable of \( Q^k(x_1', \ldots, x_n') \).

Each derivation step issued from \( Q^k \) uses either a clause pseudo-empty over \( Q^{k+1} \) and
we deduce \( Q^k \geq_{\text{Prog}} Q^{k+1} \), or an empty clause over \( Q^{k+1} \) and we deduce \( Q^k \cong_{\text{Prog}} Q^{k+1} \).
At least one step uses a pseudo-empty clause otherwise no variable from \( x_1, \ldots, x_n \) would
be instantiated by a non-variable term containing at least one variable in \( x'_1, \ldots, x'_n \). We conclude that \( P = Q^0 \circ p_1 Q^1 \circ p_2 Q^2 \ldots Q^{p-1} \circ p_k Q^p = P \) with each \( p_k \) is \( >_{\text{Prog}} \) or \( \geq_{\text{Prog}} \)
and there exists \( k \) such that \( p_k \) is \( >_{\text{Prog}} \). Therefore \( P >^*_{\text{Prog}} P \), so \( >_{\text{Prog}} \) is cyclic.
So, if a CS-program $Prog$ does not involve $>_\text{prog}$ to be cyclic, then all is fine. Otherwise, we have to transform $Prog$ into another CS-program $Prog'$ such as $>_\text{prog}'$ is not cyclic and $\text{Mod}(Prog) \subseteq \text{Mod}(Prog')$.

The transformation is based on the following observation. If $>_\text{prog}$ is cyclic, there is at least one pseudo-empty clause over a given predicate that participates in a cycle. Note that this remark can be checked in Example 24 where $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ is a pseudo-empty clause over $Q$ involving the cycle. To remove cycles, we transform some pseudo-empty clauses into clauses that are not pseudo-empty anymore. It boils down to unsynchronize some variables. The process is mainly described in Definition 28. Definitions 26 and 27 are intermediary definitions involved in Definition 28.

Definition 26 (simplify). Let $H \leftarrow A_1, \ldots, A_n$ be a CS-clause, and for each $i$, let us write $A_i = P_i(\ldots)$.

If there exists $P_i$ s.t. $L(P_i) = \emptyset$ then simplify($H \leftarrow A_1, \ldots, A_n$) is the empty set, otherwise it is the set that contains only the clause $H \leftarrow B_1, \ldots, B_m$ such that:

- $\{B_i \mid 0 \leq i \leq m\} \subseteq \{A_i \mid 0 \leq i \leq n\}$ and
- $\forall i \in \{1, \ldots, n\}, \neg(\exists j, B_j = A_i) \Leftrightarrow \text{Var}(A_i) \cap \text{Var}(H) = \emptyset$.

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

Definition 27 (unSync). Let $P(t_1, \ldots, t_n) \leftarrow B$ be a pseudo-empty CS-clause.

$\text{unSync}(P(t_1, \ldots, t_n) \leftarrow B) = \text{simplify}(P(t_1, \ldots, t_n) \leftarrow \sigma_0(B), \sigma_1(B))$ where $\sigma_0$, $\sigma_1$ are substitutions built as follows:

$$
\sigma_0(x) = \begin{cases} 
  x & \text{if } \exists i, t_i = x \\
  \text{a fresh variable otherwise}
\end{cases}
\sigma_1(x) = \begin{cases} 
  x & \text{if } \exists i, t_i \not\in \text{Var} \land x \in \text{Var}(t_i) \\
  \neg(\exists j, t_j = x) & \text{a fresh variable otherwise}
\end{cases}
$$

Definition 28 (removeCycles). Let $Prog$ be a CS-program.

removeCycles($Prog$) = $\begin{cases} 
  Prog & \text{if } >_\text{prog} \text{ is not cyclic} \\
  \text{removeCycles}([\text{unSync}(H \leftarrow B)] \cup Prog') & \text{otherwise}
\end{cases}$

where $H \leftarrow B$ is a pseudo-empty clause involved in a cycle and $Prog' = Prog \setminus \{H \leftarrow B\}$.

Example 29. Let $Prog$ be the CS-program of Example 24. Since $Prog$ is cyclic, let us compute removeCycles($Prog$). The pseudo-empty CS-clause $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ is involved in the cycle. Consequently, unSync is applied on it. According to Definition 27, one obtains $\sigma_0$ and $\sigma_1$ 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 CS-clause $P(x, h(y), f(z)) \leftarrow Q(x, x_2), R(x_1), Q(x_3, z), R(y)$. Note that according to Definition 27, simplify has to be applied on the CS-clause above-mentioned. Following Definitions 26 and 28, one has to remove $P(x, h(y), f(z)) \leftarrow Q(x, z), R(y)$ from $Prog$ and to add $P(x, h(y), f(z)) \leftarrow Q(x, x_2), Q(x_3, z), R(y)$ instead. Note that the atom $R(x_1)$ has been removed using simplify. Note also that there is no cycle anymore.

Lemma 30 describes that our transformation preserves at least and may extend the initial least Herbrand Model.

Lemma 30. Let $Prog$ be a CS-program and $Prog' = \text{removeCycles}(Prog)$.

Then $>_\text{prog}'$ is not cyclic, and $\text{Mod}(Prog) \subseteq \text{Mod}(Prog')$. Moreover, if $Prog$ is normalized and preserving, then so is $Prog'$.

Proof. Proof is given in Appendix A.
At this point, given a CS-program \( \text{Prog} \), if \( >_{\text{prog}} \) is not cyclic then the number of critical 

pairs is finite. Otherwise, we transform \( \text{Prog} \) into another CS-program \( \text{Prog}' \) in such a way 

that \( >_{\text{prog}}' \) is not cyclic and \( \text{Mod}(\text{Prog}) \subseteq \text{Mod}(\text{Prog}') \). Since \( \text{Prog}' \) is not cyclic, the 

finiteness of the number of critical pairs is ensured.

3.3 Normalizing critical pairs

In Section 3.1, we have defined the notion of critical pair and we have shown in Theorem 

14 that this notion is useful for a matter of rewriting closure. Moreover, as mentioned at 

the very beginning of Section 3, non-convergent critical pairs correspond to the CS-clauses 

that we would like to add in the current CS-program. Unfortunately, these CS-clauses are 

not necessarily in the expected form (normalized).

Definition 34 describes the normalization process that transforms a non-normalized 

CS-clause \( P(f(g(x)), b) \) into \( P'(x) \). We want to generate a set of normalized CS-clauses 

covering at least the same Herbrand model. The following set of CS-clauses \( \{ P(f(x_1), b) \leftrightarrow 

P_{\text{new}}(x_1), P_{\text{new}}(g(x_1)) \leftrightarrow P'(x_1) \} \) is a good candidate with \( P_{\text{new}} \) a new predicate symbol.

Definition 31 introduces tools for manipulating parameters of predicates (tuple of terms). 

Definition 32 formalizes a way for cutting a clause head, at depth 1. An example is given 

after Definition 34.

Definition 31. A tree-tuple \( (t_1, \ldots, t_n) \) is normalized if for all \( i, t_i \) is a variable or contains 

only one function-symbol.

We define tuple concatenation by \( (t_1, \ldots, t_n) \cdot (s_1, \ldots, s_k) = (t_1, \ldots, t_n, s_1, \ldots, s_k) \).

The arity of the tuple \( (t_1, \ldots, t_n) \) is \( \text{ar}(t_1, \ldots, t_n) = n \).

Definition 32. Consider a tree-tuple \( \vec{t} = (t_1, \ldots, t_n) \). We define:

\( \vec{t}^{\text{cut}} = (t_1^{\text{cut}}, \ldots, t_n^{\text{cut}}) \), where 

\( t_i^{\text{cut}} = \begin{cases} x_i' & \text{if } t_i \text{ is a variable} \\ t_i & \text{if } t_i \text{ is a constant} \\ t_i(e)(x_1', \ldots, x_{\text{ar}(t_i(e)))} & \text{otherwise} \end{cases} \)

and variables \( x_i' \) are new variables that do not occur in \( \vec{t} \).

for each \( i \), \( \text{Var}(t_i^{\text{cut}}) \) is the (possibly empty) tuple composed of the variables of \( t_i^{\text{cut}} \) (taken 
in the left-right order).

\( \text{Var}(\vec{t}^{\text{cut}}) = \text{Var}(t_1^{\text{cut}}) \cdots \text{Var}(t_n^{\text{cut}}) \) (concatenation of tuples).

for each \( i \), \( t_i^{\text{rest}} \) is the tree-tuple \( t_i^{\text{rest}} = \begin{cases} (t_i) & \text{if } t_i \text{ is a variable} \\ (t_i)[1, \ldots, t_i|\text{ar}(t_i(e))] & \text{otherwise} \end{cases} \)

\( \vec{t}^{\text{rest}} = (t_1^{\text{rest}}, \ldots, t_n^{\text{rest}}) \) (concatenation of tuples).

Example 33. Let \( \vec{t} \) be a tree-tuple such that \( \vec{t} = (x_1, x_2, g(x_3, h(x_1)), h(x_4), b) \) where 

\( x_i \)'s are variables. Thus, 

\( \vec{t}^{\text{cut}} = (y_1, y_2, g(y_3, y_4), h(y_5), b) \) with \( y_i \)'s new variables;

\( \text{Var}(\vec{t}^{\text{cut}}) = (y_1, y_2, y_3, y_4, y_5) \);

\( \vec{t}^{\text{rest}} = (x_1, x_2, x_3, h(x_1), x_4) \).

Note that \( \vec{t}^{\text{cut}} \) is normalized, \( \text{Var}(\vec{t}^{\text{cut}}) \) is linear, \( \text{Var}(\vec{t}^{\text{cut}}) \) and \( \vec{t}^{\text{rest}} \) have the same arity.

Notation: \( \text{card}(S) \) denotes the number of elements of the finite set \( S \).
Definition 34 (norm). Let $Prog$ be a normalized CS-program.
Let $Pred$ be the set of predicate symbols of $Prog$, and for each positive integer $i$, let $Pred_i = \{ P \in Pred \mid \text{ar}(P) = i \}$ where ar means arity.
Let arity-limit and predicate-limit be positive integers s.t. $\forall P \in Pred$, $\text{arity}(P) \leq \text{arity-limit}$, and $\forall i \in \{1, \ldots, \text{arity-limit}\}$, $\text{card}(Pred_i) \leq \text{predicate-limit}$. Let $H \leftarrow B$ be a CS-clause.
Function $\text{norm}_{Prog}(H \leftarrow B)$

Res = $Prog$

If $H \leftarrow B$ is normalized
then $Res = Res \cup \{ H \leftarrow B \}$ (a)
else if $H \rightarrow_{Res} A$ by a synchronizing and non-empty clause
then (note that $A$ is an atom) $Res = \text{norm}_{Res}(A \leftarrow B)$ (b)
else let us write $H = P^t$
If $\text{ar}(\text{Var}(t^{cut})) \leq \text{arity-limit}$
then let $c'$ be the clause $P^t \leftarrow P'(\text{Var}(t^{cut}))$
where $P'$ is a new or an existing predicate symbol$^5$
$Res = \text{norm}_{Res \cup \{ c' \}}(P'(t^{rest}) \leftarrow B)$ (c)
else choose tuples $v_{t1} \rightarrow t1, \ldots, v_{tk} \rightarrow tk$ and tuples $tt_1, \ldots, tt_k$ s.t.
$\forall v_{t1} \ldots v_{tk} \equiv \text{Var}(t^{cut})$ and $tt_1 \ldots tt_k = t^{rest}$,
and for all $j$, $\text{ar}(v_{tj}) = \text{ar}(tt_j)$ and $\text{ar}(v_{tj}) \leq \text{arity-limit}$
let $c'$ be the clause $P^t \leftarrow P_j(v_{t1}, \ldots, P_k(v_{tk})$
where $P_1, \ldots, P_k$ are new or existing predicate symbols$^6$
$Res = Res \cup \{ c' \}$
For $j=1$ to $k$ do $Res = \text{norm}_{Res}(P_j(tt_j) \leftarrow B)$ EndFor (d)
EndIf
EndIf
EndIf
return Res

Example 35. Consider the CS-program $Prog =$

$\{ P_0(f(x)) \leftarrow P_1(x), P_1(a) \leftarrow, P_0(u(x)) \leftarrow P_2(x), P_2(f(x)) \leftarrow P_3(x), P_3(v(x, x)) \leftarrow P_1(x) \}$

Let $\text{arity-limit} = 1$ and $\text{predicate-limit} = 5$. Let $P_2(u(f(v(x, x)))) \leftarrow P_3(x)$ be a CS-clause to normalize. According to Definition 34, we are not in case (a) nor in (b), we are in case (c). Then, according to Definition 32, $u(f(v(x, x)))^{cut} = u(x_1)$ with $x_1$ a new variable. Since for now the number of predicates with arity 1 is equal to $4 < \text{predicate-limit}$, a new predicate $P_4$ can be created and then one has to add the CS-clause $P_2(u(x_1)) \leftarrow P_4(x_1)$. Then we have to solve the recursive call $\text{norm}_{Prog \cup \{ P_2(u(x_1)) \leftarrow P_4(x_1) \}}(P_4(f(v(x, x))) \leftarrow P_3(x))$.
The same process is applied except for the creation of a new predicate, because $\text{predicate-limit}$ would be exceeded. Consequently, no new predicate with arity 1 can be generated. One has to choose an existing one. Let us try with $P_3$. So, the CS-clause $P_4(f(x_2)) \leftarrow P_3(x_2)$ is added into $Prog$ (because $f(v(x, x))^{cut} = f(x_2)$) and then, norm is called with the parameter $P_3(v(x, x)) \leftarrow P_3(x)$. Finally, $P_3(v(x, x)) \leftarrow P_3(x)$ is also added into $Prog$ since this clause is already normalized. To summarize, the normalization of the CS-clause

$^5$ If $\text{card}(\text{Pred}_{\text{ar}(v_{tj}^{cut})}(Res)) < \text{predicate-limit}$, then $P'$ is new, otherwise $P'$ is arbitrarily chosen in $\text{Pred}_{\text{ar}(v_{tj}^{cut})}(Res)$.

$^6$ For all $j$, $P'_j$ is new if $\text{card}(\text{Pred}_{\text{ar}(v_{tj})}(Res)) + j - 1 < \text{predicate-limit}$. 
\[ P_2(u(f(v(x,x)))) \leftarrow P_3(x) \] has produced three new clauses, which are \( P_2(u(x_1)) \leftarrow P_3(x_1) \), \( P_2(f(x_2)) \leftarrow P_3(x_2) \) and \( P_3(v(x,x)) \leftarrow P_3(x) \).

Obviously, termination of \( \text{norm} \) is guaranteed according to Lemma 36.

\begin{lemma}
Function \( \text{norm} \) always terminates.
\end{lemma}

\begin{proof}
Consider a run of \( \text{norm}_{\text{Prog}}(H \leftarrow B) \), and any recursive call \( \text{norm}_{\text{Prog}'}(H' \leftarrow B') \). We can see that \( |H'|_\Sigma < |H|_\Sigma \). Consequently a normalized clause is necessarily reached, and there is no recursive call in this case.
\end{proof}

Given a normalized CS-program \( \text{Prog} \), Theorem 37 raises two important points:
1. given a non-normalized clause \( H \leftarrow B \), one obtains \( H \rightarrow^{\text{norm}} \text{Prog}(H \leftarrow B) \), and
2. adding the CS-clauses provided by \( \text{norm} \) into \( \text{Prog} \) may increase the least Herbrand model of \( \text{Prog} \).

\begin{theorem}
Let \( c \) be a critical pair in \( \text{Prog} \). Then \( c \) is convergent in \( \text{norm}_{\text{Prog}}(c) \).
Moreover for any CS-clause \( c' \), we have \( \text{Mod}(\text{Prog} \cup \{c'\}) \subseteq \text{Mod(\text{norm}_{\text{Prog}}(c'))} \).
\end{theorem}

\begin{proof}
The second item of the theorem is a consequence of the first item. Let us now prove the first item. Let \( c = (H \leftarrow B) \) and let us prove that \( H \rightarrow^{\text{ind-hyp}} \text{Prog} \). The proof is by induction on recursive calls to Function \( \text{norm} \) (we write \( \text{ind-hyp} \) for “induction hypothesis”). We consider items (a), (b),... in Definition 34 :

(a) From Lemma 30.
(b) We have \( H \rightarrow A \rightarrow^{*_{\text{ind-hyp}}} B \).
(c) \( H = P(t) \rightarrow c' P'(t_{\text{rest}}) \rightarrow^{*_{\text{ind-hyp}}} B \).
(d) \( H = P(t) \rightarrow c' (P'_1(t_1),...,P'_k(t_k)) \rightarrow^{*_{\text{ind-hyp}}} (B,...,B) \) (up to variable renamings).
\end{proof}

\subsection{3.4 Completion}

In Sections 3.1 and 3.3, we have described how to detect critical pairs and how to convert them into normalized clauses. Moreover, in a given finite CS-program the number of critical pairs is finite as shown in Section 3.2. Definition 38 explains precisely our technique for computing over-approximation using a CS-program completion.

\begin{definition}[comp]
Let \( R \) be a left-linear rewrite system, and \( \text{Prog} \) be a finite and normalized CS-program s.t.
\begin{itemize}
  \item \( \text{Prog} \) is not cyclic (otherwise apply \( \text{removeCycles} \) to remove cycles),
  \item \( \forall P \in \text{Pred}, \text{arity}(P) \leq \text{arity-limit} \),
  \item \( \forall i \in \{1,\ldots,\text{arity-limit}\}, \text{card}(\text{Pred}_i) \leq \text{predicate-limit} \),
\end{itemize}
where \( \text{card}(\text{Pred}_i) \) is the number of predicate symbols of arity \( i \).

Function \( \text{comp}_R(\text{Prog}) \)
\begin{algorithmic}
  \While{there exists a non-convergent critical pair \( H \leftarrow B \)}
    \State \( \text{Prog} = \text{removeCycles}(\text{norm}_{\text{Prog}}(H \leftarrow B)) \)
  \EndWhile
  \Return \text{Prog}
\end{algorithmic}

Theorem 39 and Corollary 40 illustrate that our technique leads to a finite CS-program whose least Herbrand model over-approximates the descendants obtained by a left-linear rewrite system \( R \).
Theorem 39. Function $\text{comp}$ always terminates, and all critical pairs are convergent in $\text{comp}_R(\text{Prog})$. Moreover $\text{Mod}(\text{Prog}) \subseteq \text{Mod}(\text{comp}_R(\text{Prog}))$.

Proof. Proof is given in Appendix B.

Moreover, thanks to Theorem 14, $\text{Mod}(\text{comp}_R(\text{Prog}))$ is closed under rewriting by $R$. Then:

Corollary 40. If in addition $\text{Prog}$ is preserving, $R^*(\text{Mod}(\text{Prog})) \subseteq \text{Mod}(\text{comp}_R(\text{Prog}))$.

4 Examples

In this section, our technique is applied on several examples. In Examples 41, 42 and 43, $I$ is the initial set of terms and $R$ is the rewrite system. Moreover, initially, we define a CS-program $\text{Prog}$ that generates $I$.

Example 41. In this example, we define $\Sigma$ as follows: $\Sigma = \{c^2, a^0\}$. Let $I$ be the set of terms $I = \{f(t) \mid t \in T_\Sigma\}$. Let $R$ be the rewrite system $R = \{f(x) \rightarrow b(x, x)\}$. Obviously, one can easily guess that $R^*(I) = \{b(t,t) \mid t \in T_\Sigma\} \cup I$. Note that $R^*(I)$ is not a regular, nor a context-free language [1, 12].

Initially, $\text{Prog} = \{P_0(f(x)) \leftarrow P_1(x).\ P_1(c(x, y)) \leftarrow P_1(x), P_1(y).\ P_1(a) \leftarrow .\}$. Using our approach, the critical pair $P_0(b(x,x)) \leftarrow P_1(x)$ is detected. This critical pair is already normalized, then it is immediately added into $\text{Prog}$. Thus, there is no more critical pair and the procedure stops. Note that we get exactly the set of descendants, i.e. $L(P_0) = R^*(I)$. So, given $t, t' \in T_\Sigma$ such that $t \neq t'$, one can show that $b(t,t') \notin R^*(I)$.

The example right above shows that non-context-free descendants can be handled in a conclusive manner with our approach. Such example cannot be handled by [13] in an exact way, because they use context-free languages. Actually, the classes of languages covered by our approach and theirs are in some sense orthogonal. However, the examples below shows that our approach can also be relevant for other problems.

Example 42.

Let $I$ be the set of terms $I = \{f(a,a)\}$, and $R$ be the rewrite system $R = \{f(x) \rightarrow u(f(v(x), w(y)))\}$. Intuitively, the exact set of descendants is $R^*(I) = \{u^n(f(v^a(a), w^a(a))) \mid n \in \mathbb{N}\}$. We define $\text{Prog} = \{P_0(f(x,y)) \leftarrow P_1(x), P_1(y).\ P_1(a) \leftarrow .\}$. We choose predicate-limit = 4 and arity-limit = 2.

First, the following critical pair is detected: $P_0(u(f(v(x), w(y)))) \leftarrow P_1(x), P_1(y)$. According to Definition 34, the normalization of this critical pair produces three new CS-clauses: $P_0(u(x)) \leftarrow P_2(x)$, $P_2(f(x,y)) \leftarrow P_3(x,y)$ and $P_3(v(x), w(y)) \leftarrow P_1(x), P_1(y)$. Adding these three CS-clauses into $\text{Prog}$ produces the new critical pair $P_2(u(f(v(x), w(y)))) \leftarrow P_3(x,y)$. This critical pair can be normalized without exceeding predicate-limit. So, we add: $P_2(u(x)) \leftarrow P_3(x)$. $P_2(f(x,y)) \leftarrow P_3(x,y)$. and $P_3(v(x), w(y)) \leftarrow P_3(x,y)$.

Once again, a new critical pair has been introduced: $P_4(u(f(v(x), w(y)))) \leftarrow P_3(x,y)$. Note that, from now, we are not allowed to introduce any new predicate of arity 1. Let us proceed the normalization of $P_4(u(f(v(x), w(y)))) \leftarrow P_3(x,y)$ step by step. We choose to reuse the predicate $P_4$. Thus, we first generate the following CS-clause: $P_4(u(x)) \leftarrow P_4(x)$. So, we have to normalize now $P_4(f(v(x), w(y))) \leftarrow P_3(x,y)$. Note that $P_4(f(v(x), w(y))) \rightarrow_{\text{Prog}} P_3(x,y)$. Consequently, the CS-clause $P_4(x,y) \leftarrow P_3(x,y)$ is added into $\text{Prog}$.

Note that there is no critical pair anymore.

To summarize, we obtain the final CS-program $\text{Prog}_f$ composed of the following CS-clauses:
Over-approximating Descendants by Synchronized Tree Languages

For applying Function Example 43, Example 43 shows that this limitation can now be overcome. The authors propose an example that cannot be handled by regular approximations. Example 43 proves that we have presented a procedure that always terminates and that computes an over-approximation of the set of descendants, expressed by a synchronized tree language. This is the first attempt using synchronized tree languages. It could be improved or extended:

- In Definition 34, when predicate-limit is reached (in items (c) and (d)), an (several in item (d)) existing predicate of the right arity is chosen arbitrarily and re-used, instead of creating a new one. Of course, if there are several existing predicates of the right arity, the achieved choice affects the quality of the approximation. When using regular languages [7], a similar difficulty happens: to make the procedure terminate, it is sometimes necessary to choose and re-use an existing state instead of creating a new one. Some ideas have been proposed to make this choice in a smart way [10]. We are going to extend these ideas in order to improve the choice of existing predicates.

- A similar problem arises when arity-limit is reached (item (d)): a tuple is divided into several smaller tuples in an arbitrary way, and there may be several possibilities, which may affect the quality of the approximation.

- To compute descendants, we have used synchronized tree languages, whereas context-free languages have been used in [13]. Each approach has advantages and drawbacks. Therefore, it would be interesting to mix the two approaches to get the advantages of both.

References


Appendix: Proofs

A Proof of Lemma 30

In order to prove this result, we need to use intermediary lemmas.

Lemma 44. Let $\text{Prog} \cup \{cl\}$ be a CS-program. Then $\text{Mod}(\text{Prog} \cup \{cl\}) = \text{Mod}(\text{Prog} \cup \{\text{simplify}(cl)\})$.

Proof. Obvious.

Lemma 45. Let cl be a CS-clause. Then $\text{unSync}(cl)$ is a CS-clause that is not pseudo-empty. Moreover, if cl is normalized and preserving, then so is $\text{unSync}(cl)$.

Proof. To write the proof, as well as the proof of Lemma 46, we need to define precisely what the fresh variables are. Moreover the proof goes easier if every variable is renamed by $\sigma_0$ and by $\sigma_1$, which is not the case in Definition 27. This is why we consider another expression of Definition 27:

Function $\text{UnSync}(P(t_1,\ldots,t_n) \leftarrow B)$

- let us write $X = \text{Var}(P(t_1,\ldots,t_n) \leftarrow B) = \{x_1,\ldots,x_k\} = X_0 \uplus X_1 \uplus X_2$ where $X_0 = \{t_i \mid t_i$ is a variable$\}$
$X_1 = \{x \mid \exists t_i, t_i$ is not a variable and $x \in \text{Var}(t_i)\}\setminus X_0$
$X_2 = \text{Var}(B)\setminus\text{Var}(P(t_1,\ldots,t_n))$

- we consider sets of variables $Y = \{y_1,\ldots,y_k\} \uplus \{y'_1,\ldots,y'_k\} \uplus \{y''_1,\ldots,y''_k\}$
$Z = \{z_1,\ldots,z_k\} \uplus \{z'_1,\ldots,z'_k\} \uplus \{z''_1,\ldots,z''_k\}$

- let $\sigma_0$ and $\sigma_1$ defined on $X$ by $\sigma_0(x_i) = y_i$ if $x_i \in X_0$
$\sigma_0(x_i) = y'_i$ if $x_i \in X_1$ and $\sigma_1(x_i) = z'_i$ if $x_i \in X_1$
$\sigma_0(x_i) = y''_i$ if $x_i \in X_2$

- let $\sigma$ defined on $X_0 \uplus X_1$ by $\sigma(x_i) = \begin{cases} \sigma_0(x_i) & \text{if } x_i \in X_0 \\ \sigma_1(x_i) & \text{if } x_i \in X_1 \end{cases}$

- return simplify($\sigma(P(t_1,\ldots,t_n)) \leftarrow \sigma_0(B),\sigma_1(B)$)

Note that the images of $\sigma_0$ and $\sigma_1$ are disjoint. Moreover $\sigma_0$ (resp. $\sigma_1$) is an injection going from $X$ to $Y$ (resp. $Z$). Therefore the body of $\text{unSync}(cl)$ is linear and flat, hence cl is a CS-clause.

Let $x_i \in X_0$ and $x_j \in X_1$, and let us write $cl = (H \leftarrow B)$, and $\text{unSync}(cl) = (H' \leftarrow B')$. Recall that $\sigma(x_i) = y_i$ and $\sigma(x_j) = z'_j$, and $H' = \sigma(H)$. However $B' = \sigma_0(B),\sigma_1(B)$, and $\text{Var}(\sigma_0(B)) \subseteq Y$, and $\text{Var}(\sigma_1(B)) \subseteq Z$. Consequently $y_i$ and $z'_j$ cannot occur in the same atom of $H'$, hence $\text{unSync}(cl)$ is not pseudo-empty.

Now, suppose that cl is normalized and preserving. Since $\sigma$, $\sigma_0$, $\sigma_1$ are substitutions, $\text{unSync}(cl)$ is normalized. Any variable vv occurring in $H'$ is equal to $\sigma_0(x_i)$ or $\sigma_1(x_i)$ for some $x_i \in X$. Necessarily $x_i$ occurs in $B$, then $vv$ occurs in $\sigma_0(B)$ or $\sigma_1(B)$, hence in $B'$.  

Lemma 46. Let $\text{Prog} \cup \{cl\}$ be a CS-program. Then $\text{Mod}(\text{Prog} \cup \{cl\}) \subseteq \text{Mod}(\text{Prog} \cup \{\text{unSync}(cl)\})$.

Proof. Suppose $A \not\sim^* B$. The proof is by induction on the length of the derivation. Let $cl = (H \leftarrow B)$ and $cl' = \text{unSync}(cl) = (H' \leftarrow B')$, and suppose that the first step of
the derivation uses cl. Then \( \delta(A) \rightarrow_{cl} G \rightarrow^*_{\text{Prog} \cup \{cl\}} \emptyset \). There exists a substitution \( \theta \) s.t. \( \delta(A) = \theta(H) \) and \( G = \theta(B) \). Then \( \theta(H) \rightarrow_{cl} \theta(B) \).

Note that \( \sigma_0 \) and \( \sigma_1 \) going from \( X \) to theirs images, are bijective. \( \sigma \) going from \( X_0 \cup X_1 \) to its image is also bijective. Let \( \delta \leq \sigma^{-1} \), \( \sigma^{-1} \) theirs converse mappings. Note that \( \sigma_0^{-1}, \sigma_1^{-1} \) are defined on disjoint sets, and \( (\sigma_0^{-1} \cup \sigma_1^{-1})|_{\text{Var}(H')} = \sigma^{-1} \). Let \( \gamma = \sigma_0^{-1} \cup \sigma_1^{-1} \). Then \( H = \gamma(H') \) and the first part of \( \gamma(B') \) is equal to \( B \), as well as the second part of \( \gamma(B') \). Therefore \( \delta(A) = \theta(H) = \theta(\gamma(H')) \) and \( G = \theta(B) = \theta(f_{\text{sp}}((\gamma(B')))) \). Let \( fp \) and \( sp \) mean first part and second part respectively. Consequently \( \delta(A) \rightarrow_{cl'} G,G \rightarrow^*_{\text{Prog} \cup \{cl\}} \emptyset \).

By induction hypothesis, we get \( \delta(A) \rightarrow_{cl'} G,G \rightarrow^*_{\text{Prog} \cup \{cl'\}} \emptyset \). Thus \( A \leadsto_{\text{Prog} \cup \{cl'\}} \emptyset \). ◀

Now, let us prove Lemma 30. We want to prove that given a CS-program \( \text{Prog} \) and \( \text{Prog}' = \text{removeCycles}(\text{Prog}) \),

1. \( \rightarrow_{\text{Prog}'} \) is not cyclic, and \( \text{Mod}(\text{Prog}) \subseteq \text{Mod}(\text{Prog}') \)
2. if \( \text{Prog} \) is normalized and preserving, then so is \( \text{Prog}' \).

Proof. Because of the loop condition, if \( \text{removeCycles} \) terminates, \( \rightarrow \) is not cyclic. In the loop, one pseudo-empty clause is removed and replaced by a non-pseudo-empty one (from Lemma 45). Thus, the number of pseudo-empty clauses decreases, until \( \rightarrow \) is not cyclic (which necessarily happens because if there are no pseudo-empty clauses anymore, \( \rightarrow \) is not cyclic), and \( \text{removeCycles} \) terminates. Thanks to Lemma 46, \( \text{Mod}(\text{Prog}) \subseteq \text{Mod}(\text{Prog}') \).

On the other hand, thanks to Lemma 45, \( \text{Prog}' \) is normalized and preserving if \( \text{Prog} \) is. ◀

B Proof of Theorem 39

In order to prove this theorem, we need to use intermediary lemmas.

▶ Lemma 47. Let \( \text{Prog}' \) be a normalized CS-program. Then each clause \( H \leftarrow A_1, \ldots, A_n \) in \( \text{removeCycles}(\text{Prog}') \) satisfies \( n \leq \text{arity-limit} \ast \text{max-arity}(\Sigma) \).

Proof. When applying \( \text{removeCycles} \), \( \text{simplify} \) is applied, then each \( A_i \) contains at least one variable of \( H \). Moreover the body is linear. Then \( n \) is less than or equal to the number of variables of \( H \), which is normalized. ◀

▶ Lemma 48. There are finitely many normalized tree-tuples of arity not greater than \( \text{arity-limit} \) (up to a variable renaming).

Proof. Obvious. ◀

▶ Lemma 49. There exists \( k \in \mathbb{N} \) s.t. at all step of Function \( \text{comp} \), the number of clauses\(^7\) in \( \text{Prog} \) is not greater than \( k \).

Proof. Because of Function \( \text{norm} \), the number of predicate symbols in \( \text{Prog} \) is necessarily less than \( \text{arity-limit} \ast \text{arity-limit} \). Since clauses in \( \text{Prog} \) are always normalized and from Lemmas 47 and 48, we get the result. ◀

Thus, let us prove that Function \( \text{comp} \) always terminates, and all critical pairs are convergent in \( \text{comp}_R(\text{Prog}) \). Moreover \( \text{Mod}(\text{Prog}) \subseteq \text{Mod}(\text{comp}_R(\text{Prog})) \).

\(^7\) Considering that two clauses identical up to a variable renaming, are equal.
Proof. When running $\text{norm}_{\text{Prog}}(H \leftarrow B)$, either new clauses are added, or not (when the added clauses already exist in $\text{Prog}$). From Lemma 49 the number of clauses is bounded, then there exists a step $k$ from which no new clause is added. Moreover, at any step, $>_{\text{Prog}}$ is acyclic. Therefore, from Lemma 25, at step $k$, the number of existing critical pairs is finite. However, some of them may be non-convergent. Then, for all (finitely many) non-convergent critical pairs, $\text{norm}$ is run (without adding any clause), which makes them convergent (from Theorem 37). Then all critical pairs are convergent, and $\text{comp}$ terminates. Moreover, thanks to Theorem 37 and Lemma 30, we get $\text{Mod}(\text{Prog}) \subseteq \text{Mod}(\text{comp}_{\text{R}}(\text{Prog}))$. ◀