An Algorithm for Comparing Deterministic Regular Tree Grammars

Byron Long

Abstract

An algorithm to decide inclusion for languages defined by deterministic regular tree grammars is presented. The algorithm is shown to run in polynomial time based on the number of constructors, constructor arity, and number of non-terminals. Correctness proofs are included.

1 Regular tree grammars

A *regular tree grammar*, G, is a grammar for constructing trees. Formally, $G = (C, N, P, S)$ where:

1. C is a finite, non-empty set of constructors. Each constructor c^n is of arity n. Constants are 0-arity constructors.

2. N is a finite, non-empty set of non-terminals where $N \cap C = \emptyset$.

3. P is a finite set of productions of the form $A \rightarrow c^n(X_1, \ldots, X_n)$ where A is a non-terminal, c^n is a constructor, and each X_i is a non-terminal. If the constructor is a constant, then we write c^0 as a shorthand for $c^0()$.

4. S is a special element of N called the start symbol.

A regular tree is constructed from a regular tree grammar in the following manner:

1. If A is a non-terminal, then the tree whose only node is A is a regular tree.

2. If t is a regular tree, B is an occurrence of a non-terminal in t, and there is a production $B \rightarrow c^n(X_1, \ldots, X_n)$, then the result of replacing in t the occurrence of B with a tree whose root is c^n and whose children in left to right order are $X_1, \ldots, X_n$ is a regular tree.

If the regular tree t is constructed starting from a non-terminal A then we say that A derives t. A tree is a *proper* tree if it contains only constructors and is an *improper* tree if it contains a non-terminal. For a grammar $G = (C, N, P, S)$, the language generated by G, $L(G)$, is the set of all proper trees derivable from S.

A regular tree grammar is *deterministic* if for each non-terminal A and constructor c^n, there is at most one production of the form $A \rightarrow c^n(X_1, \ldots, X_n)$. In this paper we will consider only deterministic regular tree grammars and their languages.

Deterministic regular tree grammars provide a convenient way of describing recursive data. [Liu98] presents a method of analyzing recursive data that uses these grammars as results of a fixed point computation. A fixed point is found when consecutive grammars G_1 and G_2 are iterated such that $L(G_2)$ contains $L(G_1)$.

This paper describes a two-part polynomial time algorithm that determines if $L(G_1) \subseteq L(G_2)$ given grammars G_1 and G_2. Because the main algorithm requires a grammar where each non-terminal can derive a proper tree, we initially present a supplemental algorithm to change a grammar to this required form. Following sections describe the main algorithm and a runtime analysis of both supplemental and main algorithms. Correctness proofs are included for each algorithm.

2 Removing improper non-terminals

We say that a non-terminal is *proper* if it can derive a proper tree. Otherwise the non-terminal is *improper*. If every non-terminal is proper in a given grammar then we call the grammar proper.

The algorithm we will present for deciding if $L(G_1) \subseteq L(G_2)$ in the next section requires grammars that are proper. Thus we start with an algorithm to convert a grammar G into a proper grammar G' such that $L(G) = L(G')$ by removing improper non-terminals. A bottom-up approach is used by first marking as proper each non-terminal A for which there is a production of the form $A \rightarrow c^0$. Subsequently the non-terminal B is marked as proper if there is a production $B \rightarrow c^n(X_1, \ldots, X_n)$ such that every X_i is already marked as proper.

This conversion algorithm (Figure 1) will associate with each non-terminal a list of the productions whose right hand side contains the non-terminal. If the non-terminal appears more than once in a given production's right hand side, the production will also occur the same number of times in the non-terminal's list. In addition, a counter will be given to each production to record the number of improper non-terminals in that production's right hand side.

Theorem 1 *Non-terminal A is marked as proper by the* **Conversion** *algorithm if and only if it is proper.*

Proof: Assume A is marked as proper by the algorithm. It can be shown that A is proper by induction on k, the number of iterations of the while loop of lines 18-27 performed before A is marked as proper.

$k = 0$. A is marked as proper in line 14 if there is a production $A \rightarrow c^0$.

$k = u + 1$. Assume the lemma holds for non-terminals marked during the first u iterations of the loop. A is marked as proper if there is a production $A \rightarrow c^n(X_1, \ldots, X_n)$ whose counter reaches 0 by being decremented n times. The production $A \rightarrow c^n(X_1, \ldots, X_n)$ occurs a total of exactly n times in the *rhs-lists* of all non-terminals, specifically the *rhs-lists* of the non-terminals $X_1, \ldots, X_n$. Since each X_i contains the production at least once in its *rhs-list*, each X_i must have been marked as proper and placed in *propagate* during the first t iterations. By the assumption, each X_i is indeed proper. This means that A is also proper.

Now assume that A is proper. Induction on h, the height of t, a proper tree derivable from A, can show that A is marked as proper and placed in *propagate*.

$h = 0$. The tree t consists of the single constructor c^0, and A is marked as proper and placed in *propagate* in lines 13-14.

$h = u + 1$. Assume the lemma for non-terminals that derive proper trees of height u or less. Let $A \rightarrow c^n(X_1, \ldots, X_n)$ be the first production used in the derivation of t. Each X_i is of height u or less and by the assumption, is marked as proper and placed in *propagate*. The production $A \rightarrow c^n(X_1, \ldots, X_n)$ occurs a total of n times in the *rhs-lists* of non-terminals $X_1 \ldots X_n$. Since all *rhs-lists* of the X_i's are examined, the counter for $A \rightarrow c^n(X_1 \ldots X_n)$ is decremented n times. Since the counter was initialized with n at line 4, it will reach 0, and A is marked as proper.

Algorithm: Conversion
input: grammar G.
output: grammar G with improper non-terminals removed.

```
// initialize rhs-lists and production counters
for-each non-terminal A do
    set A's rhs-list to NIL
    for-each production A → c^n(X_1, ..., X_n) do
        set the production's counter to n
        for-each i from 1 to n do
            add the production to X_i's rhs-list
        end for-each
    end for-each
end for-each
// identify and collect initial proper non-terminals
propagate ← ∅
for each non-terminal A do
    if there is a production A → c^0
        propagate ← propagate ∪ {A}
        mark A as proper
    else
        mark A as improper
end for-each
// find other proper non-terminals
while propagate ≠ ∅ do
    remove B from propagate
    for-each production A → ... in B's rhs-list do
        if A is marked as improper
            decrement the production's counter
            if the counter = 0
                mark A as proper
                propagate ← propagate ∪ {A}
    end for-each
end while
// remove productions that contain improper non-terminals
for-each production A → c^n(X_1, ..., X_n) in G do
   if A is marked as improper
      remove A → c^n(X_1, ..., X_n) from G
   else
     for-each X_i in A → c^n(X_1, ..., X_n) do
       if X_i is marked as improper
          remove A → c^n(X_1, ..., X_n) from G
     end for-each
end for-each
return G
```

Figure 1: Conversion algorithm

Theorem 2 *Let G be a grammar and G' be the grammar derived by the* **Conversion** *algorithm. G' is a proper deterministic grammar such that $L(G) = L(G')$.*

Proof: Grammar G' is derived from G by only removing productions, so G' is deterministic. By Theorem 1, the non-terminals marked as improper derive only improper trees. For any tree derived using such a non-terminal, it must be the case that the entire tree is improper, and hence is not in $L(G)$. Thus, removing any productions containing that non-terminal from the grammar does not reduce the language it generates. These are exactly the productions removed in the loop of lines 28-36. Thus $L(G) = L(G')$, and every non-terminal in G' can derive a proper tree.

3 Testing for inclusion

Given proper grammars, G_1 and G_2, the algorithm for deciding inclusion proceeds by comparing non-terminal pairs $\langle A_1, A_2 \rangle$, where A_1 is in G_1 and A_2 is in G_2, and checking that every tree derivable from A_1 is also derivable from A_2. This is done by comparing the productions of A_1 to the productions of A_2. This comparison yields one of two results. First, it may be immediately clear that A_1 can derive a tree that A_2 cannot or that A_2 can derive every tree that A_1 can derive. Second, it may be the case that A_2's ability to derive all of A_1's trees depends on the comparison of other non-terminal pairs. For example, there may be the productions $A_1 \rightarrow c^2(B_1, C_1)$ and $A_2 \rightarrow c^2(B_2, C_2)$. In this case, if B_2 can derive all trees derivable from B_1, and C_2 can derive all trees derivable from C_1, then A_2 can derive everything that A_1 derives.

The algorithm (Figure 2) uses two sets. The first set, *to-test*, contains non-terminal pairs for which the algorithm will test that the trees derivable by the first non-terminal are derivable by the second. The *to-test* set starts as a singleton containing the pair of start symbols from the two grammars, $\langle S_1, S_2 \rangle$. As each pair is examined it is removed from *to-test*. When the decision for a pair of non-terminals in *to-test* depends on other non-terminal pairs, those pairs are also placed in *to-test*.

The second set, *testing*, is used to prevent non-terminal pairs from being placed in *to-test* more than once. As each pair is removed from *to-test*, it is then placed in *testing*. It may be convenient to think of non-terminal pairs in *testing* as being in the midst of testing by the algorithm (due to their dependence on other non-terminal pairs). As such, there is no need to re-insert them into *to-test*.

Pairs placed in *to-test* are dependencies for the original pair $\langle S_1, S_2 \rangle$. Thus, when a pair is found such that the first non-terminal can derive a tree not derivable from the second non-terminal, the algorithm halts indicating that S_1 can derive something not derivable by S_2. If *to-test* can be emptied without finding such a pair, then the algorithm stops and indicates that every tree derivable from S_1 is also derivable from S_2.

Checking for membership of $\langle A_1, A_2 \rangle$ in *testing* will be done in constant time by maintaining the set as an array of boolean values indexed by the non-terminals A_1 and A_2.

The correctness of the algorithm can be proven by showing that the algorithm returns false if and only if there is a tree that is derivable from S_1 but not derivable from S_2. In order to do this, for each non-terminal pair $\langle A_1, A_2 \rangle$ in *to-test*, the algorithm will track the path of non-terminals and productions needed to get from S_1 to A_1 and from S_2 to A_2.

Definition 1 *For two grammars G_1 and G_2, a* step *is a triple of the form:*

$$\langle A_1 \rightarrow c^n(X_{11}, \ldots, X_{1n}), A_2 \rightarrow c^n(X_{21}, \ldots, X_{2n}), i \rangle$$

Algorithm: Inclusion
input: proper grammars G_1 and G_2.
output: TRUE if $L(G_1) \subseteq L(G_2)$ otherwise FALSE.

// initialize *testing* and *to-test*
for-each non-terminal A_1 in G_1 **do**
  for-each non-terminal A_2 in G_2 **do**
    $testing[A_1, A_2] \leftarrow$ FALSE
$to\text{-}test \leftarrow \{\langle S_1, S_2 \rangle\}$
// check current dependencies
while $to\text{-}test \neq \emptyset$
  remove $\langle A_1, A_2 \rangle$ from *to-test*
  $testing[A_1, A_2] \leftarrow$ TRUE
// check if A_2 derives all that A_1 derives
  for-each constructor c^n in G_1 **do**
    if $A_1 \rightarrow c^n(X_{11}, \ldots, X_{1n})$ is in G_1
      if $A_2 \rightarrow c^n(X_{21}, \ldots, X_{2n})$ is in G_2
// add new dependencies
        for-each i from 1 to n **do**
          if not $testing[X_{1i}, X_{2i}]$
            $to\text{-}test \leftarrow to\text{-}test \cup \{\langle X_{1i}, X_{2i} \rangle\}$
        end for-each
      else
        return FALSE
  end for-each
end while
return TRUE

Figure 2: Inclusion algorithm

where $A_1 \rightarrow c^n(X_{11}, \ldots, X_{1n})$ *is a production of* G_1*,* $A_2 \rightarrow c^n(X_{21}, \ldots, X_{2n})$ *is a production of* G_2*, and* $1 \leq i \leq n$.

Definition 2 *A* path *is inductively defined as follows:*

1. *The empty path, symbolized by* $\circ$*, is a path.*
2. *A step is a path.*
3. *If* s *is a step, then* $\circ : s$ *is a path where* $\circ : s$ *is the concatenation of the empty path and* s.
4. *If* p *is a non-empty path, whose last step is:*
   $\langle A_1, \rightarrow c^n(X_{11}, \ldots, X_{1n}), A_2 \rightarrow c^n(X_{21}, \ldots, X_{2n}), i \rangle$
   and s *is a step of the form:*
   $\langle X_{1i} \rightarrow c^m(Y_{11}, \ldots, Y_{1m}), X_{2i} \rightarrow c^m(Y_{21}, \ldots, Y_{2m}), j \rangle$
   then $p : s$ *is a path.*

Definition 3 *The* length *of a path* p*,* $len(p)$ *is inductively defined as follows:*

1. *If* p *is the empty path or a single step then* $len(p) = 0$.
2. *If* p *is of the form* $q : s$*, then the* $len(p) = len(q) + 1$.

For non-terminal pair $\langle X_{1i}, X_{2i} \rangle$, the algorithm can be modified (Figure 3) to use the step

$\langle A_1 \rightarrow c^n(X_{11}, \ldots, X_{1n}), A_2 \rightarrow c^n(X_{21}, \ldots, X_{2n}), i\rangle$ to record the fact that X_{1i} is derivable from A_1 and X_{2i} is derivable from A_2 using the given productions. In addition, for $\langle X_{1i}, X_{2i}\rangle$ a path is used to record the sequence of ancestor non-terminals and productions used to get from S_1 to X_{1i}, and from S_2 to X_{2i}. The **Inclusion** algorithm can then be modified so that that the path for each $\langle X_{1i}, X_{2i}\rangle$ placed in *to-test* is recorded.

Proposition 1 *For every $\langle A_1, A_2, p\rangle$ placed in* to-test *by the* **Inclusion-r** *algorithm, p is a path.*

Algorithm: Inclusion-r
input: proper grammars G_1 and G_2.
output: TRUE if $L(G_1) \subseteq L(G_2)$ otherwise FALSE.

```
// initialize testing and to-test
for-each non-terminal A1 in G1 do
  for-each non-terminal A2 in G2 do
    testing[A1, A2] ← FALSE
to-test ← {⟨S1, S2, ∘⟩}
// check current dependencies
while to-test ≠ ∅
  remove ⟨A1, A2, p⟩ from to-test
  testing[A1, A2] ← TRUE
// check if A2 derives all that A1 derives
  for-each constructor c^n in G1 do
    if A1 → c^n(X11, ..., X1n) is in G1
      if A2 → c^n(X21, ..., X2n) is in G2
// add new dependencies
        for-each i from 1 to n do
          if not testing[X1i, X2i]
            to-test ← to-test ∪
                     {⟨X1i,
                       X2i,
                       p : ⟨A1 → c^n(X11, ..., X1n),
                            A2 → c^n(X21, ..., X2n),
                            i⟩⟩}
        end for-each
      else
        return FALSE
  end for-each
end while
return TRUE
```

Figure 3: Inclusion-r algorithm

Lemma 1 *Let G_1 and G_2 be proper grammars, and let p be a path whose last step is*

$$\langle A_1 \rightarrow c^n(X_{11}, \ldots, X_{1n}), A_2 \rightarrow c^n(X_{21}, \ldots, X_{2n}), i\rangle$$

If X_{1i} derives a proper tree that X_{2i} cannot, then for every step $\langle B_1 \rightarrow \ldots, B_2 \rightarrow \ldots, l\rangle$ in p, B_1 derives a proper tree that B_2 cannot derive.

Proof: Induction on l, the length of p.

$l = 0$. Path p consists of a single step. Since G_1 is proper, and G_2 is deterministic, A_1 derives a proper tree not derivable by A_2.

$l = u + 1$. Assume the lemma holds for paths of length u. Consider path p' which consists of the last u steps of p. Let s be the first step of the original path p, and let s' be the first step of p'. In path p, s is followed by s'. This means that s and s' must respectively be of the forms $\langle C_1 \rightarrow c^m(Y_{11}, \ldots, Y_{1m}), C_2 \rightarrow c^m(Y_{21}, \ldots, Y_{2m}), j\rangle$ and $\langle Y_{1j} \rightarrow \ldots, Y_{2j} \rightarrow \ldots, k)\rangle$. By the inductive assumption, Y_{1j} can derive a proper tree that Y_{2j} cannot. C_1 can derive $c^m(Y_{11}, \ldots, Y_{1m})$ and because the grammars are deterministic, the only c^m production from C_2 goes to $c^m(Y_{21}, \ldots, Y_{2m})$. Since G_1 is proper, C_1 can derive a proper tree that C_2 cannot derive.

Proposition 2 *In* **Inclusion-r***, every triple, $\langle A, B, p\rangle$, with non-empty path p placed into* to-test *was inserted in line 13.*

Lemma 2 *In* **Inclusion-r***, for every triple, $\langle A_1, A_2, p\rangle$, inserted into* to-test*, in line 13, p contains a step of the form $\langle S_1 \rightarrow \ldots, S_2 \rightarrow \ldots, i\rangle$.*

Proof: Induction on k the number of iterations of the while loop (line 5) performed when the triple $\langle A_1, A_2, p\rangle$ is inserted into *to-test*.

$k = 1$. On the first iteration the only triple in *to-test* is $\langle S_1, S_2, \circ\rangle$ and hence selected in line 6. This means that the path p of the triple $\langle A_1, A_2, p\rangle$ inserted into *to-test* in line 13 is of the form $\circ : \langle S_1 \rightarrow \ldots, S_2 \rightarrow \ldots, i\rangle$.

$k = u+1$. Assume that if a triple is inserted on the u-th iteration or before, its path contains a step of the form $\langle S_1 \rightarrow \ldots, S_2 \rightarrow \ldots, i\rangle$. If $\langle A_1, A_2, p\rangle$ is inserted on the $u + 1$-st iteration, then p must be of the form $q : s$, where q came from a triple that was already in *to-test* and thus inserted on a previous iteration. By the assumption, q contains a step of the form $\langle S_1 \rightarrow \ldots, S_2 \rightarrow \ldots, i\rangle$.

Theorem 3 *If* **Inclusion-r** *returns false then there is a proper tree that is derivable from S_1 but is not derivable from S_2.*

Proof: The algorithm returns false if it finds a triple $\langle A_1, A_2, p\rangle$ in *to-test* such that A_1 can derive a proper tree that A_2 cannot. If p is empty then the triple is $\langle S_1, S_2, \circ\rangle$, and we are done. Otherwise by Lemma 2, there is a step in p of the form $\langle S_1 \rightarrow \ldots, S_2 \rightarrow \ldots, i\rangle$ and hence by Lemma 1, there is a proper tree that S_1 can derive but S_2 cannot.

Proposition 3 *If* **Inclusion-r** *returns true, then every triple in* to-test *is examined.*

Proposition 4 *For every $\langle A_1, A_2\rangle$ in* testing*, $\langle A_1, A_2, p\rangle$ was placed in* to-test *for some path p.*

Lemma 3 *If* **Inclusion-r** *returns true, then for every path p whose first two steps are $\circ : \langle S_1 \rightarrow \ldots, S_2 \rightarrow \ldots, i\rangle$ and whose last step is $\langle B_1 \rightarrow c^m(Y_{11}, \ldots, Y_{1m}), B_2 \rightarrow c^m(Y_{21}, \ldots, Y_{2m}), k\rangle$, a triple $\langle Y_{1j}, Y_{2j}, q\rangle$ is placed in* to-test.

Proof: Induction on l, the length of path p.

$l = 1$. The path is $\circ : \langle S_1 \rightarrow c^m(Y_{11}, \ldots, Y_{1m}), S_2 \rightarrow c^m(Y_{21}, \ldots, Y_{2m}), i\rangle$. The triple $\langle S_1, S_2, \circ\rangle$ is selected by line 6 on the first iteration of the while loop of lines 5-18. Since the algorithm does not return false, and there are productions $S_1 \to c^m(Y_{11}, \ldots, Y_{1m})$ and $S_2 \to c^m(Y_{21}, \ldots, Y_{2m})$, the triple $\langle Y_{1j}, Y_{2j}, p\rangle$ is placed in *to-test*.

$l = u + 1$. Assume the lemma holds for paths of length u. Consider the last two steps of p, $\langle A_1 \to c^n(X_{11}, \ldots, X_{1n}), A_2 \to c^n(X_{21}, \ldots, X_{2n}), j\rangle$ and $\langle B_1 \to c^m(Y_{11}, \ldots, Y_{1m}), B_2 \to c^m(Y_{21}, \ldots, Y_{2m}), k\rangle$ where B_1 is X_{1i} and B_2 is X_{2i}. By the assumption, $\langle X_{1i}, X_{2i}, q\rangle$ is placed in *to-test*, and, by Proposition 3, will be examined. When it is examined, $\langle Y_{1j}, Y_{2j}, q'\rangle$ is placed in *to-test* if $\langle Y_{1j}, Y_{2j}\rangle$ is not in *testing*. If it is in *testing*, then by Proposition 4, the triple must have already been placed in *to-test* and examined earlier.

Theorem 4 *If there is a proper tree derivable from S_1 but not from S_2, then* **Inclusion-r** *returns false.*

Proof: Consider a proper tree t that is derivable from S_1 but is not derivable from S_2. If t can be built from a single S_1 production (i.e. there is a production $S_1 \to c^0$ but no similar production for S_2), then the algorithm will return false on the first iteration of its while loop.

If t cannot be built from a single S_1 production, then build a path by choosing an S_1 production, $S_1 \to c^n(X_{11}, \ldots, X_{1n})$, such that either there is no production from S_2 to the same constructor, or there is a production $S_2 \to c^n(X_{21}, \ldots, X_{2n})$ where there is an i such that X_{1i} derives a proper tree that X_{2i} cannot derive. Such an S_1 production must exist because S_1 can derive a proper tree that S_2 cannot derive.

The initial path is then $\circ : \langle S_1 \to c^n(X_{11}, \ldots, X_{1n}), S_2 \to c^n(X_{21}, \ldots, X_{2n}), i\rangle$. Lengthen the path as follows. Let the last step of the current path be $\langle A_1 \to c^n(X_{11}, \ldots, X_{1n}), A_2 \to (X_{21}, \ldots, X_{2n}), j\rangle$. If there is a production $X_{1j} \to c^m(Y_{11}, \ldots, Y_{1m})$ such that there is no production from X_{2j} to c^m, then the path is finished and no additional step is added.

Otherwise, choose the productions $X_{1j} \to c^m(Y_{11}, \ldots, Y_{1m})$ and $X_{2j} \to c^m(Y_{21}, \ldots, Y_{2m})$ where there is a k such that Y_{1k} derives a proper tree that Y_{2k} does not derive, and add the step $\langle X_{1j} \to c^m(Y_{11}, \ldots, Y_{1m}), X_{2j} \to c^m(Y_{21}, \ldots, Y_{2m}), k\rangle$ to the end of the path, and continue to add steps to the path.

Now assume that the algorithm returns true. By Proposition 3 and Lemma 5, this means that the algorithm will examine the triple $\langle X_{1i}, X_{2i}, p\rangle$ where $\langle A_1 \to c^n(X_{11}, \ldots, X_{1n}), A_2 \to (X_{21}, \ldots, X_{2n}), i\rangle$ is the last step of the constructed path. But this means that the algorithm returns false because there is a production $X_{1i} \to c^m(Y_{11}, \ldots, Y_{1m})$, but no similar production from X_{2i} to c^m. This contradicts the assumption, so the algorithm must indeed return false.

Theorem 5 *On input grammars G_1 and G_2, the* **Inclusion** *algorithm returns true if and only if* **Inclusion-r** *returns true.*

4 Analysis

The complexity of our algorithms will be measured in terms of:

- m_1 the number of non-terminals in G_1
- m_2 the number of non-terminals in G_2
- r the total number of constructors in G_1 and G_2
- k the maximum arity of any constructor in G_1 and G_2

We will assume that for each non-terminal A of a grammar, the productions from A are stored in a list ordered by constructor. Each list can contain up to r elements. A grammar may have as many as m lists, one for each non-terminal.

4.1 Conversion

The correctness proof for the **Inclusion-r** algorithm requires that only G_1 be proper, therefore the complexity of the **Conversion** algorithm can be expressed in terms of m_1, r, and k.

The body of the outer loop of lines 1-9 is executed m_1 times. The middle loop's body is executed at most times r times on each pass, and the body of the inner loop is done at most k times. A total of $O(m_1 rk)$ time is needed for lines 1-9.

For lines 11-17, the body of the for-each loop is executed m_1 times. On each pass, as many as r productions will be searched to find one using a 0-arity constructor. Thus $O(m_1 r)$ time is used.

Each pass through the loop of lines 18-27 removes one non-terminal from *propagate* resulting in at most m_1 passes. On each pass, the counters for productions that contain the non-terminal are decremented. These counters are for productions in the non-terminal's *rhs-list*. The length of the list can be at most $m_1 rk$, so lines 18-27 require $O(m_1^2 rk)$ time.

The for-each loop at line 28 is iterated for each of the $m_1 r$ productions. At lines 29,30,33, and 34, both examining a non-terminal's mark and removing a production can be done in constant time. The for-each loop at line 32 is done no more than k times. Thus lines 18-25 require $O(m_1 rk)$ time.

The entire algorithm then needs $O(m_1^2 rk)$ time. However, if the number of occurrences of a non-terminal in the right hand side is bounded by a constant, then the algorithm runs in $O(m_1)$ time.

4.2 Inclusion

The nested loops of lines 1-3 use $O(m_1 m_2)$ time to generate every non-terminal pair for initialization.

On each pass through the while loop of lines 5-18, a single non-terminal pair is removed from *to-test* at line 6 meaning that at most $m_1 m_2$ passes are needed. The body of the for-each loop at lines 8-17 is done at most r times. The tests of lines 9 and 10 can be done in constant time by stepping down the list of productions of the pair of non-terminals. The body of the inner for-each loop at lines 11-14 will be done at most k times. The inner loop body itself requires constant time. Thus the lines 5-18 uses $O(m_1 m_2 rk)$ time and the entire algorithm uses $O(m_1 m_2 rk)$ time.

Combining the running time of the **Conversion** algorithm with the running time of the **Inclusion** algorithm results in $O(m_1(m_1 + m_2)rk)$ time being required to compare two deterministic regular tree grammars.

References

[GS84] Ferenc Gecseg and Magnus Steinb. *Tree Automata.* Akademiai Kiado, Budapest, 1984.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesley, 1979.

[Liu98] Y. A. Liu. Dependence analysis for recursive data. In *Proceedings of the 1998 IEEE International Conference on Computer Languages*, Chicago, 1998. IEEE Computer Society Press.