

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

Rapport de Recherche

Weak Inclusion for recursive XML Types (full version)

Joshua Amavi, Jacques Chabin, Pierre Réty
LIFO, Université d'Orléans

Rapport n° RR-2012-02

Weak Inclusion for recursive XML Types (full version of [4])

Joshua Amavi Jacques Chabin Pierre Réty

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

Abstract. Considering that the *unranked* tree languages $L(G)$ and $L(G')$ are those defined by given *possibly-recursive XML types* G and G' , this paper proposes a method to verify whether $L(G)$ is “approximatively” included in $L(G')$. The approximation consists in weakening the father-children relationships. Experimental results are discussed, showing the efficiency of our method in many situations.

Keywords: XML type, regular unranked-tree grammar, approximative inclusion.

1 Introduction

In database area, an important problem is schema evolution, particularly when considering XML types. XML is also used for exchanging data on the web. In this setting, we want to compare XML types in a loose way. To do it, we address the more general problem of approximative comparison of unranked-tree languages defined by regular grammars.

Example 1. Suppose an application where we want to replace an XML type G by a new type G' (eg., a web service composition where a service replaces another, each of them being associated to its own XML message type). We want to analyse whether the XML messages supported by G' contains (in an approximate way) those supported by G . XML types are regular tree grammars where we just consider the structural part of the XML documents, disregarding data attached to leaves. Thus, to define leaves we consider rules of the form $A \rightarrow a[\epsilon]$. Suppose that G and G' contain the following rules:

$F \rightarrow \text{firstName}[\epsilon]$, $L \rightarrow \text{lastName}[\epsilon]$, $T \rightarrow \text{title}[\epsilon]$ and $Y \rightarrow \text{year}[\epsilon]$.

P defines a publication, and B is the bibliography.

In G : $P \rightarrow \text{publi}[(F.L)^+.T.B^?]$, $B \rightarrow \text{biblio}[P^+]$.

In G' : $P \rightarrow \text{publi}[A^*.Pa]$, $A \rightarrow \text{author}[F.L]$, $Pa \rightarrow \text{paper}[T.Y.B^?]$, $B \rightarrow \text{biblio}[P^+]$

We want to know whether messages valid with respect to G can be accepted (in an approximate way) by G' . Notice that G accepts trees such as t in Figure 1 that are not valid with respect to schema G' but that represent the same kind of information G' deals with. Indeed, in G' , the same information would be organized as the tree t' in Figure 1. □

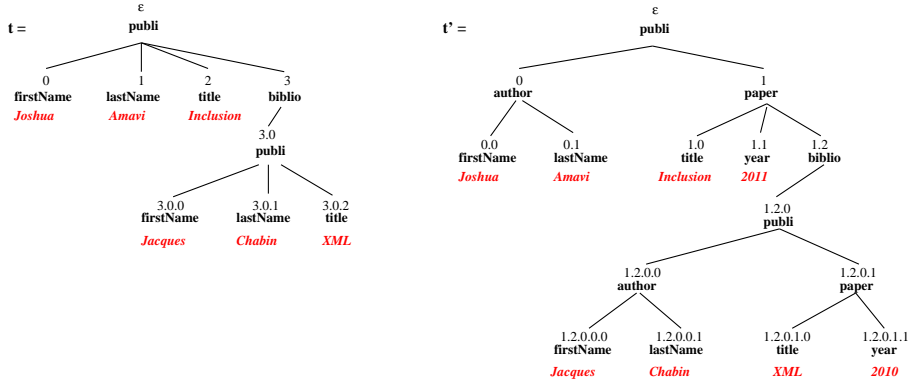


Fig. 1. Examples of trees t and t' valid with respect to G and G' , respectively.

The approximative criterion for comparing trees that is commonly used consists in weakening the father-children relationships (*i.e.*, they are implicitly reflected in the data tree as only ancestor-descendant). In this paper, we consider this criterion in the context of tree languages. We denote this relation *weak inclusion* to avoid confusion with the usual inclusion of languages (*i.e.*, the inclusion of a set of trees in another one).

Given two types G and G' , we call $L(G)$ and $L(G')$ the sets (also called languages) of XML documents valid with respect to G and G' , respectively. Our paper proposes a method for deciding whether $L(G)$ is *weakly included* in $L(G')$, in order to know if the substitution of G by G' can be envisaged. The unranked-tree language $L(G)$ is *weakly included* in $L(G')$ if for each tree $t \in L(G)$ there is a tree $t' \in L(G')$ such that t is weakly included in t' . Intuitively, t is weakly included in t' (denoted $t \triangleleft t'$) if we can obtain t by removing nodes from t' (a removed node is replaced by its children, if any). For instance, in Figure 1, t can be obtained by removing nodes *author*, *paper*, *year* from t' , *i.e.* we have $t \triangleleft t'$.

To decide whether $L(G)$ is weakly included in $L(G')$, we consider the set of trees $WI(L(G')) = \{t \mid \exists t' \in L(G'), t \triangleleft t'\}$. Note that¹ $L(G)$ is weakly included in $L(G')$ iff $L(G) \subseteq WI(L(G'))$.

To compute $WI(L(G'))$, we have already proposed [3] a direct and simple approach using regular unranked-tree grammars (hedge grammars), assuming that $L(G')$ is bounded in depth, *i.e.* G' is not recursive. Given a hedge grammar G' , the idea consists in replacing each non-terminal occurring in a right-hand-side of a production rule, by itself or its children. For example, if G' contains the rules $A \rightarrow a[B]$, $B \rightarrow b[C]$, $C \rightarrow c[\epsilon]$, we get the grammar $G'' = \{C \rightarrow c[\epsilon], B \rightarrow b[C|\epsilon], A \rightarrow a[B|(C|\epsilon)]\}$. This grammar generates $WI(L(G'))$, because in each regular expression, whenever we have a non-terminal X ($X \in \{A, B, C\}$)

¹ If $L(G)$ is weakly included in $L(G')$, for $t \in L(G)$ there exists $t' \in L(G')$ s.t. $t \triangleleft t'$. Then $t \in WI(L(G'))$, hence $L(G) \subseteq WI(L(G'))$.

Conversely if $L(G) \subseteq WI(L(G'))$, for $t \in L(G)$ we have $t \in WI(L(G'))$. Thus there exists $t' \in L(G')$ s.t. $t \triangleleft t'$. Therefore $L(G)$ is weakly included in $L(G')$.

that generates x ($x \in \{a, b, c\}$), we can also directly generate the children of x (instead of x), and more generally the successors of x .

Unfortunately, it does not work if G' is recursive. Consider $G'_1 = \{A \rightarrow a[B.(A|\epsilon).C], B \rightarrow b[\epsilon], C \rightarrow c[\epsilon]\}$. If we work as previously, we get the rule $A \rightarrow a[B.(A|(B.(A|\epsilon).C)|\epsilon).C]$. However, a new occurrence of A has appeared in the rhs, and if we replace it again, this process does not terminate. If we stop at some step, the resulting grammar does not generate $WI(L(G'))$. In this very simple example, it is easy to see that $G''_1 = \{A \rightarrow a[B^*.(A|\epsilon).C^*], B \rightarrow b[\epsilon], C \rightarrow c[\epsilon]\}$ generates $WI(L(G'_1))$. But if we now consider $G'_2 = \{A \rightarrow a[B.(A.A|\epsilon).C], B \rightarrow b[\epsilon], C \rightarrow c[\epsilon]\}$, then in $WI(L(G'_2))$, a may be the left-sibling of b (which was not possible in $L(G'_2)$ nor in $WI(L(G'_1))$). Actually, $WI(L(G'_2))$ can be generated by the grammar $G''_2 = \{A \rightarrow a[(A|B|C)^*], B \rightarrow b[\epsilon], C \rightarrow c[\epsilon]\}$. In other words, the recursive case is much more difficult.

In this paper, we address the general case: some symbols may be recursive, and some others may not. Given an arbitrary regular unranked-tree grammar G' , we present a direct approach that computes a regular grammar, denoted $WI(G')$, that generates the language $WI(L(G'))$. To do it, terminal-symbols are divided into 3 categories: non-recursive, 1-recursive, 2-recursive. For $n > 2$, n -recursivity is equivalent to 2-recursivity. Surprisingly, the most difficult situation is 1-recursivity. We prove that our algorithm for computing $WI(G')$ is correct and complete. An implementation has been done in Java, and experiments are presented.

Consequently, for arbitrary regular grammars G and G' , checking that $L(G)$ is weakly included into $L(G')$, i.e. checking that $L(G) \subseteq WI(L(G'))$, is equivalent to check that $L(G) \subseteq L(WI(G'))$, which is decidable since G and $WI(G')$ are regular grammars.

Paper organisation: Section 2 gives some theoretical background. Section 3 presents how to compute $WI(G)$ for a given possibly-recursive regular grammar G , while Section 4 analyses some experimental results. Related work and other possible methods are addressed in Section 5. Proofs are in the appendix.

2 Preliminaries

An XML document is an *unranked tree*, defined in the usual way as a mapping t from a set of positions $Pos(t)$ to an alphabet Σ . The set of the trees over Σ is denoted by T_Σ . For $v \in Pos(t)$, $t(v)$ is the label of t at the position v , and $t|_v$ denotes the sub-tree of t at position v . *Positions* are sequences of integers in \mathbb{N}^* and $Pos(t)$ satisfies: $\forall u, i, j (j \geq 0, u.j \in Pos(t), 0 \leq i \leq j) \Rightarrow u.i \in Pos(t)$ (char “.” denotes the concatenation). The *size* of t (denoted $|t|$) is the cardinal of $Pos(t)$. As usual, ϵ denotes the empty sequence of integers, i.e. the root position. t, t' will denote trees.

Figure 1 illustrates trees with positions and labels: we have, for instance, $t(1) = lastName$ and $t'(1) = paper$. The sub-tree $t'|_{1.2}$ is the one whose root is *biblio*.

Definition 1. Position comparison: Let $p, q \in Pos(t)$. Position p is an *ancestor* of q (denoted $p < q$) if there is a non-empty sequence of integers r such that $q = p.r$. Position p is *to the left of* q (denoted $p <_l q$) if there are sequences of integers u, v, w , and $i, j \in \mathbb{N}$ such that $p = u.i.v$, $q = u.j.w$, and $i < j$. Position p is *parallel to* q (denoted $p \parallel q$) if $\neg(p < q) \wedge \neg(q < p)$. \square

Definition 2. Resulting tree after node deletion: For a tree t' and a non-empty position q of t' , let us note $Rem_q(t') = t$ the tree t obtained from t' by removing the node at position q (a removed node is replaced by its children, if any). We have:

1. $t(\epsilon) = t'(\epsilon)$,
2. $\forall p \in Pos(t')$ such that $p < q$: $t(p) = t'(p)$,
3. $\forall p \in Pos(t')$ such that $p <_l q$: $t|_p = t'|_p$,
4. Let $q.0, q.1, \dots, q.n \in Pos(t')$ be the positions of the children of position q , if q has no child, let $n = -1$. Now suppose $q = s.k$ where $s \in \mathbb{N}^*$ and $k \in \mathbb{N}$. We have:

- $t|_{s.(k+n+i)} = t'|_{s.(k+i)}$ for all i such that $i > 0$ and $s.(k+i) \in Pos(t')$ (the siblings located to the right of q shift),
- $t|_{s.(k+i)} = t'|_{s.k.i}$ for all i such that $0 \leq i \leq n$ (the children go up). \square

Definition 3. Weak inclusion for unranked trees: The tree t is *weakly included in* t' (denoted $t \triangleleft t'$) if there exists a series of positions $q_1 \dots q_n$ such that $t = Rem_{q_n}(\dots Rem_{q_1}(t'))$. \square

Example 2.

In Figure 1, $t = Rem_0(Rem_1(Rem_{1.1}(Rem_{1.2.0.0}(Rem_{1.2.0.1}(Rem_{1.2.0.1.1}(t'))))))$, then $t \triangleleft t'$. Notice that for each node of t , there is a node in t' with the same label, and this mapping preserves vertical order and left-right order. However the tree $t_1 = paper(biblio, year)$ is not weakly included in t' since *biblio* should appear to the right of *year*. \square

Definition 4. Regular Tree Grammar: A *regular tree grammar* (RTG) (also called *hedge grammar*) is a 4-tuple $G = (NT, \Sigma, S, P)$, where NT is a finite set of *non-terminal symbols*; Σ is a finite set of *terminal symbols*; S is a set of *start symbols*, where $S \subseteq NT$ and P is a finite set of *production rules* of the form $X \rightarrow a[R]$, where $X \in NT$, $a \in \Sigma$, and R is a regular expression over NT . We recall that the set of regular expressions over $NT = \{A_1, \dots, A_n\}$ is inductively defined by: $R ::= \epsilon \mid A_i \mid R|R \mid R.R \mid R^+ \mid R^* \mid R^? \mid (R)$ \square

Grammar in Normal Form: As usual, in this paper, we only consider regular tree grammars such that (i) every non-terminal generates at least one tree containing only terminal symbols and (ii) distinct production rules have distinct left-hand-sides (*i.e.*, tree grammars *in normal form* [13]).

Thus, given an RTG $G = (NT, \Sigma, S, P)$, for each $A \in NT$ there exists in P exactly one rule of the form $A \rightarrow a[E]$, *i.e.* whose left-hand-side is A . \square

Example 3. The grammar $G_0 = (NT_0, \Sigma, S, P_0)$, where $NT_0 = \{X, A, B\}$, $\Sigma = \{f, a, c\}$, $S = \{X\}$, and $P_0 = \{X \rightarrow f[A.B], A \rightarrow a[\epsilon], B \rightarrow a[\epsilon], A \rightarrow c[\epsilon]\}$, is not in normal form. The conversion of G_0 into normal form gives the sets $NT_1 = \{X, A, B, C\}$ and $P_1 = \{X \rightarrow f[(A|C).B], A \rightarrow a[\epsilon], B \rightarrow a[\epsilon], C \rightarrow c[\epsilon]\}$.

Definition 5. Let $G = (NT, \Sigma, S, P)$ be an RTG (in normal form). Consider a non-terminal $A \in NT$, and let $A \rightarrow a[E]$ be the unique production of P whose left-hand-side is A .

$L^w(E)$ denotes the set of words (over non-terminals) generated by E .

The set $L_G(A)$ of trees generated by A is defined recursively by:

$$L_G(A) = \{a(t_1, \dots, t_n) \mid \exists u \in L^w(E), u = A_1 \dots A_n, \forall i, t_i \in L_G(A_i)\}$$

The language $L(G)$ generated by G is: $L(G) = \{t \in T_\Sigma \mid \exists A \in S, t \in L_G(A)\}$.

A *nt-tree* is a tree whose labels are non-terminals. The set $L_G^{nt}(A)$ of nt-trees generated by A is defined recursively by:

$$L_G^{nt}(A) = \{A(t_1, \dots, t_n) \mid \exists u \in L^w(E), u = A_1 \dots A_n, \forall i (t_i \in L_G^{nt}(A_i) \vee t_i = A_i)\}$$

3 Weak Inclusion for possibly-recursive Tree Grammars

First, we need to compute the recursivity types of non-terminals. Intuitively, the non-terminal A of a grammar G is *2-recursive* if there exists $t \in L_G^{nt}(A)$ and A occurs in t at (at least) two non-empty positions $p, q \in Pos(t)$ s.t. $p \parallel q$. A is *1-recursive* if A is not 2-recursive, and A occurs in some $t \in L_G^{nt}(A)$ at a non-empty position. A is *not recursive*, if A is neither 2-recursive nor 1-recursive.

Example 4. Consider the grammar G of Example 1. P is 2-recursive since P may generate B , and B may generate the tree $biblio(P, P)$. B is also 2-recursive. On the other hand F, L, T, Y are not recursive. No non-terminal of G is 1-recursive.

Definition 6. Let $G = (NT, \Sigma, S, P)$ be an RTG in normal form. For a regular expression E , $NT(E)$ denotes the set of non-terminals occurring in E .

- We define the relation $>$ over non-terminals by:
 $A > B$ if $\exists A \rightarrow a[E] \in P$ s.t. $B \in NT(E)$.
- We define $>$ over multisets² of non-terminals, whose size is at most 2, by:
 - $\{A\} > \{B\}$ if $A > B$,
 - $\{A, B\} > \{C, D\}$ if $A = C$ and $B > D$,
 - $\{A\} > \{C, D\}$ if there exists a production $A \rightarrow a[E]$ in G and a word $u \in L(E)$ of the form $u = u_1 C u_2 D u_3$.

Remark 1. To check whether $\{A\} > \{C, D\}$, i.e. $\exists u \in L(E), u = u_1 C u_2 D u_3$, we can use the recursive function "in" defined by $in(C, D, E) =$

- if $E = E_1 | E_2$, return $in(C, D, E_1) \vee in(C, D, E_2)$,
- if $E = E_1 . E_2$, return $(C \in NT(E_1) \wedge D \in NT(E_2)) \vee (C \in NT(E_2) \wedge D \in NT(E_1)) \vee in(C, D, E_1) \vee in(C, D, E_2)$,

² Since we consider multisets, note that $\{A, B\} = \{B, A\}$ and $\{C, D\} = \{D, C\}$.

- if $E = E_1^*$ or $E = E_1^+$, return $(C \in NT(E_1)) \wedge (D \in NT(E_1))$,
- if $E = E_1^?$, return $in(C, D, E_1)$,
- if E is a non-terminal or $E = \epsilon$, return *false*.

This function terminates since recursive calls are always on regular expressions smaller than E . The runtime is $O(|E|)$, where $|E|$ is the size of E .

Definition 7. Let $>^+$ be the transitive closure of $>$

- The non-terminal A is 2-recursive iff $\{A\} >^+ \{A, A\}$.
- A is 1-recursive iff $A >^+ A$ and A is not 2-recursive.
- A is not recursive iff A is neither 2-recursive nor 1-recursive.

Remark 2. The transitive closure of $>$ can be computed using Warshall algorithm. If there are n non-terminals in G , there are $p = n + \frac{n \cdot (n+1)}{2}$ multisets of size at most 2. Then a boolean matrix $p \times p$ can represent $>$, consequently the runtime for computing $>^+$ is $O(p^3) = O(n^6)$, which is polynomial.

Example 5. Using grammar G of Example 1, we have $\{P\} > \{B\} > \{P, P\}$. Therefore P is 2-recursive.

We have $\neg(\{F\} >^+ \{F, F\})$ and $\neg(F >^+ F)$, therefore F is not recursive.

Now, to define an RTG that generates $WI(L(G))$, we need additional notions.

Definition 8. Let $G = (NT, \Sigma, S, P)$ be an RTG in normal form.

- \equiv is the relation over non-terminals defined by $A \equiv B$ if $A >^* B \wedge B >^* A$, where $>^*$ denotes the reflexive-transitive closure of $>$.
- Note that \equiv is an equivalence relation, and if $A \equiv B$ then A and B have the same recursivity type. \hat{A} will denote the equivalence class of A .
- $Succ(A)$ is the set of non-terminals s.t. $Succ(A) = \{X \in NT \mid A >^* X\}$.
- For a set $Q = \{A_1, \dots, A_n\}$ of non-terminals, $Succ(Q) = Succ(A_1) \cup \dots \cup Succ(A_n)$.
- $Left(A)$ is the set of non-terminals defined by $Left(A) = \{X \in NT \mid \exists B, C \in \hat{A}, \exists B \rightarrow b[E] \in P, \exists u \in L^w(E), u = u_1 X u_2 C u_3\}$.
- Similarly, $Right(A)$ is the set of non-terminals defined by $Right(A) = \{X \in NT \mid \exists B, C \in \hat{A}, \exists B \rightarrow b[E] \in P, \exists u \in L^w(E), u = u_1 C u_2 X u_3\}$.
- $RE(A)$ is the regular expression E , assuming $A \rightarrow a[E]$ is the production rule of G whose left-hand-side is A .
- $\widehat{RE}(A) = RE(A) | RE(B_1) | \dots | RE(B_n)$ where $\hat{A} = \{A, B_1, \dots, B_n\}$.

Example 6. With the grammar G' of Example 1, we have :

- $P \equiv B$, because $P > Pa > B$ and $B > P$.
- $\hat{P} = \{P, Pa, B\}$.
- $Succ(A) = \{A, F, L\}$.
- $Left(P)$ is defined using non-terminals equivalent (\equiv) to P , i.e. B, P, Pa , and grammar G' , which contains rules (among others):
 $B \rightarrow \text{biblio}[P^+]$, $P \rightarrow \text{publi}[A^*.Pa]$, $Pa \rightarrow \text{paper}[T.Y.B^?]$.
 P^+ may generate $P.P$, therefore $Left(P) = \{P\} \cup \{A\} \cup \{T, Y\} = \{P, A, T, Y\}$.

- $RE(P) = A^*.Pa$
- $\widehat{RE}(P) = RE(P)|RE(Pa)|RE(B) = (A^*.Pa)|(T.Y.B^2)|P^+$.

Lemma 1. *Let A, B be non-terminals.*

- *If $A \equiv B$ then $Succ(A) = Succ(B)$, $Left(A) = Left(B)$, $Right(A) = Right(B)$, $\widehat{RE}(A) = \widehat{RE}(B)$.*
- *If A, B are not recursive, then $A \neq B$ implies $A \not\equiv B$, therefore $\widehat{A} = \{A\}$ and $\widehat{B} = \{B\}$, i.e. equivalence classes of non-recursive non-terminals are singletons.*

Proof. The first part is obvious.

$(A >^+ A) \implies (A >^+ A \wedge \neg(\{A\} >^+ \{A, A\})) \vee (\{A\} >^+ \{A, A\})$ which implies that A is (1 or 2)-recursive. Consequently, if A is not recursive, then $A \not>^+ A$. Now, if A, B are not recursive, $A \neq B$ and $A \equiv B$, then $A >^+ B \wedge B >^+ A$, therefore $A >^+ A$, which is impossible as shown above. \square

To take all cases into account, the following definition is a bit intricate. To give intuition, consider the following very simple situations:

- If the initial grammar G contains production rules $A \rightarrow a[B]$, $B \rightarrow b[\epsilon]$ (here A and B are not recursive), we replace these rules by $A \rightarrow a[B|\epsilon]$, $B \rightarrow b[\epsilon]$ to generate $WI(L(G))$. Intuitively, b may be generated or removed (replaced by ϵ). See Example 7 below for a more general situation.
- If G contains $A \rightarrow a[(A.A.B)|\epsilon]$, $B \rightarrow b[\epsilon]$ (here A is 2-recursive and B is not recursive), we replace these rules by $A \rightarrow a[(A|B)^*]$, $B \rightarrow b[\epsilon]$ to generate $WI(L(G))$. Actually the regular expression $(A|B)^*$ generates all words composed of elements of $Succ(A)$. See Example 8 for more intuition.
- The 1-recursive case is more complicated and is illustrated by Examples 9 and 10.

Definition 9. For each non-terminal A , we recursively define a regular expression $Ch(A)$ (Ch for children). Here, any set of non-terminals, like $\{A_1, \dots, A_n\}$, is also considered as being the regular expression $(A_1 | \dots | A_n)$.

- if A is 2-recursive, $Ch(A) = (Succ(A))^*$
- if A is 1-rec, $Ch(A) = (Succ(Left(A)))^* . Ch_A^{rex}(\widehat{RE}(A)) . (Succ(Right(A)))^*$
- if A is not recursive, $Ch(A) = Ch_A^{rex}(RE(A))$

and $Ch_A^{rex}(E)$ is the regular expression obtained from E by replacing each non-terminal B occurring in E by $Ch_{\widehat{A}}(B)$, where

$$Ch_{\widehat{A}}(B) = \begin{cases} \widehat{B}|\epsilon & \text{if } B \text{ is 1-recursive and } B \in \widehat{A} \\ Ch(B) & \text{if } (B \text{ is 2 recursive}) \text{ or } (B \text{ is 1-recursive and } B \notin \widehat{A}) \\ B|Ch(B) & \text{if } B \text{ is not recursive} \end{cases}$$

By convention $Ch_A^{rex}(\epsilon) = \epsilon$ and $Ch(\epsilon) = \epsilon$.

Algorithm. *Input:* let $G = (NT, T, S, P)$ be a regular grammar in normal form. *Output:* grammar $G' = (NT, T, S, P')$ obtained from G by replacing each production $A \rightarrow a[E]$ of G by $A \rightarrow a[Ch(A)]$.

Theorem 1. *The computation of Ch always terminate, and $L(G') = WI(L(G))$.*

The proof is given in the appendix. Let us now consider several examples to give more intuition about the algorithm and show various situations.

Example 7. Consider grammar $G = \{A \rightarrow a[B], B \rightarrow b[C], C \rightarrow c[\epsilon]\}$.

A is the start symbol. Note that A, B, C are not recursive.

$$Ch(C) = Ch_{\widehat{C}}^{rex}(RE(C)) = Ch_{\widehat{C}}^{rex}(\epsilon) = \epsilon.$$

$$Ch(B) = Ch_{\widehat{B}}^{rex}(RE(B)) = Ch_{\widehat{B}}^{rex}(C) = Ch_{\widehat{B}}(C) = C|Ch(C) = C|\epsilon.$$

$$Ch(A) = Ch_{\widehat{A}}^{rex}(RE(A)) = Ch_{\widehat{A}}^{rex}(B) = Ch_{\widehat{A}}(B) = B|Ch(B) = B|(C|\epsilon).$$

Thus, we get the grammar $G' = \{A \rightarrow a[B|(C|\epsilon)], B \rightarrow b[C|\epsilon], C \rightarrow c[\epsilon]\}$ that generates $WI(L(G))$ indeed. In this particular case, where no non-terminal is recursive, we get the same grammar as in our previous work [3], though the algorithm was formalized in a different way.

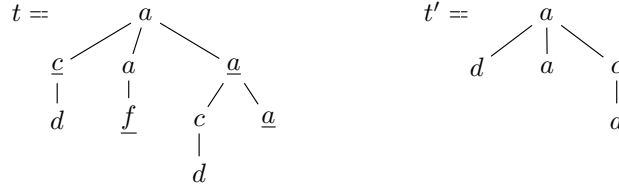
Example 8. Consider grammar G that contains the rules:

$$A \rightarrow a[(C.A.A^?)|F|\epsilon], C \rightarrow c[D], D \rightarrow d[\epsilon], F \rightarrow f[\epsilon]$$

A is 2-recursive; C, D, F are not recursive. $Ch(D) = Ch(F) = \epsilon$. $Ch(C) = D|\epsilon$. $Succ(A) = \{A, C, D, F\}$. Considered as a regular expression, $Succ(A) = A|C|D|F$. Therefore $Ch(A) = (A|C|D|F)^*$. We get the grammar G' :

$$A \rightarrow a[(A|C|D|F)^*], C \rightarrow c[D|\epsilon], D \rightarrow d[\epsilon], F \rightarrow f[\epsilon]$$

The tree t below is generated by G . By removing underlined symbols, we get $t' \triangleleft t$, and t' is generated by G' indeed. Note that a is a left-sibling of c in t' , which is impossible in a tree generated by G .



Example 9. Consider grammar G that contains the rules (A is the start symbol):

$$A \rightarrow a[(B.C.A^?.H)|F], B \rightarrow b[\epsilon], C \rightarrow c[D], D \rightarrow d[\epsilon], H \rightarrow h[\epsilon], F \rightarrow f[\epsilon]$$

A is 1-recursive; B, C, D, H, F are not recursive. $Ch(B) = Ch(D) = Ch(H) = Ch(F) = \epsilon$. $Ch(C) = D|\epsilon$. $\widehat{A} = \{A\}$, then $\widehat{RE}(A) = RE(A) = (B.C.A^?.H)|F$.

$$Ch_{\widehat{A}}^{rex}(\widehat{RE}(A)) = Ch_{\widehat{A}}^{rex}((B.C.A^?.H)|F) = (Ch_{\widehat{A}}(B).Ch_{\widehat{A}}(C).Ch_{\widehat{A}}(A)^?.Ch_{\widehat{A}}(H))|Ch_{\widehat{A}}(F)$$

$$= ((B|Ch(B)).(C|Ch(C)).(A|\epsilon)^?.(H|Ch(H)))|(F|Ch(F))$$

$$= ((B|\epsilon).(C|(D|\epsilon)).(A|\epsilon)^?.(H|\epsilon))|(F|\epsilon), \text{ simplified into } (B^?.(C|D|\epsilon).A^?.H^?)|F^?.$$

$Left(A) = \{B, C\}$, then $Succ(Left(A)) = \{B, C, D\}$.

$Right(A) = \{H\}$, then $Succ(Right(A)) = \{H\}$.

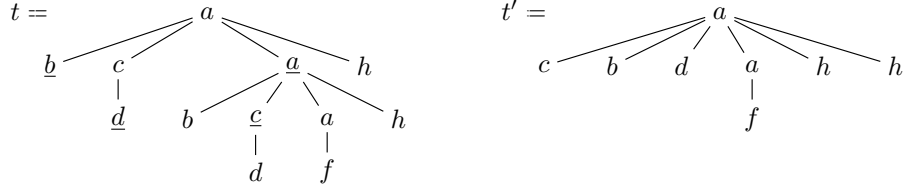
Considered as regular expressions (instead of sets), $Succ(Left(A)) = B|C|D$ and

$Succ(Right(A)) = H$.

Therefore $Ch(A) = (Succ(Left(A)))^* . Ch_{\widehat{A}}^{rex}(\widehat{RE}(A)) . (Succ(Right(A)))^* = (B|C|D)^* . [(B^? . (C|D|\epsilon) . A^? . H^?) | F^?] . H^*$, which could be simplified into $(B|C|D)^* . (A^? | F^?) . H^*$; we get the grammar G' :

$A \rightarrow a[(B|C|D)^* . (A^? | F^?) . H^*], B \rightarrow b[\epsilon], C \rightarrow c[D|\epsilon], D \rightarrow d[\epsilon], H \rightarrow h[\epsilon], F \rightarrow f[\epsilon]$

The tree t below is generated by G . By removing underlined symbols, we get $t' \triangleleft t$, and t' is generated by G' indeed. Note that c is a left-sibling of b in t' , which is impossible for a tree generated by the initial grammar G . On the other hand, b, c, d are necessarily to the left of h in t' .



Example 10. The previous example does not show the role of equivalence classes. Consider $G = \{A \rightarrow a[B^?], B \rightarrow b[A]\}$. A and B are 1-recursive.

$A \equiv B$ then $\widehat{A} = \widehat{B} = \{A, B\}$. $Left(A) = Left(B) = Right(A) = Right(B) = \emptyset$.

Therefore $Ch(A) = Ch_{\widehat{A}}^{rex}(\widehat{RE}(A)) = Ch_{\widehat{A}}^{rex}(B^?|A) = (Ch_{\widehat{A}}(B))^?|(Ch_{\widehat{A}}(A)) = (\widehat{B}|\epsilon)^?|(\widehat{A}|\epsilon) = (A|B|\epsilon)^?|(A|B|\epsilon)$. Note that \widehat{A} and \widehat{B} have been replaced by $A|B$, which is needed as shown by trees t and t' below. $Ch(A)$ can be simplified into $A|B|\epsilon$. Since $A \equiv B$, $Ch(B) = Ch(A)$.

Then we get the grammar $G' = \{A \rightarrow a[A|B|\epsilon], B \rightarrow b[A|B|\epsilon]\}$.

The tree $t = a(b(a))$ is generated by G . By removing b , we get $t' = a(a)$ which is generated by G' indeed.

4 Implementation and Experiments

Our prototype is implemented in Java and the experiments are done on an Intel Quad Core i3-2310M with 2.10GHz and 8GB of memory. The only step that takes time is the computation of recursivity types of non-terminals. The difficulty is for deciding whether a recursive non-terminal is 2- or 1-recursive. To do it, we have implemented two algorithms: one using Warshall algorithm for computing $>^+$, whose runtime is $O(n^6)$ where n is the number of non-terminals³, and another based on comparison of cycles in a graph representing relation $>$ (over non-terminals, not over multisets). In the worst case, the runtime of the second algorithm is at least exponential, since all cycles should be detected. Actually, the runtime of the first algorithm depends on the number n of non-terminals, whereas the runtime of the second one depends on the number of cycles in the graph.

In Table 1, $\#1-rec$ denotes the number of 1-recursive non-terminals (idem for $\#0-rec$ and $\#2-rec$), $\#Cycles$ is the number of cycles, and $|G|$ (resp. $|WI(G)|$)

³ Since grammars are in normal form, n is also the number of production rules.

denotes the sum of the sizes of the regular expressions⁴ occurring in the initial grammar G (resp. in the resulting grammar $WI(G)$). Results in lines 1 to 4 concern synthetic DTDs, while those in lines 5 to 6 correspond to real DTDs. The experiments show: if $n < 50$, the Warshall-based algorithm takes less than 6 seconds. Most often, the cycle-based algorithm runs faster than the Warshall-based algorithm. An example with $n = 111$ (line 3) took 7 minutes with the first algorithm, and was immediate with the second one. When the number of cycles is less than 100, the second algorithm is immediate, even if the runtime in the worst case is bad.

Now, consider the DTD (line 5) specifying the linguistic annotations of named entities performed within the National Corpus of Polish project [2, page 22]. After transforming this DTD into a grammar, we get 17 rules and some non-terminals are recursive. Both algorithms are immediate (few rules and few cycles). The example of line 6 specifies XHTML DTD⁵ (with $n = 85$ and $\#Cycles = 9620$). The Warshall-based algorithm and the cycle-based algorithm respond in 2 minutes.

	Unranked grammars				Runtime (s)		Sizes	
	#0-rec	#1-rec	#2-rec	#Cycles	Warshall	Cycle-compar.	G	WI(G)
1	9	2	38	410	5.48	0.82	183	1900
2	34	4	12	16	5.51	0.08	126	1317
3	78	12	21	30	445	0.2	293	4590
4	8	2	16	788	0.38	1.51	276	397
5	14	0	2	1	0.08	0.01	30	76
6	30	0	55	9620	136.63	113.91	1879	22963

Table 1. Runtimes in seconds for the Warshall-based and the Cycle-comparison algorithms.

5 Related Work and Discussion

The (weak) tree inclusion problem was first studied in [12], and improved in [5, 7, 15]. Our proposal differs from these approaches because we consider the weak inclusion with respect to *tree languages* (and not only with respect to trees). Testing precise inclusion of XML types is considered in [6, 8, 9, 14]. In [14], the authors study the complexity, identifying efficient cases. In [6] a polynomial algorithm for checking whether $L(A) \subseteq L(D)$ is given, where A is an automaton for unranked trees and D is a deterministic DTD.

In this paper, given a regular unranked-tree grammar G (hedge grammar), we have presented a direct method to compute a grammar G' that generates the set of trees (denoted $WI(L(G))$) weakly included in trees generated by G .

In [1], we have computed G' by transforming unranked-tree languages into binary-tree ones, using first-child next-sibling encoding. Then the weak-inclusion relation \triangleleft is expressed by a context-free synchronized ranked-tree language, and

⁴ The size of a regular expression E is the number of non-terminal occurrences in E .

⁵ <http://www.w3.org/TR/xhtml1/dtds.html>

using join and projection, we get G_1 . By transforming G_1 into an unranked-tree grammar, we get G' . This method is complex, and gives complex grammars.

Another way to compute G' could be the following. For each rule $A \rightarrow a[E]$ in G we add the *collapsing rule* $A \rightarrow E$. The resulting grammar G_1 generates $WI(L(G))$ indeed, but is not a hedge grammar: it is called *extended grammar*⁶ in [11], and can be transformed into a context-free hedge grammar G_2 (without collapsing rules). Each hedge H of G_2 is a context-free word language over non-terminals defined by a word grammar, and if we consider its closure by sub-word⁷, we get a regular word language H' defined by a regular expression [10]. Let G' be the grammar obtained from G_2 by transforming every hedge in this way. Then G' is a regular hedge grammar, and the language generated by G' satisfies $L(G') = L(G_2) \cup L_2$ where $L_2 \subseteq WI(L(G_2))$ (because of sub-word closure of hedges). Moreover $L(G_2) = L(G_1) = WI(L(G))$. Then $L_2 \subseteq WI(L(G_2)) = WI(WI(L(G))) = WI(L(G))$. Therefore $L(G') = WI(L(G)) \cup L_2 = WI(L(G))$.

References

1. Amavi, J.: Comparaison des langages d'arbres pour la substitution de services web (in french). Tech. Rep. RR-2010-13, LIFO, Université d'Orléans (2010)
2. Amavi, J., Bouchou, B., Savary, A.: On correcting XML documents with respect to a schema. Tech. Rep. 301, LI, Université de Tours (2012)
3. Amavi, J., Chabin, J., Halfeld Ferrari, M., Réty, P.: Weak inclusion for XML types. In: CIAA. vol. 6807, pp. 30–41. LNCS, Springer (2011)
4. Amavi, J., Chabin, J., Réty, P.: Weak inclusion for recursive XML types. In: CIAA. vol. xxx. LNCS, Springer (2012)
5. Bille, P., Gørtz, I.L.: The tree inclusion problem: In optimal space and faster. In: Automata, Languages and Programming, 32nd ICALP. pp. 66–77 (2005)
6. Champavère, J., Guilleron, R., Lemay, A., Niehren, J.: Efficient inclusion checking for deterministic tree automata and DTDs. In: Int. Conf. Language and Automata Theory and Applications. LNCS, vol. 5196, pp. 184–195. Springer (2008)
7. Chen, Y., Shi, Y., Chen, Y.: Tree inclusion algorithm, signatures and evaluation of path-oriented queries. In: Symp. on Applied Computing. pp. 1020–1025 (2006)
8. Colazzo, D., Ghelli, G., Pardini, L., Sartiani, C.: Linear inclusion for XML regular expression types. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM. pp. 137–146. ACM Digital Library (2009)
9. Colazzo, D., Ghelli, G., Sartiani, C.: Efficient asymmetric inclusion between regular expression types. In: Proceeding of International Conference of Database Theory, ICDT. pp. 174–182. ACM Digital Library (2009)
10. Courcelle, B.: On constructing obstruction sets of words. Bulletin of the EATCS 44, 178–185 (June 1991)
11. Jacquemard, F., Rusinowitch, M.: Closure of hedge-automata languages by hedge rewriting. In: RTA. pp. 157–171 (2008)
12. Kilpeläinen, P., Mannila, H.: Ordered and unordered tree inclusion. SIAM J. Comput. 24(2), 340–356 (1995)

⁶ In [11], they consider automata, but by reversing arrows, we can get grammars.

⁷ A sub-word of a word w is obtained by removing symbols from w . For example, $abeg$ is a sub-word of $abcdefgh$.

13. Mani, M., Lee, D.: XML to relational conversion using theory of regular tree grammars. In: In VLDB Workshop on EEXTT. pp. 81–103. Springer (2002)
14. Martens, W., Neven, F., Schwentick, T.: Complexity of decision problems for simple regular expressions. In: Int. Symp. Mathematical Foundations of Computer Science, MFCS. pp. 889–900 (2004)
15. Richter, T.: A new algorithm for the ordered tree inclusion problem. In: Apostolico, A., Hein, J. (eds.) Combinatorial Pattern Matching, LNCS, vol. 1264, pp. 150–166. Springer (1997)

Appendix: Proof of Theorem 1

A Termination

Definition 10. Relation $>$ over non-terminals was introduced in Definition 6. We extend it to equivalence classes by $\widehat{A} \widehat{\succ} \widehat{B} \iff A >^+ B \wedge A \not\equiv B$.

Remark 3. Suppose $\widehat{A}' = \widehat{A}$, $\widehat{B}' = \widehat{B}$ and $\widehat{A} \widehat{\succ} \widehat{B}$. We have $A' \equiv A$ ($A' \in \widehat{A}$) and $B' \equiv B$ then $A' >^* A >^+ B >^* B'$, consequently $A' >^+ B'$. On the other hand, if $A' \equiv B'$ then $A \equiv B$ which is impossible. Therefore $\widehat{A}' \widehat{\succ} \widehat{B}'$.

Lemma 2. $\widehat{\succ}$ is transitive.

Proof. Suppose $\widehat{A} \widehat{\succ} \widehat{B} \widehat{\succ} \widehat{C}$, therefore $A >^+ B >^+ C$ then $A >^+ C$. If $A \equiv C$, $C >^* A$ and as $A >^+ B$ we have $C >^* B$. By hypothesis, $B >^+ C$ then $B \equiv C$ which is impossible because $\widehat{B} \widehat{\succ} \widehat{C}$. So $A \not\equiv C$, and $A >^+ C$ then $\widehat{A} \widehat{\succ} \widehat{C}$.

Lemma 3. $\widehat{\succ}$ is noetherian.

Proof. Suppose $\widehat{A}_1 \widehat{\succ} \widehat{A}_2 \widehat{\succ} \dots \widehat{\succ} \widehat{A}_n \widehat{\succ} \dots$. Since the set NT of non-terminal symbols is finite, the set of equivalence classes over NT is also finite, then $\exists i, j$ such that $i > j$ and $\widehat{A}_i = \widehat{A}_j = \widehat{A}$. Therefore $\widehat{A} \widehat{\succ} \dots \widehat{\succ} \widehat{A}$. From previous lemma $\widehat{\succ}$ is transitive, then $\widehat{A} \widehat{\succ} \widehat{A}$ which implied $A \not\equiv A$, which is impossible.

Theorem 2. For each non-terminal A , the computation of $Ch(A)$ terminates.

Proof. We prove that, when computing $Ch(A)$, if we obtain $Ch(B)$ as a recursive call, then $\widehat{A} \widehat{\succ} \widehat{B}$. As $\widehat{\succ}$ is noetherian we will conclude that the computing of $Ch(A)$ terminates.

If A is 2-recursive, the definition of $Ch(A)$ is not recursive. Suppose that A is 1-recursive or not recursive. $Ch(B)$ is contained in the expression $Ch_{\widehat{A}}^{rex}(\widehat{RE}(A))$ (if A is not recursive, $\widehat{RE}(A) = RE(A)$ because $\widehat{A} = \{A\}$). B is a non-terminal in $\widehat{RE}(A)$, then there exists $A_i \in \widehat{A}$ and a rule $A_i \rightarrow a_i[E_i]$ such that $B \in NT(E_i)$. We deduce $A_i > B$. $Ch(B)$ must be contained in $Ch_{\widehat{A}}(B)$. If $A_i \equiv B$, A_i and B are in \widehat{A} then B is 1-recursive (because A is not 2-recursive), but in this case, by definition $Ch_{\widehat{A}}(B)$ does not contain $Ch(B)$. We deduce that if $Ch_{\widehat{A}}(B)$ contains $Ch(B)$ then $A_i \not\equiv B$ and then $\widehat{A}_i \widehat{\succ} \widehat{B}$. Since $\widehat{A}_i = \widehat{A}$, we have $\widehat{A} \widehat{\succ} \widehat{B}$.

B Soundness ($L(G') \subseteq WI(L(G))$)

Definition 11. We define the relation R over $\widehat{NT} \times \mathbb{N}$ by $(\widehat{A}, n) R (\widehat{A}', n') \iff (\widehat{A} \widehat{\succ} \widehat{A}') \vee (\widehat{A} = \widehat{A}' \wedge n > n')$. In other words, $R = (\widehat{\succ}, >)$ lexicographically where $\widehat{\succ}$ is defined above and $>$ is the usual order over natural integers.

Lemma 4. R is noetherian.

Proof. Because $\hat{>}$ and $>$ are noetherian.

Lemma 5.

If $B_1 \cdots B_n \in L^\omega(\text{Ch}(A))$, then $\begin{array}{c} A \\ / \quad | \quad \backslash \\ B_1 \quad \cdots \quad B_n \end{array} \in \text{WI}(L_G(A))$.

Proof. The proof is by noetherian induction on R over (\hat{A}, n) .

- If A is not recursive. Let $A \rightarrow a[E] \in G$, we have $\text{Ch}(A) = \text{Ch}_{\hat{A}}^{\text{rex}}(\text{RE}(A)) = \text{Ch}_{\hat{A}}^{\text{rex}}(E)$. So, there exists a word $C_1 \cdots C_k \in L^\omega(E)$ and words w_1, \dots, w_k such that $\forall i \in \{1, \dots, k\}$, $w_i \in L^\omega(\text{Ch}(C_i))$ if C_i is 1-recursive or 2-recursive, and $w_i \in L^\omega(\text{Ch}(C_i))$ or $w_i = C_i$ if C_i is not recursive. Let's note $B_1 \cdots B_n = w_1, \dots, w_k$. We have $\forall i A > C_i$ and $A \neq C_i$ because A is not recursive. Then $\hat{A} \hat{>} \hat{C}_i$ then $(\hat{A}, n) R (\hat{C}_i, |w_i|)$.

By induction hypothesis, $C_i \in \text{WI}(L_G^{\text{nt}}(C_i))$ (except when $w_i = C_i$)

\downarrow
 w_i

As $\begin{array}{c} A \\ / \quad | \quad \backslash \\ C_1 \quad \cdots \quad C_k \end{array} \in \text{WI}(L_G^{\text{nt}}(A))$ we have $\begin{array}{c} A \\ / \quad | \quad \backslash \\ C_1? \quad \cdots \quad C_k? \\ | \quad \quad \quad | \\ w_1 \quad \quad \quad w_k \end{array} \in \text{WI}(L_G^{\text{nt}}(A))$

(subtree $C_i?$ is w_i when $w_i = C_i$).

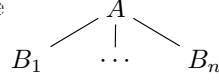
Then $\begin{array}{c} A \\ / \quad | \quad \backslash \\ w_1 \quad \cdots \quad w_k \end{array} = \begin{array}{c} A \\ / \quad | \quad \backslash \\ B_1 \quad \cdots \quad B_n \end{array} \in \text{WI}(L_G^{\text{nt}}(A))$.

- If A is 1-recursive.
 - If $B_1 \in \text{Succ}(\text{Left}(A))$, by definition of *Succ* and *Left*, and as A is 1-recursive, it exists in $L_G^{\text{nt}}(A)$ a term with A as root, a leaf A and at the left of the path from root to the leaf A , a leaf B_1 . We deduce that the tree $\begin{array}{c} A \\ / \quad \backslash \\ B_1 \quad A \end{array} \in \text{WI}(L_G^{\text{nt}}(A))$. However $B_2 \cdots B_n \in L^\omega(\text{Ch}(A))$

(due to construction of $\text{Ch}(A)$ when A is 1-recursive), and $|B_2 \cdots B_n| = n - 1$ so $(\hat{A}, n) R (\hat{A}, n - 1)$. We can apply induction hypothesis to obtain $\begin{array}{c} A \\ / \quad | \quad \backslash \\ B_2 \quad \cdots \quad B_n \end{array} \in \text{WI}(L_G^{\text{nt}}(A))$ then $\begin{array}{c} A \\ / \quad \backslash \\ A_1 \quad A \\ / \quad | \quad \backslash \\ B_2 \quad \cdots \quad B_n \end{array} \in \text{WI}(L_G^{\text{nt}}(A))$

and we conclude $\begin{array}{c} A \\ / \quad | \quad \backslash \\ B_1 \quad \cdots \quad B_n \end{array} \in \text{WI}(L_G^{\text{nt}}(A))$.

- Else if $B_n \in Succ(Right(A))$, we apply the same reasoning that the previous case and we also conclude $A \in WI(L_G^{nt}(A))$.



- Else $B_1 \cdots B_n \in L^\omega(Ch_{\widehat{A}}^{rex}(\widehat{RE}(A)))$. $\exists C \rightarrow c[E] \in G$, $C \in \widehat{A}$ and $B_1 \cdots B_n \in L^\omega(Ch_{\widehat{A}}^{rex}(E))$. Then $\exists D_1 \cdots D_k \in L^\omega(E)$ and words w_1, \dots, w_k such that $\forall i \in \{1, \dots, k\}$

- * $w_i \in \widehat{D}_i$ or $w_i = \epsilon$ if $D_i \in \widehat{A}$.
- * $w_i = D_i$ or $w_i \in L^\omega(Ch(D_i))$ if D_i is not recursive.
- * $w_i \in L^\omega(Ch(D_i))$ else.

and $B_1 \cdots B_n = w_1 \cdots w_k$. We have $\forall i, A >^* C > D_i$ then $A >^+ D_i$.

- * If $A \equiv D_i$ then $D_i \in \widehat{A}$ and ($w_i \in \widehat{D}_i$ or $w_i = \epsilon$) then we have a tree in $L_G^{nt}(D_i)$ with root D_i and w_i as leaves (with perhaps other leaves). Then $D_i? \in WI(L_G^{nt}(D_i))$.



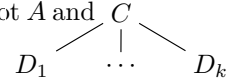
- * If D_i is not recursive and $w_i = D_i$ then $D_i? \in WI(L_G^{nt}(D_i))$.



- * If $w_i \in L^\omega(Ch(D_i))$ and $A \not\equiv D_i$. From $A >^+ D_i$ and $A \not\equiv D_i$ we have $\widehat{A} \widehat{>} \widehat{D}_i$ then $(\widehat{A}, n) R(\widehat{D}_i, |w_i|)$. By induction hypothesis, we have $D_i \in WI(L_G^{nt}(D_i))$.

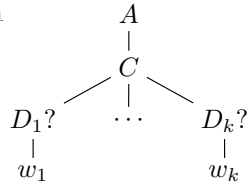


However $C \in \widehat{A}$ then it exists a tree in $L_G^{nt}(A)$ with root A and

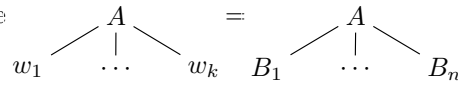


as subterm (perhaps other subterms exist).

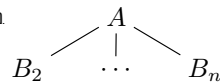
Then $A \in WI(L_G^{nt}(A))$



we can conclude $w_1 \cdots w_k = B_1 \cdots B_n \in WI(L_G^{nt}(A))$.



- If A is 2-recursive. We have $B_2 \cdots B_n \in L^\omega(Ch(A))$ (due to construction of $Ch(A)$ when A is 2-recursive) and $(\widehat{A}, n) R(\widehat{A}, n-1)$. We can apply induction hypothesis to obtain



Moreover, $B_1 \in Succ(A)$ and A is 2-recursive, then it exists in $L_G^{nt}(A)$ a tree with root A and 2 leaves B_1 and A (perhaps other leaves exist). Then

$$\begin{array}{ccc}
\begin{array}{c} A \\ / \quad \backslash \\ B_1 \quad A \end{array} \in WI(L_G^{nt}(A)), & \text{therefore} & \begin{array}{c} A \\ / \quad \backslash \\ B_1 \quad A \\ / \quad | \quad \backslash \\ B_2 \quad \dots \quad B_n \end{array} \in WI(L_G^{nt}(A)) \\
\text{and we conclude} & & \\
\begin{array}{c} A \\ / \quad | \quad \backslash \\ B_1 \quad \dots \quad B_n \end{array} & \in WI(L_G^{nt}(A)). &
\end{array}$$

Theorem 3. Correctness

$\forall A \in NT, \forall t \in Tree(NT), t \in L_{G'}^{nt}(A) \implies t \in WI(L_G^{nt}(A)).$

Proof. By induction on $|t|$.

Let $t = \begin{array}{c} A \\ / \quad | \quad \backslash \\ t_1 \quad \dots \quad t_n \end{array} \in L_{G'}^{nt}(A)$, then $\exists B_1 \dots B_n \in L^w(Ch(A))$ such that

$\forall i, t_i \in L_{G'}^{nt}(B_i).$

$\forall i, |t_i| < |t|$. By induction hypothesis, $t_i \in WI(L_G^{nt}(B_i))$. From Lemma 5 we get

$$\begin{array}{ccc}
\begin{array}{c} A \\ / \quad | \quad \backslash \\ B_1 \quad \dots \quad B_n \end{array} \in WI(L_G^{nt}(A)). & \text{Then } t = & \begin{array}{c} A \\ / \quad | \quad \backslash \\ t_1 \quad \dots \quad t_n \end{array} \in WI(L_G^{nt}(A)).
\end{array}$$

C Completeness ($L(G') \supseteq WI(L(G))$)

Definition 12. A *forest* is a (possibly empty) sequence of trees. The empty forest is denoted by ϵ . The size $|f|$ of the forest $f = t_1 \dots t_n$ is defined by $|f| = |t_1| + \dots + |t_n|$.

Given a word $u = A_1 \dots A_n$ over non-terminals, $L_G^f(u)$ is the set of forests (composed of nt-trees) defined by $L_G^f(u) = \{t_1 \dots t_n \mid \forall i, t_i \in L_G^{nt}(A_i)\}$. By convention, $L_G^f(\epsilon) = \{\epsilon\}$.

Lemma 6. $\forall A \in NT, \epsilon \in L^w(Ch(A)).$

Proof. The proof is by noetherian induction on $\hat{>}$.

- If A is not recursive, we have $Ch(A) = Ch_{\hat{A}}^{rex}(RE(A)). \forall B \in RE(A), A > B$ and $A \not\equiv B$ (because A is not recursive) and then $\hat{A} \hat{>} \hat{B}$. By induction hypothesis $\epsilon \in L^w(Ch(B))$ and so we conclude that $\epsilon \in L^w(Ch(A))$.
- If A is 1-recursive, we must show that $\epsilon \in Ch_{\hat{A}}^{rex}(\widehat{RE}(A))$. Let $B \in \widehat{RE}(A)$. We have $A >^+ B$ and two possible cases:
 - If $B \not\equiv A$, then $\hat{A} \hat{>} \hat{B}$. By induction hypothesis we have $\epsilon \in L^w(Ch(B))$.
 - Else $B \equiv A$, and therefore $B \in \hat{A}$. To obtain $Ch_{\hat{A}}^{rex}(\widehat{RE}(A))$, B was replaced by $\hat{B}|\epsilon$ (which generates ϵ).

By considering this two cases, if we replace each B in $\widehat{RE}(A)$ we can generate ϵ and therefore we have $\epsilon \in Ch_{\hat{A}}^{rex}(\widehat{RE}(A))$.

- If A is 2-recursive, due to the value of $Ch(A) = (Succ(A))^*$, it is easy to verify that $\epsilon \in L^w(Ch(A))$.

Corollary 1. $\forall A, B \in NT, \epsilon \in L^w(Ch_{\hat{A}}(B))$.

Lemma 7. *If A is 1-recursive or 2-recursive, then $A \in L^w(Ch(A))$.*

Proof. – If A is 2-recursive, we have $A \in Succ(A)$ and therefore $A \in L^w(Ch(A))$.

- If A is 1-recursive, we show that $A \in Ch_{\hat{A}}^{rex}(\widehat{RE}(A))$.

For all $C \in \widehat{RE}(A)$, from Lemma 6, we have $\epsilon \in L^w(Ch(C)) \subseteq L^w(Ch_{\hat{A}}(C))$.

Since A is 1-recursive, we know also that there exists $B \in \widehat{RE}(A)$ such that $B \equiv A$. To obtain $Ch_{\hat{A}}^{rex}(\widehat{RE}(A))$, we replace B by $Ch_{\hat{A}}(B) = \hat{B}|\epsilon = \hat{A}|\epsilon$ and we have $A \in L^w(Ch_{\hat{A}}(B))$. Since we can generate ϵ with $Ch_{\hat{A}}(C)$, we conclude that $A \in Ch_{\hat{A}}^{rex}(\widehat{RE}(A))$.

Lemma 8.
$$\begin{array}{c} A \\ / \quad | \quad \backslash \\ B_1 \quad \dots \quad B_n \end{array} \in L_{G'}^{nt}(A) \iff B_1 \dots B_n \in L^\omega(Ch(A)).$$

Proof. Trivial by the definition of the Algorithm.

Lemma 9. *If $A \equiv B$ then $Ch(A) = Ch(B)$.*

Proof. If A is not recursive, from Lemma 1, $B = A$; then $Ch(B) = Ch(A)$. Otherwise, from Lemma 1 we have $Succ(A) = Succ(B)$; $Succ(Left(A)) = Succ(Left(B))$; $Succ(Right(A)) = Succ(Right(B))$; $\widehat{RE}(A) = \widehat{RE}(B)$. Therefore $Ch(A) = Ch(B)$.

Corollary 2. $\forall C \in NT, \text{ If } A \equiv B \text{ then } Ch_{\hat{C}}(A) = Ch_{\hat{C}}(B)$.

Lemma 10. $\forall B \in NT, (B \in NT(Ch(A)) \implies A >^* B)$.

Proof. The proof is by noetherian induction on \hat{S} .

- If A is 2-recursive, $Ch(A) = (Succ(A))^*$. We deduce that $B \in NT(Succ(A))$ and so $A >^* B$.
- If A is 1-rec, $Ch(A) = (Succ(Left(A)))^* \cdot Ch_{\hat{A}}^{rex}(\widehat{RE}(A)) \cdot (Succ(Right(A)))^*$. We have two possible cases:
 - $B \in NT(Succ(Left(A)))$ or $B \in NT(Succ(Right(A)))$: in this case there exists $C \in \hat{A}$ such that $C >^* B$, and since $A >^* C$ (because $C \in \hat{A}$) we obtain $A >^* B$.
 - $B \in NT(Ch_{\hat{A}}^{rex}(\widehat{RE}(A)))$. Then $\exists C \in \widehat{RE}(A), B \in NT(Ch_{\hat{A}}(C))$.
As $C \in \widehat{RE}(A)$, there exists $A_1 \in \hat{A}$ s.t. $C \in RE(A_1)$. Then $A \equiv A_1$ and $A_1 > C$. Then $A >^* A_1$ and $A_1 > C$. Therefore $A >^+ C$.
If $C = B$ then we have automatically $A >^* B$. Otherwise we have three situations for $Ch_{\hat{A}}(C)$:

- i) C is 1-recursive and $C \in \widehat{A}$: $Ch_{\widehat{A}}(C) = \widehat{C}|\epsilon$ and so $B \in NT(\widehat{C}|\epsilon)$. Then $B \in \widehat{C} = \widehat{A}$, then $B \equiv A$, which establishes $A >^* B$.
 - ii) (C is 2-recursive) or (C is 1-recursive and $C \notin \widehat{A}$): $Ch_{\widehat{A}}(C) = Ch(C)$ and so $B \in NT(Ch(C))$. As C is 2-recursive and A is 1-recursive or $C \notin \widehat{A}$ we have in the two cases that $A \not\equiv C$. Since $A >^+ C$ and $A \not\equiv C$, by Definition 10 we have $\widehat{A} \succ \widehat{C}$. We know that $B \in NT(Ch(C))$ and $\widehat{A} \succ \widehat{C}$, by applying the induction hypothesis we obtain $C >^* B$. And since $A >^+ C$, we conclude that $A >^* B$.
 - iii) C is not recursive: $Ch_{\widehat{A}}(C) = C|Ch(C)$ and so $B \in NT(C|Ch(C))$. If $B = C$, we have automatically $A >^* B$. If $B \in NT(Ch(C))$, as C is not recursive and A is 1-recursive we have $A \not\equiv C$. By the same argument as in ii) we claim that $A >^* B$.
- If A is not recursive, the proof is as in the case 1-recursive (b), except that we replace \widehat{RE} by RE .

Theorem 4. $\forall p \in \mathbb{N}, [\forall A \in NT, \forall w \in L^w(RE(A)), \forall f \in L_G^f(w), \forall f' \triangleleft f, (|f| \leq p \implies \exists w' \in L^w(Ch(A)), f' \in L_{G'}^f(w'))]$.

Proof. The proof is by induction on p . Let $w = (A_1, \dots, A_n)$ and $f = (t_1, \dots, t_n)$. We have $t = A(t_1, \dots, t_n) \in L_G^{nt}(A)$ which is described by Figure 2.

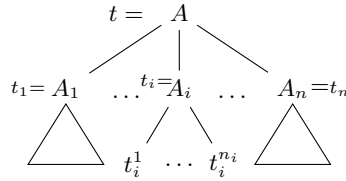
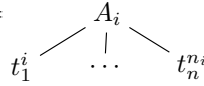


Fig. 2. Example for the tree $t \in L_G^{nt}(A)$ with $f = (t_1, \dots, t_n)$

Let $f' = (f'_1, \dots, f'_n)$ such that $\forall i \in [1, \dots, n], f'_i \triangleleft t_i$ (note that the root of t_i can be removed, in other words, t_i may be a forest).

- if $p = 0$ then $w = \epsilon, w' = \epsilon, f = \epsilon$ and $f' = \epsilon$. By Lemma 6, $\epsilon \in L^w(Ch(A))$ and therefore $f' \in L_{G'}^f(\epsilon) = \{\epsilon\}$.
- else $p \neq 0$ and then $n \neq 0$. $t_i =$



as described in Figure 2,

and $f_i = (t_i^1, \dots, t_i^{n_i})$. Thus $f'_i = A_i$ (illustrated in Figure 3(a)) or



$f'_i = f''_i$ (illustrated in Figure 3(b)). Since $|f_i| < |f|$, by induction hypothesis we obtain: $\exists w''_i \in L^w(Ch(A_i)), f''_i \in L_{G'}^f(w''_i)$. However:

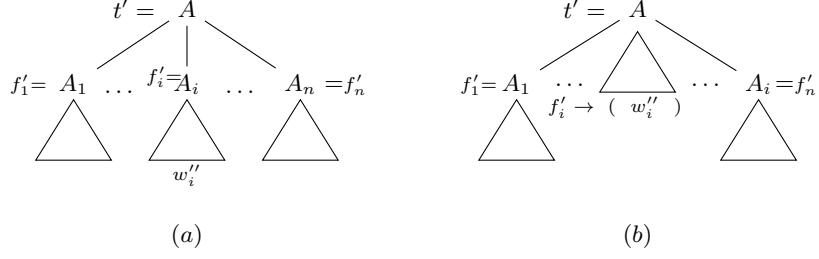


Fig. 3. Example of the two possibilities for the tree $t' \triangleleft t$, with $f' = (f'_1, \dots, f'_n)$ and $[f'_i = A_i(f''_i) \text{ or } f'_i = f''_i]$

$$\left\{ \begin{array}{l} \text{if } f'_i = A_i \text{ since } A_i \rightarrow a_i[Ch(A_i)] \in G', \text{ we have } f'_i \in L_{G'}^{n_i}(A_i). \text{ Let } w'_i = A_i \\ \quad \downarrow \\ \quad f''_i \\ \text{else } f'_i = f''_i \in L_{G'}^f(w''_i) \text{ with } w''_i \in L^w(Ch(A_i)). \text{ Let } w'_i = w''_i \end{array} \right.$$

Let $w' = (w'_1, \dots, w'_n)$, we have $f' \in L_{G'}^f(w')$.

We now show that $w' \in L^w(Ch(A))$:

- i) A is 2-recursive: since $Ch(A) = (Succ(A))^*$, It is sufficient to show that each non-terminal in w' is in $Succ(A)$. We have $w = (A_1, \dots, A_n) \in L^w(RE(A))$, and therefore $A > A_i$.
 - if $w'_i = A_i$, we have $w_i \in Succ(A)$,
 - if $w'_i = w''_i \in L^w(Ch(A_i))$, let $w'_i = B_1 \cdots B_k$. By Lemma 10, $\forall j \in [1, \dots, k]$ $A_i >^* B_j$, thus $B_j \in Succ(A)$.
- ii) A is not recursive: to obtain $Ch(A) = Ch_{\hat{A}}^{rex}(RE(A))$ from $RE(A)$, we replace in $RE(A)$ each A_i by:
 - $Ch(A_i)$ if A_i is 2-recursive or 1-recursive (note that the case A_i is 1-recursive and $A_i \in \hat{A}$ is impossible). And by Lemma 7 we have $A_i \in L^w(Ch(A_i))$
 - $A_i|Ch(A_i)$ if A_i is not recursive
However $w = (A_1 \cdots A_n) \in L^w(RE(A))$, $w' = (w'_1 \cdots w'_n)$ and $\forall i, w'_i = A_i$ or $w'_i \in L^w(Ch(A_i))$. Thus $w' \in L^w(Ch(A))$.
- iii) A is 1-rec: $Ch(A) = (Succ(Left(A)))^* \cdot Ch_{\hat{A}}^{rex}(\widehat{RE}(A)) \cdot (Succ(Right(A)))^*$
 - Suppose that $\forall i, A_i \not\equiv A$: we show that $w' \in L^w(Ch_{\hat{A}}^{rex}(\widehat{RE}(A))) \subseteq L^w(Ch(A))$ and the proof is similar to ii).
 - Else $\exists i, A_i \equiv A$ and then A_i is 1-recursive. Therefore
 - $\{ A_i, \dots, A_{i-1} \in Left(A_i) = Left(A) \subseteq Succ(Left(A))$
 - $\{ A_{i+1}, \dots, A_n \in Right(A_i) = Right(A) \subseteq Succ(Right(A))$
Also $\forall j \in [1, \dots, i-1]$, $(w'_j = B_1 \cdots B_k \implies \forall p, A_i >^* B_p \implies B_p \in Succ(Left(A)))$. Therefore all the non-terminals of $w'_1 \cdots w'_{i-1}$ are in $Succ(Left(A))$. By using the same argument all the non-terminals of $w'_{i+1} \cdots w'_n$ are in $Succ(Right(A))$.

In the other hand to obtain $Ch_A^{rex}(\widehat{RE}(A))$ from $\widehat{RE}(A)$, we replace all the non-terminals B by $\widehat{Ch}_A(B)$. By using Corollary 1, we have $\epsilon \in L^w(Ch_A(B))$. Also on the other hand, when $B = A_i$ and $Ch_A(A_i) = A_i|\epsilon$, it follows that $A_i \in L^w(Ch_A^{rex}(\widehat{RE}(A)))$. We now analyse the two possible values for the word w'_i :

- * if $w'_i = A_i$ then $w' = (w'_1, \dots, w'_i, \dots, w'_n) \in L^w(Ch(A))$
- * else $w'_i = w''_i \in L^w(Ch(A_i))$. By Lemma 9,

$$\begin{aligned} Ch(A_i) &= (Succ(Left(A_i)))^* . Ch_{A_i}^{rex}(\widehat{RE}(A_i)) . (Succ(Right(A_i)))^* \\ &= (Succ(Left(A)))^* . Ch_A^{rex}(\widehat{RE}(A)) . (Succ(Right(A)))^* \end{aligned}$$

And then $w' = (w'_1, \dots, w'_i, \dots, w'_n) \in$

$$\begin{aligned} &L^w(Succ(Left(A))^* . [Succ(Left(A))^* . Ch_A^{rex}(\widehat{RE}(A)) . Succ(Right(A))^*] . Succ(Right(A))^*) \\ &= L^w(Succ(Left(A))^* . Ch_A^{rex}(\widehat{RE}(A)) . Succ(Right(A))^*) = L^w(Ch(A)) \end{aligned}$$

This establishes the proof of Theorem 4.

Corollary 3. $\forall A \in NT, \forall t \in L_G(A), \forall t' \triangleleft t, t' \in L_{G'}(A)$.

Proof. Let $t \in L_G(A)$. When we replace each terminal symbol in t by the corresponding non-terminal symbol, we obtain $tt =$

$$\begin{array}{ccc} & A & \\ & | & \\ t_1 & \dots & t_n \end{array} \in L_G^{nt}(A) \text{ and}$$

$\forall i \in [1, \dots, n], t_i \in L_G^{nt}(A_i)$. We also replace each terminal symbol in t' by the corresponding non-terminal symbol and we obtain:

$$tt' = \begin{array}{ccc} & A & \\ & | & \\ t'_1 & \dots & t'_m \end{array} \text{ and } \forall j \in [1, \dots, m], t'_j(\epsilon) = A'_j.$$

Let $f = (t_1, \dots, t_n), w = (A_1, \dots, A_n), f' = (t'_1, \dots, t'_m)$ and $w' = (A'_1, \dots, A'_m)$. Furthermore we have $t \in L_G(A)$ and $t' \triangleleft t$ which implies that $w \in L^w(RE(A))$ and $f' \triangleleft f$. Thus by using Theorem 4, we have $w' \in L^w(Ch(A))$ and $f' \in L_{G'}^f(w')$. Therefore with Lemma 8, we claim that $tt' \in L_G^{nt}(A)$ as $w' \in L^w(Ch(A))$. We can conclude that $t' \in L_{G'}(A)$ after replacing non-terminal symbols by terminal symbols.