Inclusion Algorithms for One-Unambiguous Regular Expressions and Their Applications

Haiming Chen[a], Zhiwu Xu[b]

[a]*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing* 100190*, China*
[b]*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen* 518060*, China*

Abstract

One-unambiguous regular expressions are used in DTD and XML Schema. It is known that inclusion for one-unambiguous regular expressions is in PTIME. However, there has been few studies on algorithms for the inclusion. In this paper we present algorithms for checking inclusion of one-unambiguous regular expressions. A classical way is based on automata, following which one algorithm is provided and improvements are given. The other algorithm is based on derivatives, utilizing a property presented here that the number of derivatives of a one-unambiguous regular expression is finite. We have applied the algorithms to XML typechecking. The results of experiments with the algorithms are also included. First we give comparisons of the efficiency of our algorithms by experiments. Since after our work Hovland has given another algorithm, we also included his algorithm in the experiments. The results show that both of our algorithms are more efficient than Hovland's algorithm for one-unambiguous regular expressions, and under the inclusion mode (see Section 6) the derivative-based algorithm is more efficient than the automata-based one for small expressions, while for large expressions the latter is more efficient. Then we have conducted preliminary experiments by implementing typechecking of XML using the algorithms. The results show that typechecking using our algorithms is more efficient than typechecking using XDuce. Comparisons of the algorithms with CDuce are also given.

Keywords: One-unambiguous regular expressions, inclusion, algorithms, applications

1. Introduction

Extensible Markup Language (XML) is a simple, very flexible text format for semistructured data, which is popular for the Web and other applications. Usually in XML applications data are provided with schemas that the XML data must conform to. These schemas are very helpful for XML manipulation. In many tasks it is required to check inclusion of schemas (*i.e.*, the set of XML documents satisfying one schema is contained in the set of documents satisfying the other one), for example, in query processing, schema update, typechecking, and so on. In many cases, the inclusion problem of XML schemas is closely related to the inclusion problem of regular expressions [1]. Thus it is useful to study the inclusion problem of regular expressions used in XML schemas.

Many results for the complexity of the inclusion problem for regular expressions exist. For traditional regular expressions, inclusion is PSPACE-complete [2]. Martens et al. [1] give complexity of decision problems for CHAin Regular Expressions (CHAREs) occurring in practice in XML schemas. Their results show that inclusion is already co-NP-complete for very innocent expressions such as expressions with factors of the form a or a^*. Several researchers give complexity for regular expressions with interleaving and/or numerical occurrence indicators [3, 4, 5]. Inclusion for these expressions is in EXPSPACE. In order to get tractable inclusion checking, Suzuki [6] proposes a polynomial-time algorithm for solving a subproblem of the inclusion problem defined by edit operations.

Email addresses: chm@ios.ac.cn (Haiming Chen), xuzhiwu@szu.edu.cn (Zhiwu XU)

The most commonly used XML schema languages are Document Type Definition (DTD) and XML Schema which are recommended by W3C [7, 8]. In these scheme languages, *One-unambiguous regular expressions* [9] are used. Therefore, algorithms for one-unambiguous regular expressions are useful in practice. And the complexity of inclusion problem for these scheme languages reduces to the complexity of inclusion problem of one-unambiguous regular expressions [1]. One-unambiguous regular expressions reflect the requirement that a symbol in the input word is matched uniquely to a position in a regular expression without looking ahead in the word. As one-unambiguous regular expressions can be transformed to deterministic finite automata (DFAs) in polynomial time, inclusion for one-unambiguous regular expressions is in PTIME. However, there has been few studies on algorithms for the inclusion for this kind of regular expressions in the literature before us. In addition, there are some suggestions on using some other regular expressions instead of one-unambiguous ones in these scheme languages, mainly because the latter is not a syntactic concept. But the complexity of inclusion problem will probably become higher, and here we will focus on one-unambiguous regular expressions.

In this paper, we first present two inclusion algorithms for one-unambiguous regular expressions. A classical way to solve this inclusion problem is based on automata, namely, to convert the one-unambiguous regular expressions into automata and then compare the automata. Brüggemann-Klein and Wood [9] have shown that the Glushkov automaton [10, 11] for a one-unambiguous regular expression is deterministic. We have already shown in [12] that for a one-unambiguous regular expression in star normal form, the equation automaton [13] is deterministic, and the Brzozowski's deterministic automaton [14] can be easily computed. Hence any one of the DFAs above can be used in the algorithm. Two improvements on the basic algorithm are also given: (1) we first consider only the reachable states from the start state (*i.e.*, the trim automata), which could improve a little both the time and space complexities (see Table 1); (2) instead of constructing the product automaton, we employ a M_{E_1}-directed search (see Section 3.2 for more detail), which could improve the space complexities. Moreover, although the complexities are still PTIME, both improvements enable us to reduce the time and space requirements in practice. Another algorithm is based on derivatives [14]. For one-unambiguous regular expressions, we give an equivalent calculation of derivatives, and show that the number of derivatives is finite, while this may be infinite for general regular expressions. Then an algorithm for inclusion is given, which repeatedly calculates the derivatives of the expressions and will give an answer in this process. Table 1 summaries the complexities of our inclusion algorithms with the assumption that $\Sigma_{E_1} \subseteq \Sigma_{E_2}$, where Trim-Automata and M_{E_1}-Directed denote two improved versions of the automata-based algorithm, and $|E|, ||E||$ and Σ_E denote the size, width and alphabet of E, respectively (see Section 2 for more detail).

Table 1: Complexities of the inclusion algorithms on $L(E_1) \subseteq L(E_2)$.

	Time	Space
Automata-Based	$O(\|E_1\| \cdot \|E_2\| \cdot \|\Sigma_{E_2}\|)$	$O(\|E_1\| \cdot \|E_2\| \cdot \|\Sigma_{E_2}\|)$
Trim-Automata	$O(\|E_1\| \cdot \|E_2\| \cdot \|\Sigma_{E_1}\|)$	$O(\|E_1\| \cdot \|E_2\| \cdot \|\Sigma_{E_1}\|)$
M_{E_1}-Directed	$O(\|E_1\| \cdot \|E_2\| \cdot \|\Sigma_{E_1}\|)$	$O(\|E_1\| \cdot \|\Sigma_{E_1}\| + \|E_2\| \cdot \|\Sigma_{E_2}\|)$
Derivative-Based	$O((\|E_1\|^2 + \|E_2\|^2)\|E_1\|^2\|E_2\|^2)$	$O(\|E_1\|^2\|E_1\| + \|E_2\|^2\|E_2\|)$

Then we briefly present the applications of our algorithms to one of the tasks mentioned above, that is, XML typechecking. XML typechecking is one important issue of XML processing, in which XML schema languages are regarded as types and subtype relations (*i.e.* inclusion relations) are checked at compile-time to ensure type correctness of programs. In particular, we developed typechecking methods for DTDs and for regular tree grammars with disjoint production rules (RRTGs). Investigation reveals that 13.5% XML documents in practice contains a reference to a DTD [15], so typechecking methods for DTDs are useful in practice. For those XML schemas that are not DTDs, we can use the typechecking method for RRTGs.

At last we conduct some experiments with the algorithms. We first conduct some experiments to evaluate the efficiency of our algorithms. We also compare our algorithms with Hovland's algorithm [16], which is guaranteed to decide the inclusion problem if the second expression E_2 is one-unambiguous, and might either

decide the problem correctly or report that the one-ambiguity is a problem if E_2 is one-ambiguous. The results show that both of our algorithms are more efficient than Hovland's algorithm for one-unambiguous regular expressions, and under the inclusion mode (see Section 6) the derivative-based algorithm is more efficient than the automata-based one for small expressions, while for large expressions the latter is more efficient. Finally, we conduct some experiments to compare our XML typechecking algorithms, where our inclusion algorithms are used, with XDuce [17], an XML processing language which supports regular tree languages as schemas, on XML typechecking. The results show that both our typechecking algorithms are more efficient than XDuce. Comparisons of the algorithms with CDuce [18] are also given.

Extension statement: Part of the preliminary work of the above has been published in the Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC'08) [19]. This paper further contains largely improved representation with more contents, all proofs and our new recent results, which include applications of our algorithms onto XML typechecking, the comparison of the inclusion algorithms, and the experiments to compare our XML typechecking algorithms, where our inclusion algorithms are applied, with XDuce and CDuce.

Section 2 introduces notation required in the paper. Section 3 presents the automata-based algorithm. Section 4 gives the derivative-based algorithm. Section 5 briefly introduces the applications of our algorithms. Sections 6 and 7 describe the experiments for inclusion algorithms and their applications on XML typechecking respectively. Section 8 contains related work. Section 9 gives concluding remarks.

2. Notation

We assume the readers are familiar with basic regular languages and automata theory [20], so we introduce here only some notation used later in the paper.

2.1. Regular Expressions

Let Σ be an alphabet of symbols and ε denote the empty word. $|\Sigma|$ denotes the size of Σ, and Σ^* is the set of all words over Σ. A regular expression E over Σ is $\emptyset, \varepsilon$ or $a \in \Sigma$, or the union $E_1 + E_2$, the concatenation E_1E_2 or the star E_1^*, where E_1 and E_2 are two regular expressions. For a regular expression E, the language specified by E is denoted by $L(E)$. The size of E, denoted by $|E|$, is the length of E when written in postfix (parentheses are not counted); the (alphabetic) width of E, denoted by $\|E\|$, is the number of symbols occurring in E; and the (smallest) alphabet of E, denoted by Σ_E, is the set of symbols that occur in E.

For a regular expression E over Σ and a symbol $a \in \Sigma$, we define the following sets:

$first(E) = \{b \mid bw \in L(E), b \in \Sigma, w \in \Sigma^*\}$
$last(E) = \{b \mid wb \in L(E), w \in \Sigma^*, b \in \Sigma\}$
$follow(E, a) = \{b \mid uabv \in L(E), u, v \in \Sigma^*, b \in \Sigma\}$
$followlast(E) = \{b \mid vbw \in L(E), v \in L(E), v \neq \varepsilon, b \in \Sigma, w \in \Sigma^*\}$

2.2. One-unambiguous Regular Expressions

One-unambiguous regular expressions are also called deterministic regular expressions, which came from Brüggemann-Klein and Wood [9].

For a regular expression we can mark symbols with subscripts so that in the marked expression each marked symbol occurs only once. For example $(a_1 + b_1)^*a_2b_2(a_3 + b_3)$ is a marking of the expression $(a + b)^*ab(a + b)$. A marking of an expression E is denoted by $E^\sharp$. The reverse of marking is the dropping of subscripts from the marked symbols, denoted by $^\natural$. Then we have $(E^\sharp)^\natural = E$. We extend the notation for words and automata in an obvious way. One-unambiguous expressions are defined as follows:

Definition 2.1. *An expression E is one-unambiguous if and only if, for all words $uxv, uyw \in L(E^\sharp)$ s.t. $|x| = |y| = 1$, if $x \neq y$ then $x^\natural \neq y^\natural$. A regular language is one-unambiguous if it is denoted by some one-unambiguous expression.*

For example, a^*aa^* is not one-unambiguous, since there are two words $a_1a_2, a_2a_3 \in L(a_1^*a_2a_3^*)$ such that $a_1 \neq a_2$ but $a_1^\natural = a_2^\natural$. While aa^* is one-unambiguous, which describes the same language as a^*aa^*.

As shown by Brüggemann-Klein [21], we have the following result.

Proposition 2.1. *It can be decided in linear time whether a regular expression is one-unambiguous.*

2.3. Glushkov Automaton and Star Normal Form

The Glushkov automaton (or position automaton) is introduced independently by Glushkov [10] and McNaughton and Yamada [11].

Definition 2.2. *Given a regular expression E, we define its Glushkov automaton M_E as a 5-tuple $(Q_E, \Sigma, \delta_E, q_E, F_E)$, where:*

(1). $Q_E = \Sigma_{E^\sharp} \cup \{q_E\}$

(2). $\delta_E(q_E, a) = \{x \mid x \in first(E^\sharp), x^\natural = a\}$, *for* $a \in \Sigma$

(3). $\delta_E(x, a) = \{y \mid y \in follow(E^\sharp, x), y^\natural = a\}$, *for* $x \in \Sigma_{E^\sharp}$ *and* $a \in \Sigma$

(4). $F_E = \begin{cases} last(E^\sharp) \cup \{q_E\}, & \text{if } \varepsilon \in L(E), \\ last(E^\sharp), & \text{otherwise} \end{cases}$

Example 2.1. *The Glushkov automaton M_{E_1} of the regular expression $E_1 = (ab(c+\varepsilon))^*$ is in Figure 1, where a_1^1, b_2^1, c_3^1 denote the mark symbols a_1, b_2, c_3 in $E_1^\sharp$ respectively.*

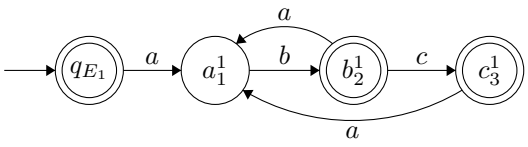


Figure 1: The Glushkov automaton of $(ab(c+\varepsilon))^*$.

Let $L(M)$ denote the language accepted by the automaton M. As shown by Glushkov [10], McNaughton and Yamada [11] and Brüggemann-Klein and Wood [9], we have the following properties.

Proposition 2.2. $L(M_E) = L(E)$.

Proposition 2.3. *A regular expression E is one-unambiguous if and only if M_E is deterministic.*

A naive algorithm to compute Glushkov automata takes time cubic in the size of the expression. Brüggemann-Klein [21] gave a quadratic time algorithm for regular expressions in star normal form, which takes linear time for one-unambiguous regular expressions.

Star normal forms of regular expressions are defined as follows [21]:

Definition 2.3. *A regular expression E is in star normal form if, for each starred subexpression H^* of E, $followlast(H^\sharp) \cap first(H^\sharp) = \emptyset$ and $\varepsilon \notin L(H)$.*

For example, $(a^*b^*)^*$ is not in star normal form, while $(a+b)^*$ is in star normal form, although they describe the same language.

Brüggemann-Klein and Wood [21] have proved any regular expression can be transformed into star normal form in linear time.

Proposition 2.4. *Given a regular expression E, it can be transformed into star normal form in linear time.*

2.4. Derivatives

Derivatives of regular expressions were introduced by Brzozowski [14], which are defined as follows:

Definition 2.4. *Given a regular expression E and a symbol a, the derivative $d_a(E)$ of E with respect to a is defined inductively as follows:*

$$\begin{array}{rcl} d_a(\emptyset) & = & d_a(\varepsilon) = \emptyset \\ d_a(b) & = & \begin{cases} \varepsilon, & \text{if } b = a \\ \emptyset, & \text{otherwise} \end{cases} \\ d_a(F+G) & = & d_a(F) + d_a(G) \\ d_a(FG) & = & \begin{cases} d_a(F)G + d_a(G), & \text{if } \varepsilon \in L(F) \\ d_a(F)G, & \text{otherwise} \end{cases} \\ d_a(F^*) & = & d_a(F)F^* \end{array}$$

The derivatives can be extended with respect to a word: $d_\varepsilon(E) = E$ and $d_{wa}(E) = d_a(d_w(E))$. Taking E_1 from Example 2.1 for example, $d_a(E_1) = b(c+\varepsilon)(ab(c+\varepsilon))^*$ and $d_{ab}(E_1) = (c+\varepsilon)(ab(c+\varepsilon))^*$.

Given a regular expression E, the set of all the derivatives of E is denoted by $D(E)$. In general, $D(E)$ may be infinite without some reductions by similarity. While under similarity, there exists a finite number of derivatives in $D(E)$, which are called the characteristic derivatives of E [14]. And an equivalent automaton can be constructed from the characteristic derivative set [14].

Proposition 2.5. *Given a regular expression E on Σ and a finite set $D_c(E)$ of characteristic derivatives for E, let M be the automaton $(D_c(E), \Sigma, \delta, E, F)$, where $F = \{\epsilon \in L(E_c) \mid E_c \in D_c(E)\}$ and $\delta(E_c, a) = d_a(E_c)$. Then $L(M) = L(E)$.*

3. Automata-Based Algorithm

As one-unambiguous regular expressions can be directly converted to DFAs [9, 12], we give a basic algorithm for one-unambiguous regular expressions based on automata in this section.

3.1. The Algorithm

A classical way to check inclusion of regular expressions is to convert the regular expressions into automata and then compare the automata. Brüggemann-Klein and Wood [9] have shown that the Glushkov automaton [10, 11] for a one-unambiguous regular expression is deterministic. We have already shown in [12] that the equation automaton [13] for a one-unambiguous regular expression in star normal form is deterministic, and the Brzozowski's deterministic automaton [14] can also be easily computed for a one-unambiguous regular expression in star normal form. Hence any one of the Glushkov DFA, Brzozowski's DFA, or equation DFA can be used in the algorithm. As an example, here we just show how Glushkov DFA is used in the algorithm. The other two types of the DFAs above are similar.

Given two one-unambiguous regular expressions E_1 and E_2, our aim is to determine whether the inclusion relation $L(E_1) \subseteq L(E_2)$ holds or not. As $\Sigma_{E_1} \not\subseteq \Sigma_{E_2}$ implies $L(E_1) \not\subseteq L(E_2)$, we assume $\Sigma_{E_1} \subseteq \Sigma_{E_2}$ in the following. To do that, we first convert the expressions into Glushkov automata, and then check the inclusion of the automata. This yields an automata-based algorithm, which consists of the following four steps:

Step (1): construct the automata M_{E_1} and M_{E_2} from the expressions E_1 and E_2 respectively.
The Glushkov DFA can be computed by the algorithm in [21]. Let $M_{E_1} = (Q_{E_1}, \Sigma_{E_1}, \delta_{E_1}, q_{E_1}, F_{E_1})$ with $\|E_1\| + 1$ states, and $M_{E_2} = (Q_{E_2}, \Sigma_{E_2}, \delta_{E_2}, q_{E_2}, F_{E_2})$ with $\|E_2\| + 1$ states.

Step (2): compute an automaton M' for the complement of $L(M_{E_2})$.
We first make M_{E_2} complete if it is not. Suppose the complete one is $M'_{E_2} = (Q'_{E_2}, \Sigma_{E_2}, \delta'_{E_2}, q_{E_2}, F_{E_2})$, then let $M' = (Q'_{E_2}, \Sigma_{E_2}, \delta'_{E_2}, q_{E_2}, Q'_{E_2} - F_{E_2})$. It is clear that $L(M') = \overline{L(M'_{E_2})} = \overline{L(M_{E_2})}$.

Step (3): construct an automaton B such that $L(B) = L(M_{E_1}) \cap L(M')$.
The automaton can be constructed by product, that is, $B = (Q_{E_1} \times Q'_{E_2}, \Sigma, \delta, (q_{E_1}, q_{E_2}), F_{E_1} \times (Q'_{E_2} - F_{E_2}))$, where $\Sigma = \Sigma_{E_1} \cup \Sigma_{E_2}$ and $\delta((p,q),a) = (\delta_{E_1}(p,a), \delta'_{E_2}(q,a))$.

Step (4): check whether $L(B) = \emptyset$ holds or not. If it does, the algorithm returns TRUE, otherwise FALSE. This can be solved by a search on the graph $G = (Q, E)$, where $Q = Q_{E_1} \times Q'_{E_2}$, E is the set of transitions of B. If there is a path from the start state to an accepting state, then $L(B)$ is not empty.

Clearly, the overall process implements checking $L(E_1) \cap \overline{L(E_2)} = \emptyset$. So the algorithm is correct.

Theorem 3.1. *Given two one-unambiguous regular expressions E_1 and E_2, the algorithm returns TRUE if and only if $L(E_1) \subseteq L(E_2)$.*

Proof. The correctness of **Step (1)** is ensured by Proposition 2.1 in [21] and **Steps (2)-(4)** are straightforward. □

Example 3.1. *Let E_1 be the expression from Example 2.1 and E_2 be $(abc^*)^*$. The constructed Glushkov automaton M_{E_1} is shown in Figure 1, and the other computations of the algorithm is shown in Figure 2, where q_t denotes the trap state of M'_{E_2}, q_0, q_1, q_2 and q_3 denote the pair states (q_{E_1}, q_{E_2}), (a_1^1, a_1^2), (b_2^1, b_2^2) and (c_3^1, c_3^2) respectively. It is clear that $L(B) = \emptyset$.*

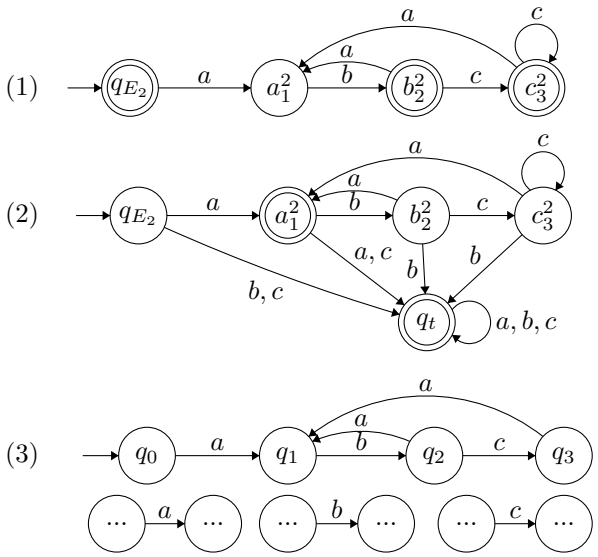


Figure 2: (1) Glushkov automaton of $(abc^*)^*$; (2) Automaton M'; (3) Main parts of automaton B.

Consider the complexity of the algorithm. In **Step (1)**, the computation of M_{E_1} and M_{E_2} can be done in $O(|E_1|)$ and $O(|E_2|)$ time respectively [21]. In **Step (2)**, the computation of M'_{E_2} can be done in $O(|Q_{E_2}||\Sigma_{E_2}|) = O((\|E_2\| + 1)|\Sigma_{E_2}|)$ time and the construction of M' is in linear time. In **Step (3)**, the construction of B can be computed in $O((||E_1|| + 1)(||E_2|| + 2)(|\Sigma_{E_1} \cup \Sigma_{E_2}|)) = O(||E_1|| \cdot ||E_2|| \cdot |\Sigma_{E_1} \cup \Sigma_{E_2}|)$ time. Finally, let us consider **Step (4)**. For a DFA B, $|E| \leq |Q| \cdot |\Sigma_{E_1} \cup \Sigma_{E_2}|$. It is known that a search on G can be done in $O(|Q| + |E|)$ time ([22, P. 534]). Therefore the time complexity is $O(|Q| + |E|) = O(\|E_1\| \cdot \|E_2\| \cdot |\Sigma_{E_1} \cup \Sigma_{E_2}|)$. The running time of the overall computation is $O(\|E_1\| \cdot \|E_2\| \cdot |\Sigma_{E_1} \cup \Sigma_{E_2}|)$. By the assumption of $\Sigma_{E_1} \subseteq \Sigma_{E_2}$, $O(\|E_1\| \cdot \|E_2\| \cdot |\Sigma_{E_1} \cup \Sigma_{E_2}|) = O(\|E_1\| \cdot \|E_2\| \cdot |\Sigma_{E_2}|)$. Moreover, the space required by the algorithm is $O(\|E_1\| \cdot \|E_2\| \cdot |\Sigma_{E_2}|)$ as well.

3.2. Improvements

In this section, we present some improvements for the algorithm above, by using properties of the automata, such as the valid paths ending with any accepting state.

For an automaton $M = (Q, \Sigma, \delta, q_0, F)$, a path in the automaton M is defined as a sequence of the form $q_1 a_1 q_2 a_2 \dots q_n a_n q_{n+1}$, where $q_i \in Q$, $a_i \in \Sigma$, $n \geq 1$ and $\delta(q_i, a_i) = q_{i+1}$. Given a path p in the product

automaton B constructed in **Step (3)**, *i.e.*, $p = (q_1^1, q_1^2)a_1 \dots (q_n^1, q_n^2)a_n(q_{n+1}^1, q_{n+1}^2)$, we write $p[(q_i^1, q_i^2)\backslash q_i^1]$ for the path obtained from p by replacing every state (q_i^1, q_i^2) in p by q_i^1, $i = 1, \dots, n+1$. We also extend this notation to a path set P from B such that $P[(q_i^1, q_i^2)\backslash q_i^1] = \{p[(q_i^1, q_i^2)\backslash q_i^1] \mid p \in P\}$. As the automaton M' in **Step (2)** is complete and deterministic and, by assumption, $\Sigma_{E_1} \subseteq \Sigma_{E_2}$, we have the following two properties.

Property 3.2. *Let $P_1 = \{p \mid p$ is a path starting from (q_{E_1}, q_{E_2}) in $B\}$, $P_2 = \{p_2 \mid p_2$ is a path starting from q_{E_1} in $M_{E_1}\}$, then $P_1[(q_i^1, q_i^2)\backslash q_i^1] = P_2$, and $|P_1| = |P_2|$.*

Proof. Given $p \in P_1$, by the definition of B, $p[(q_i^1, q_i^2)\backslash q_i^1] \in P_2$. Given $p_2 \in P_2$, since M' is complete and $\Sigma_{E_1} \subseteq \Sigma_{E_2}$, there exists a path $p \in P_1$ such that $p[(q_i^1, q_i^2)\backslash q_i^1] = p_2$. So $P_1[(q_i^1, q_i^2)\backslash q_i^1] = P_2$. As M' is deterministic, the number of paths in B could not be more than the number of paths in M_{E_1}. On the other hand, due to the fact that M' is complete and $\Sigma_{E_1} \subseteq \Sigma_{E_2}$, the number of paths in B could not be less than the number of paths in M_{E_1}. □

Property 3.3. *Let $P_1 = \{p \mid p$ is a path starting from (q_{E_1}, q_{E_2}) and ending with an accepting state in $B\}$, $P_2 = \{p_2 \mid p_2$ is a path starting from q_{E_1} and ending with an accepting state in $M_{E_1}\}$, then $P_1[(q_i^1, q_i^2)\backslash q_i^1] \subseteq P_2$, and $|P_1| \leq |P_2|$.*

Proof. It follows from Property 3.2 and the fact of $F_{E_1} \times (Q'_{E_2} - F_{E_2}) \subseteq F_{E_1} \times Q'_{E_2}$. □

An automaton is called a trim one if it only contains states reachable from the start state. As illustrated in Example 3.1, the product automaton B constructed in **Step (3)** might contain some unreachable states, that is, B might be non-trim. Following the properties above, the trim version of B can be constructed directly with the guide of M_{E_1}, which thus improve the inclusion algorithm. The product construction algorithm is shown as follows:

$Q = \{(q_{E_1}, q_{E_2})\}$
for $(p, q) \in Q$ and $a \in \Sigma$ s.t. (p, q) is unmarked and $\delta_{E_1}(p, a)$ is defined **do**
  $\delta((p, q), a) = (\delta_{E_1}(p, a), \delta'_{E_2}(q, a))$
  $Q = Q \cup \{(\delta_{E_1}(p, a), \delta'_{E_2}(q, a))\}$
  mark (p, q) in Q
end for

Let the automaton constructed by the product construction algorithm denote as B_c. This construction is more efficient than the previous construction of B, since only the reachable paths are considered during construction.

Lemma 3.4. *The automaton B_c constructed by the product construction algorithm is trim.*

Proof. According to Property 3.2, there exists a one-to-one correspondence between the reachable paths in B and M_{E_1}. This enables us to construct B_c with respect to the reachable paths in M_{E_1}. □

Example 3.2. *The trim automaton B_c of Example 3.1 is shown in Figure 3, which is smaller than B, where q_i's are the same to those in B.*

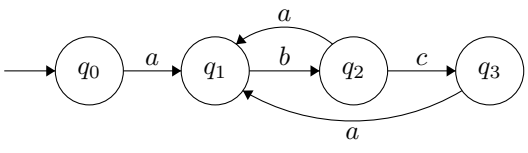


Figure 3: The trim automaton B_c of Example 3.1.

It is clear $L(B) = L(B_c)$, following Property 3.3. Moreover, if B_c does not contain any accepting state, then $L(B_c) = \emptyset$, otherwise $L(B_c) \neq \emptyset$. Therefore, to check $L(B) = \emptyset$ can be simply reduced to checking

whether B_c contains an accepting state or not. For that, we can further improve **Steps (3)** and **(4)** by a M_{E_1}-directed search on M_{E_1} and M', without constructing the automaton B_c for the intersection, which is shown as follows:

$Q = \{(q_{E_1}, q_{E_2})\}$
for $(p, q) \in Q$ and $a \in \Sigma$ s.t. (p, q) is unmarked and $\delta_{E_1}(p, a)$ is defined **do**
  if both $\delta_{E_1}(p, a)$ and $\delta'_{E_2}(q, a))$ are accepting states **then**
    return FALSE
  else
    $Q = Q \cup \{(\delta_{E_1}(p, a), \delta'_{E_2}(q, a))\}$
    mark (p, q) in Q
  end if
end for
return TRUE

During the search, if there exists an accepting state, then it stops immediately and returns FALSE. Only if there are no accepting states, a complete search is done, which is equivalent to the construction of B_c. Therefore, it has the same time complexity as the one using B_c if $L(B_c) = \emptyset$, while is more efficient otherwise. Besides, it is more efficient in space since B_c is not constructed fully.

Example 3.3. *Let E_1 and E_2 be the expressions from Example 3.1 and let us check $L(E_2) \subseteq L(E_1)$ instead. The trim automaton B_c for this check is shown in Figure 4, where q_i's are the same as those in B, q_1^t, q_2^t and q_3^t denotes the pair state (a_1^2, q_t), (b_2^2, q_t) and (c_3^2, q_t) respectively and q_t is the trap state of M'_{E_1}. Guided by the directed search, the algorithm will stop with FALSE once it reaches the accepting state q_3^t, without further visiting.*

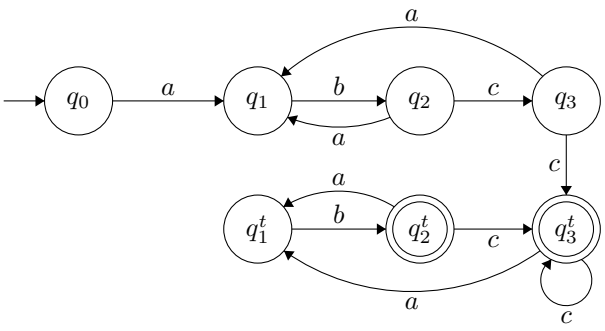


Figure 4: The trim automaton B_c of Example 3.3.

Consider the complexity of the algorithm using B_c. In **Step (3)**, the construction of B_c can be computed in $O((||E_1|| + 1)(||E_2|| + 2)|\Sigma_{E_1}|) = O(||E_1|| \cdot ||E_2|| \cdot |\Sigma_{E_1}|)$ time. In **Step (4)**, to check the existence of an accepting state is in linear time. Indeed, the accepting states have been marked in the construction of automata. If in **Step (3)** we use a variable to keep this information, then the existence of an accepting state is already known and the checking is done clearly in constant time. Therefore, the running time of the overall computation is $O(\|E_1\| \cdot \|E_2\| \cdot |\Sigma_{E_1}|)$. The space complexity is $O(\|E_1\| \cdot \|E_2\| \cdot |\Sigma_{E_1}|)$.

For the M_{E_1}-directed search algorithm, the time complexity is $O(\|E_1\| \cdot \|E_2\| \cdot |\Sigma_{E_1}|)$, but the space complexity is $O(\|E_1\| \cdot |\Sigma_{E_1}| + \|E_2\| \cdot |\Sigma_{E_2}|)$.

In addition, we can further improve the algorithm by adopting the technique of *up to congruence* [23], which can help us to find the equivalent states to reduce the DFAs, at the cost of the equivalence computations.

In the following, we use the automata-based algorithm to denote the one using Glushkov DFA with M_{E_1}-directed search, unless it is explicitly specified.

4. Derivative-Based Algorithm

In this section, we present an algorithm based on the derivatives to determine the inclusion of two one-unambiguous regular expressions. The idea is that if an expression is included in another expression then all the derivatives of the first expression are included in those of the second one, which is inspired by Brzozowski's work [24] and Antimirov's work [25].

As shown by Brüggemann-Klein and Wood [9], the derivatives of a one-unambiguous regular expression in star normal form are also one-unambiguous regular expressions in star normal form. This yields a characterization of one-unambiguity: for each regular expression E in star normal form, one-unambiguity of E can be characterized solely in terms of the sub-expressions of E.

Proposition 4.1 ([9])**.** *Let E be a regular expression in star normal form.*

- *$E = \emptyset, E = \varepsilon$, or $E = a \in \Sigma$: E is one-unambiguous.*
- *$E = F + G$: E is one-unambiguous if and only if F and G are one-unambiguous and $first(F) \cap first(G) = \emptyset$.*
- *$E = FG$:*
  - *(i) If $L(E) = \emptyset$, then E is one-unambiguous.*
  - *(ii) If $L(E) \neq \emptyset$ and $\varepsilon \in L(F)$, then E is one-unambiguous if and only if F and G are one-unambiguous, $first(F) \cap first(G) = \emptyset$, and $followlast(F) \cap first(G) = \emptyset$.*
  - *(iii) If $L(E) \neq \emptyset$ and $\varepsilon \notin L(F)$, then E is one-unambiguous if and only if F and G are one-unambiguous and $followlast(F) \cap first(G) = \emptyset$.*
- *$E = F^*$: E is one-unambiguous if and only if F is one-unambiguous and $followlast(F) \cap first(F) = \emptyset$.*

When one-unambiguous expressions are concerned, we can have the following simpler construction of derivatives.

Proposition 4.2. *For a one-unambiguous regular expression E in star normal form, the derivative of E by a symbol a can be computed as follows:*

$$
\begin{aligned}
d_a(\emptyset) &= d_a(\varepsilon) = \emptyset \\
d_a(b) &= \begin{cases} \varepsilon, & \text{if } b = a \\ \emptyset, & \text{otherwise} \end{cases} \\
d_a(F+G) &= \begin{cases} d_a(F), & \text{if } a \in first(F) \\ d_a(G), & \text{if } a \in first(G) \\ \emptyset, & \text{otherwise} \end{cases} \\
d_a(FG) &= \begin{cases} d_a(F)G, & \text{if } a \in first(F) \\ d_a(G), & \text{if } a \in first(G) \text{ and } \varepsilon \in L(F) \\ \emptyset, & \text{otherwise} \end{cases} \\
d_a(F^*) &= d_a(F)F^*
\end{aligned}
$$

Proof. We only need to prove $E = F + G$ or FG.

Assume $E = F + G$, then $first(F) \cap first(G) = \emptyset$ by Proposition 4.1. So if $a \in first(F)$ then $a \notin first(G)$, and thus $d_a(G) = \emptyset$, $d_a(E) = d_a(F)$. The same to $a \in first(G)$. If both $a \notin first(F)$ and $a \notin first(G)$, then clearly $d_a(E) = \emptyset$.

Assume $E = FG$, by Definition 2.4, $d_a(E)$ is $d_a(F)G + d_a(G)$ if $\varepsilon \in L(F)$, or $d_a(F)G$ otherwise. Here we just show $first(F) \cap first(G) = \emptyset$, the remaining proof is similar to that of the above for $E = F + G$. If $L(E) = \emptyset$, obviously $first(F) \cap first(G) = \emptyset$. Otherwise (*i.e.*,$L(E) \neq \emptyset$), according to Proposition 4.1, $first(F) \cap first(G) = \emptyset$ holds as well. □

Compared with the standard definition of general regular expressions, the main difference lies in the treatment of union and concatenation. For example, $d_a(\varepsilon a^*) = d_a(a^*) = d_a(a)a^* = \varepsilon a^*$. While with respect to the standard definition, we would get $d_a(\varepsilon a^*) = d_a(\varepsilon)a^* + d_a(a^*) = \emptyset a^* + \varepsilon a^*$. Moreover, in the case of one-unambiguous regular expressions, the number of derivatives is finite, as shown below.

Theorem 4.3. *For a one-unambiguous regular expression E in star normal form, the cardinality of the set $D(E)$ of derivatives is less than or equal to $\|E\| + 1$.*

Proof. As $d_\varepsilon(E) = E$, we only need to prove that the number of derivatives of E with respect to non-empty words, denoted $nd(E)$, is less than or equal to $\|E\|$. We prove this by induction.

Base. If $E = \emptyset, \varepsilon$, or $a \in \Sigma$, the above statement holds trivially.

Induction. (i). $E = F + G$. It is easily verified from definitions in Proposition 4.2 that, for a non-empty word $w \in \Sigma^+$, the derivatives of E is (conditions of rhs are omitted in the sequel of the proof)

$$d_w(F+G) = \begin{cases} d_w(F) \\ d_w(G) \\ \emptyset \end{cases}$$

By induction, $nd(F) \leq \|F\|$ and $nd(G) \leq \|G\|$. Hence $nd(F+G) \leq nd(F) + nd(G) \leq \|F\| + \|G\| = \|E\|$.

(ii). $E = FG$. Using the definitions in Proposition 4.2, we have

$$d_w(FG) = \begin{cases} d_w(F)G \\ d_v(G) \ \text{ for all } v \in \Sigma^+ \text{ s.t. } w = uv, u \in L(F) \\ \emptyset \end{cases}$$

By induction, $nd(F) \leq \|F\|$ and $nd(G) \leq \|G\|$. So $nd(FG) \leq nd(F) + nd(G) \leq \|F\| + \|G\| = \|E\|$.

(iii). $E = F^*$. Similarly, we have

$$d_w(F^*) = d_v(F)F^* \text{ for all } v \in \Sigma^+ \text{ s.t. } w = uv, u \in \Sigma^*$$

By induction, $nd(F) \leq \|F\|$. Therefore $nd(F*) \leq nd(F) \leq \|F\| = \|E\|$. This concludes the inductive step. □

Example 4.1. *Let E_1 be the expressions from Example 2.1. The derivative set $D(E_1)$ of E_1 is $\{E_1, b(c+\varepsilon)(ab(c+\varepsilon))^*, (c+\varepsilon)(ab(c+\varepsilon))^*\}$.*

These two properties above enable us to construct an equivalent automaton for a one-unambiguous regular expression E in star normal form directly from its finite derivative set, rather than a characteristic derivative set such as Brzozowski's *aci-similar* derivative set [14]. More on this automaton are studied in [12].

Proposition 4.4. *([12]) Given a one-unambiguous regular expression E on Σ in star normal form. Let M be the automaton $(D(E), \Sigma, \delta, E, F)$, where $F = \{\epsilon \in L(E_0) \mid E_0 \in D(E)\}$ and $\delta(E_0, a) = d_a(E_0)$. Then $L(M) = L(E)$.*

With the derivative automaton in mind, we present a derivative-based inclusion algorithm for one-unambiguous regular expressions. The idea is to reduce the inclusion of two one-unambiguous regular expressions E_1 and E_2 to the inclusion of their derivatives with respect to the first set. As mentioned in Section 2, any regular expression can be converted into star normal form in linear time, so we assume that one-unambiguous regular expressions are in star normal form (if not, the algorithm in [21] is applied first). The detail of the algorithm is shown as follows, where A denotes a set containing expression pairs that have been checked and is set initially to empty.

`include`(E_1, E_2, A)
if $\varepsilon \in L(E_1)$ and $\varepsilon \notin L(E_2)$ **then return** FALSE
if $first(E_1) \not\subseteq first(E_2)$ **then return** FALSE

for $a \in first(E_1)$ **do**
  $t_1 = d_a(E_1)$, $t_2 = d_a(E_2)$,
  if $(t_1, t_2) \notin A$ **then**
  if `include`$(t_1, t_2, A \cup \{E_1, E_2\})$=FALSE **then**
  return FALSE
end for
return TRUE

Indeed, the inclusion checking of E_1 and E_2 can be treated as the emptiness checking for an automaton M recognizing the language $L(E_1) \setminus L(E_2)$, where all the derivative pairs of E_1 and E_2 form the states of M (thanks to the finite number of the derivatives for one-unambiguous regular expressions in star normal form), those states satisfying $first(E_1) \not\subseteq first(E_2)$ are (the ones leading to) the accepting ones and A is the set of reached states that have been visited currently. Therefore, if $L(E_1) \setminus L(E_2)$ is nonempty, then the algorithm `include` will reach an accepting state. This ensures the correctness of the algorithm `include`.

Theorem 4.5. *Given two one-unambiguous regular expressions E_1 and E_2, the algorithm* `include` *returns TRUE if and only if $L(E_1) \subseteq L(E_2)$.*

Proof. Following Theorem 4.3, the cardinality of the derivative set $D(E_i)$ of E_i is finite. Let M_i denote the automaton $(D(E_i), \Sigma_{E_i}, \delta_i, E_i, F_i)$, where $F_i = \{\epsilon \in L(E) \mid E \in D(E_i)\}$ and $\delta_i(E, a) = d_a(E)$ for each $E \in D(E_i)$. By Proposition 4.4, we have $L(M_i) = L(E_i)$. According to automata theory, the algorithm is essentially the emptiness checking for an automaton M representing (a reachable part) of a direct product $M_1 \times \neg M_2$, without the construction of M_1, M_2 and M. That is, the algorithm is to check whether there exists a reachable accepting state in M. Due to the finite number of states, the algorithm terminates. In the remaining we prove the following statement instead:

$$\texttt{include} \text{ returns FALSE} \Longleftrightarrow L(E_1) \not\subseteq L(E_2)$$

$\Longleftarrow$: Assume that there exists a word w such that $w \in L(E_1) \setminus L(E_2)$. Then there is an accepting state (E_1', E_2') for M such that $\delta((E_1, E_2), w) = (E_1', E_2')$, where $E_i' \in D(E_i)$, $\epsilon \in L(E_1')$ and $\epsilon \notin L(E_2')$. Clearly `include` returns FALSE, according to the first IF statement. Moreover, if there exists a state (E_1'', E_2'') among (E_1, E_2) and (E_1', E_2'), that is $\delta((E_1, E_2), w') = (E_1'', E_2'')$, $w'w'' = w$, $|w''| > 0$, satisfying $first(E_1'') \not\subseteq first(E_2'')$, then `include` returns FALSE in advance.

$\Longrightarrow$: If a state (E_1', E_2') satisfying $\epsilon \in L(E_1')$ and $\epsilon \notin L(E_2')$ is reached during the checking, that is, an accepting state of M is reached, then, $L(M)$ is nonempty, which deduces $L(E_1) \not\subseteq L(E_2)$. Otherwise, assume that a state (E_1', E_2') satisfying $first(E_1') \not\subseteq first(E_2')$ is reached during the checking and that $a \in first(E_1') \setminus first(E_2')$, where $E_i' \in D(E_i)$. Then the state $(d_a(E_1'), d_a(E_2'))$, that is $(d_a(E_1'), \emptyset)$, can be reached, and for any $w \in L(d_a(E_1'))$, the state $(d_{aw}(E_1'), \emptyset)$ can be reached as well. Moreover, it is clear that $\epsilon \in L(d_{aw}(E_1'))$ and $\epsilon \notin \emptyset$, that is, the state $(d_{aw}(E_1'), \emptyset)$ is an accepting state of M. Therefore, $L(M)$ is nonempty, which deduces $L(E_1) \not\subseteq L(E_2)$. □

Example 4.2. *Let E_1 and E_2 be the expressions from Example 3.1 and let us check $L(E_1) \subseteq L(E_2)$. Since both E_1 and E_2 are in star normal form, we directly apply the algorithm* `include` *on them:*

$E_1 = (ab(c + \varepsilon))^*$,
$E_2 = (abc^*)^*$,
1: $first(E_1) = first(E_2) = \{a\}$,
$A = \{(E_1, E_2)\}$,
$d_a(E_1) = b(c + \varepsilon)(ab(c + \varepsilon))^* \quad r_{11}$,
$d_a(E_2) = bc^*(abc^*)^* \quad r_{21}$,
2: $first(r_{11}) = first(r_{21}) = \{b\}$,
$A = \{(E_1, E_2), (r_{11}, r_{21})\}$,
$d_b(r_{11}) = (c + \varepsilon)(ab(c + \varepsilon))^* \quad r_{12}$,
$d_b(r_{21}) = c^*(abc^*)^* \quad r_{22}$,
3: $first(r_{12}) = first(r_{22}) = \{a, c\}$,
$A = \{(E_1, E_2), (r_{11}, r_{21}), (r_{12}, r_{22})\}$,

$d_a(r_{12}) = r_{11}$,
$d_a(r_{22}) = r_{21}$,
$(d_a(r_{12}), d_a(r_{22})) \in A$,
$d_c(r_{12}) = E_1$,
$d_c(r_{22}) = r_{22}$,
4: $first(E_1) = \{a\}, first(r_{22}) = \{a, c\}$,
$A = \{(E_1, E_2), (r_{11}, r_{21}), (r_{12}, r_{22}), (E_1, r_{22})\}$,
$d_a(E_1) = r_{11}$,
$d_a(r_{22}) = r_{21}$,
$(d_a(E_1), d_a(r_{22})) \in A$.
`include` *returns TRUE, so* $L(E_1) \subseteq L(E_2)$.

Example 4.3. *Let* E_1 *and* E_2 *be the expressions from Example 4.2 and let us check* $L(E_2) \subseteq L(E_1)$ *instead: the first three steps are the same as those in Example 4.2, while the fourth step is as follows*

$d_c(r_{22}) = r_{22}$,
$d_c(r_{12}) = E_1$,
4: $first(r_{22}) = \{a, c\}, first(E_1) = \{a\}$,
$first(r_{22}) \not\subseteq first(E_1)$,
`include` *returns FALSE, so* $L(E_2) \not\subseteq L(E_1)$.

The bound is worst case optimal, one example is the expression *abc*. The set of derivatives for *abc* is $\{abc, bc, c, \varepsilon\}$, whose cardinality is 4 (*i.e.*, $\|abc\| + 1$). Therefore, the number of pairs of expressions to be checked in `include` is bounded to $\|E_1\| \cdot \|E_2\|$, which ensures the termination of the algorithm.

Although the derivative-based algorithm is rather simple, the worst case complexity is higher than the automata-based one. Recall from [13], any partial derivative of E is either ε or a subexpression of E or a concatenation of subexpressions where the number of subexpressions is no greater than the number of occurrences of concatenation and Kleene star appearing in E. Therefore, in the worst case, a partial derivative of E may have a size up to $|E|^2$. According to [12], derivatives of a one-unambiguous regular expression have the same form as the partial derivatives of the expression. Taking the possible time of comparisons in the algorithm `include` into account, the worst-case time complexity is $O((|E_1|^2 + |E_2|^2)\|E_1\|^2\|E_2\|^2)$. Also the worst-case space complexity is $O(|E_1|^2\|E_1\| + |E_2|^2\|E_2\|)$. However, in practice, computation rarely reaches the upper bounds. For example, in Example 4.2, there could be up to $4 \times 4 = 16$ pairs of derivatives, but there are only 5 pairs in A and 5 comparisons to be done.

Virtually the search strategy of the derivative-based algorithm is similar to the M_{E_1}-directed search presented in Section 3, without the need to compute all the derivatives in advance. Similar to the technique *up to congruence* [23] for the automata-based algorithm, Brzozowski's *aci-similarity* [14] can be used here to reduce the search, at the cost of the similarity computations. For example, assume the current pair is $(r_1 + r_2, r)$ and the visited set A contains $(r_2 + r_1, r + r)$. In that case, the matching of $(r_1 + r_2, r)$ with $(r_2 + r_1, r + r)$ in A succeeds, as $r_1 + r_2$ (resp. r) is similar to $r_2 + r_1$ (resp. $r + r$) under *aci-similarity*. Thus we do not need to check $(r_1 + r_2, r)$.

5. Applications

As mentioned in the Introduction, the inclusion algorithms for one-unambiguous regular expressions can be used in many tasks. In this section, we present the applications of our inclusion algorithms to one of the tasks, that is, XML typechecking.

Briefly speaking, one important issue of XML processing is typechecking, in which XML schema languages are regarded as types and subtype relations (*i.e.* inclusion relations) are checked at compile-time to ensure type correctness of programs. Many languages, such as XDuce, CDuce, etc., adopt unranked regular tree languages as types, however the complexity is EXPTIME for the inclusion of regular tree languages or, equivalently, tree automata. So the focus of prior research is on finding subclasses of regular tree languages for which subtyping is more efficient. For examples, Suzuki [6] proposes a polynomial-time algorithm for

solving a subproblem of the inclusion problem for DTDs defined by edit operations. Champavššre et al. [26] present a polynomial-time algorithm for testing inclusion between deterministic unranked tree automata and DTDs. Complexities of decision problems for several subclasses of XML schema languages involving subclasses of regular expressions are given in [1, 27].

In this section, we developed type checking algorithms for DTDs and for regular tree grammars with disjoint production rules (RRTGs). As shown in Proposition 2.4, every regular expression can be transformed into star normal form in linear time. So we assume that regular expressions are in star normal form (if not, the algorithm in [21] is applied first).

5.1. DTD Typechecking

XML schemas, such as DTD and XML Schema, are represented as *regular expression types* in typed XML processing languages, such as XDuce. Regular expressions types T (in XDuce) are defined as follows:

$$T ::= l[T] \mid\mid T, T \mid\mid T|T \mid\mid T* \mid\mid N \mid\mid ()$$

where the cases express label, concatenation, union, repetition, type name (type variable) and empty sequence respectively. We write $T?$ short for $T|()$ and $T+$ short for $T|T*$.

A type definition consists of type declarations of the form *type* $N = T$. Type name N is interpreted by type expression T which may contain any of the defined type names. Each type definition describes a set of XML documents. Type constructors of the form $l[...]$ denote tree nodes with the label l in XML documents (i.e., XML structures of the form $\langle l\rangle ...\langle /l\rangle$). Type $String$ equals to PCDATA in XML.

An example of type definitions using regular expression types is as follows (from XDuce source):

$$\begin{array}{lcl} type\ Addrbook & = & addrbook[Person*] \\ type\ Person & = & person[Name, Tel?, Email*] \\ type\ Name & = & name[String] \\ type\ Tel & = & tel[String] \\ type\ Email & = & email[String] \end{array}$$

For the type definition of DTDs, type names and labels are in *one-to-one correspondence*. The definition above satisfies the one-to-one correspondence condition, thus is a DTD. Given a label l, let N_l be the type name corresponding to the label l in the DTD and r_l be the regular expression in right-hand side of the type declaration of N_l.

Definition 5.1. *Given a regular expression type T, its depth $dp(T)$ is defined as follows:*

$$\begin{array}{rcl} dp(l[T]) & = & 1 + dp(T) \\ dp(T_1, T_2) & = & max(dp(T_1), dp(T_2)) \\ dp(T_1|T_2) & = & max(dp(T_1), dp(T_2)) \\ dp(T*) & = & dp(T) \\ dp(N) & = & 1 \\ dp(()) & = & 0 \end{array}$$

For example, considering the type definition above, $dp(Name) = 1$ and $dp(Person) = 2$.

Definition 5.2. *Given a regular expression type T, its top-level type $tlt(T)$ is defined as follows:*

$$\begin{array}{rcl} tlt(l[T]) & = & N_l \\ tlt(T_1, T_2) & = & tlt(T_1), tlt(T_2) \\ tlt(T_1|T_2) & = & tlt(T_1)|tlt(T_2) \\ tlt(T*) & = & tlt(T)* \\ tlt(N) & = & N \\ tlt(()) & = & () \end{array}$$

For example, $tlt(email[String]) = Email$ and $tlt(name[String], email[String]*) = Name, Email*$.

$$\frac{Dp(T_1) \leq 1 \quad \Pi(T_1, T_2)}{T_1 \leq T_2} \tag{1}$$

$$\frac{T_1 = l[T] \quad \Pi(N_l, T_2) \quad T \leq r_l}{T_1 \leq T_2} \tag{2}$$

$$\frac{Dp(T_1) > 1,\ T_1 = T_3, T_4 \quad \Pi((tlt(T_3), tlt(T_4)), T_2) \quad T_3 \leq tlt(T_3) \quad T_4 \leq tlt(T_4)}{T_1 \leq T_2} \tag{3}$$

$$\frac{Dp(T_1) > 1,\ T_1 = T_3 | T_4 \quad \Pi((tlt(T_3) | tlt(T_4)), T_2) \quad T_3 \leq tlt(T_3) \quad T_4 \leq tlt(T_4)}{T_1 \leq T_2} \tag{4}$$

$$\frac{Dp(T_1) > 1,\ T_1 = T* \quad \Pi(tlt(T_1), T_2) \quad T \leq tlt(T)}{T_1 \leq T_2} \tag{5}$$

Figure 5: Subtyping Rules for DTDs.

Subtyping is the most important issue in type checking, which refines the typing rules such that a value of any subtype of T can be safely used as a value of T. Let $T_1 \leq T_2$ denote type T_1 is a subtype of type T_2. For simplicity, we assume that T_2 is a regular expression over type names. This does not lose generality because T_2 can be replaced with a new type name N by adding a temporary type declaration *type* $N = T_2$ into the type definition, if T_2 contains some labels.

Due to the one-to-one correspondence, subtyping of DTDs can be reduced to the inclusion of regular expressions on top-level type names if the corresponding labels are the same, where our algorithms can be applied. For example, to check $person[Name, Email*] \leq Person$, we can reduce it to check the inclusion between $Name, Email*$ and the content of $Person$, that is $Name, Tel?, Email*$. The subtyping algorithm of DTDs is described by the rules in Figure 5, where $\Pi(r_1, r_2)$ is the inclusion function which determines if regular expression r_1 is included in r_2. Note that, $\Pi(r_1, r_2)$ can be any inclusion algorithm of one-unambiguous regular expressions, such as our algorithms presented above.

Consider the following two type expressions

$T_1 = person[Name, Email*]*$

$T_2 = Person*$

The checking procedure of $T_1 \leq T_2$ is illustrated in Figure 6. The procedure shows the subtyping check of $T_1 \leq T_2$ is finally reduced to several inclusion checks of one-unambiguous regular expressions. Besides, the algorithm can be improved by memoization such that the inclusion of each pair of regular expressions is checked only once. Indeed, this improvement has been implemented in our algorithm (see the discussion in Section 7.2).

5.2. Subtyping Algorithm for RRTG

In this section, we present a subtyping algorithm for regular tree grammars with disjoint production rules (RRTG for short) [28], where our inclusion algorithms for regular expressions can be applied. RRTG can describe the commonly used XML schema languages such as DTDs and XML Schemas. Here we briefly introduce some notions of RRTG and how our inclusion algorithms are used in the subtyping algorithm of RRTG. For technical details readers can refer to [28].

Given a regular expression r, let $\tau(r)$ denote the set of terminal strings derived by r, and $L(r)$ denote the regular language specified by r.

$$\dfrac{\dfrac{\textit{Return TRUE}}{\Pi(Person*, Person*)} \quad \dfrac{\dfrac{\textit{Return TRUE}}{\Pi(Person, Person)} \quad \dfrac{\dfrac{\textit{Return TRUE}}{\Pi((Name, Email*), (Name, Tel?, Email*))}(1)}{Name, Email* \leq r_{person}}}{person[Name, Email*] \leq Person}(2)}{person[Name, Email*]* \leq Person*}(5)$$

Figure 6: Procedure of $person[Name, Email*]* \leq Person*$.

(1)	$Store$	$\rightarrow$	$store\ (Regulars, Discounts)$
(2)	$Regulars$	$\rightarrow$	Dvd_1*
(3)	$Discounts$	$\rightarrow$	Dvd_2, Dvd_2*
(4)	Dvd_1	$\rightarrow$	$dvd\ (Title, Price)$
(5)	Dvd_2	$\rightarrow$	$dvd\ (Title, Price, Dis)$
(5)	$Title$	$\rightarrow$	$title$ PCDATA
(7)	$Price$	$\rightarrow$	$price$ PCDATA
(8)	Dis	$\rightarrow$	dis PCDATA

Figure 7: Grammar for DTD Store.

Definition 5.3. *A regular tree grammar with disjoint production rules (RRTG) G is a regular tree grammar in which any two production rules $N_i \rightarrow l_i r_i$ and $N_j \rightarrow l_j r_j$ satisfy either $l_i \neq l_j$, or $\tau(r_i) \cap \tau(r_j) = \emptyset$.*

Let us consider the regular tree grammar G_1 (not in normal form) in Figure 7, which describes a simple database of DVD store, where the lowercase words are labels and the words with the first letter capitalized are non-terminals. We have $L(Dvd_1) = \{Dvd_1\}$ and $\tau(Dvd_1) = \{dvd\ (title, price)\}$ (PCDATA are ignored). Both Rules (2) and (3) of G_1 contain no label. According to Definition 5.3, the intersection $\tau(Dvd_1*) \cap \tau(Dvd_2, Dvd_2*)$ is not required to be an empty set. Rules (4) and (5) have the same label dvd at their right-hand sides, but $\tau(Title, Price) \cap \tau(Title, Price, Dis) = \emptyset$. So G_1 is a RRTG.

In most statically typed XML processing languages, type systems are implemented based on regular expression types, and subtyping is reduced to the inclusion between tree automata, which is not efficient enough. While for RRTG, since any two production rules are disjoint, this enables us to find an *unique* type T for each bottom level expression $l[r]$ such that $l[r] \leq T$, which can be reduced to the inclusion of the content expressions, and then check the subtyping between the types obtained by replacing bottom level expression $l[r]$ by its super-type T, yielding an efficient subtyping algorithm.

Let us consider RRTG G_1 and suppose to check $T_1 \leq T_2$, where

$T_1 = store(Dvd_1, dvd(Title, Price), dvd(Title, Price, Dis))$

$T_2 = Store$

The checking procedure of $T_1 \leq T_2$ is shown in Figure 8, where T, P and D are short for $Title, Price$ and Dis, respectively, and $\Pi(r_1, r_2)$ denotes the inclusion check between r_1 and r_2. Note that, since G_1 is a RRTG, we can find the *unique* super types Dvd_1 and Dvd_2 for the bottom label expressions $dvd(T, P)$ and $dvd(T, P, D)$ respectively, which are reduced to the inclusion check of regular expressions on type names. Then we will check the subtyping with respect to the replaced types $store(Dvd_1, Dvd_1, Dvd_2)$ instead, which can be reduced to the inclusion check as well. Likewise, memoization has been used to improve the algorithm.

6. Experiments for Inclusion Algorithms

In this section, we conduct some experiments to evaluate our algorithms. We also compare our algorithms against Hovland's algorithm [16], which is guaranteed to decide the inclusion problem if the second expression

$$\dfrac{\dfrac{\dfrac{Return\ TRUE}{\Pi((T,P),\ (T,P))}}{dvd(T,P) \leq Dvd_1} \quad \dfrac{\dfrac{Return\ TRUE}{\Pi((T,P,D),\ (T,P,D))}}{dvd(T,P,D) \leq Dvd_2}}{Dvd_1, dvd(T,P), dvd(T,P,D) \leq Dvd_1, Dvd_1, Dvd_2} \quad \dfrac{\dfrac{Return\ TRUE}{\Pi((Dvd_1, Dvd_1, Dvd_2),\ (Dvd_1*, Dvd_1, Dvd_2*))}}{store(Dvd_1, Dvd_1, Dvd_2) \leq Store}$$
$$store(Dvd_1, dvd(T,P), dvd(T,P,D)) \leq Store$$

Figure 8: Procedure of $store(Dvd_1, dvd(T,P), dvd(T,P,D)) \leq Store$.

E_2 is one-unambiguous, and might either decide the problem correctly or report that the one-ambiguity is a problem if E_2 is one-ambiguous.

We have implemented our inclusion algorithms and Hovland's algorithm in C++. We use binary trees to encode the postfix expressions of one-unambiguous regular expressions and their derivatives, and implement automata as conventional five-field structs, with two-dimensional arrays of integers to represent the transition functions. Prior to inclusion checking, we check the nullable and compute the first set for each subexpression via a bottom-up and incremental traversal on the binary tree, which costs linear time. At last, we implement the algorithms as shown in Section 3 and Section 4.

Test Cases. To evaluate the algorithms, we perform the experiments on randomly generated one-unambiguous regular expression pairs. Similar to D. Colazzo *et al.*'s work [29], we generate our expression pairs using two modes. The first mode is to generate both two expressions in a pair (E_1, E_2) randomly: we randomly generate 100 expressions as E_2 whose length ranges from 1 to 150, and then for each E_2 we randomly generate 10 expressions as E_1 based on the alphabet of E_2, yielding 150000 pairs of expressions. However, as discussed in [29], the problem of generating random regular expressions for inclusion is not trivial: when a random pair of regular expressions is generated, even if they share the same alphabet, the probability that one of the two expressions is included in the other is extremely low (about 0.0279% for regular expressions). Likewise, the problem for the case of one-unambiguous regular expressions could be extremely low as well. Indeed, only 280 pairs among 150000 satisfy the inclusion relation, most of which have length smaller than 5. So these pairs could test the algorithms in a negative case.

Hence, the second mode is to generate random expression pairs such that they satisfy the inclusion relation. Motivated by D. Colazzo *et al.*'s work [29], our idea is to generate a random one-unambiguous regular expression E_2 first, and then apply a set of probabilistic rewriting rules on E_2 to obtain another expression E_1 such that E_1 is one-unambiguous and is included in E_2, yielding a pair (E_1, E_2). The rewriting rules we used here are given in the following:

- for pattern ϵ or a, return the pattern itself;
- for pattern $E_1|E_2$, let E_i^r be the expression obtained by applying the rewriting rules on E_i, then return $E_1^r|E_2^r$, $E_1^r|E_2$, $E_1|E_2^r$, $E_1|E_2$, $E_2^r|E_1^r$, $E_2^r|E_1$, $E_2|E_1^r$ or $E_2|E_1$ randomly;
- for pattern $E_1.E_2$, let E_i^r be the expression obtained by applying the rewriting rules on E_i, then return $E_1^r.E_2^r$, $E_1^r.E_2$, $E_1.E_2^r$ or $E_1.E_2$ randomly; If $\epsilon \in L(E_1)$, then E_2^r or E_2 can be returned as well;
- for pattern $(E_1)^*$, let E_1^r be the expression obtained by applying the rewriting rules on E_1, then return $(E_1^r)^*$, $(E_1)^*$, E_1^r or E_1 randomly.

In detail, we randomly generate 100 expressions as E_2 whose length ranges from 1 to 150, using the random generator in [30]; and for each expression E_2 we generate 10 expressions as E_1 by applying the rewrite rules above, yielding 150000 pairs of expressions.

Fortunately, besides satisfying the inclusion relation, the rewritten expression obtained from the application of the rewriting rules above on a one-unambiguous expression is also one-unambiguous, which is formalized in Lemma 6.1. Therefore, we do not need the one-unambiguous checking on the rewritten expressions.

Lemma 6.1. *Let E be a one-unambiguous regular expression and E^r be the rewritten expression obtained from the application of the rewriting rules above on E. Then $L(E^r) \subseteq L(E)$ and E^r is also one-unambiguous.*

Proof. Without loss of generality, we assume that E is not empty. We prove a stronger statement: the rewritten expression E^r is also one-unambiguous and satisfies that $L(E^r) \subseteq L(E)$, $first(E^r) \subseteq first(E)$ and $followlast(E^r) \subseteq followlast(E)$, which deduces this lemma immediately. We prove the statement by induction on the structure of E and E^r.

- E is ϵ or a. Then $E^r = E$, thus the result follows.
- E is $E_1|E_2$. Here we only show the case of $E^r = E_1^r|E_2^r$, where E_i^r is the expression obtained by applying the rewriting rules on E_i and $i = 1, 2$; the other cases can be proved similarly. According to Proposition 4.1, we know that E_1, E_2 are one-unambiguous and $first(E_1) \cap first(E_2) = \emptyset$. By induction on E_i, we have E_i^r is also one-unambiguous and satisfies that $L(E_i^r) \subseteq L(E_i)$, $first(E_i^r) \subseteq first(E_i)$ and $followlast(E_i^r) \subseteq followlast(E_i)$. As $first(E_1) \cap first(E_2) = \emptyset$, we can further get $first(E_1^r) \cap first(E_2^r) = \emptyset$. According to Proposition 4.1 again, E^r is one-unambiguous. Moreover, $L(E^r) = L(E_1^r) \cup L(E_2^r) \subseteq L(E_1) \cup L(E_2) = L(E)$. Similarly, we have $first(E^r) \subseteq first(E)$ and $followlast(E^r) \subseteq followlast(E)$. Thus the result follows.
- The other cases are similar.

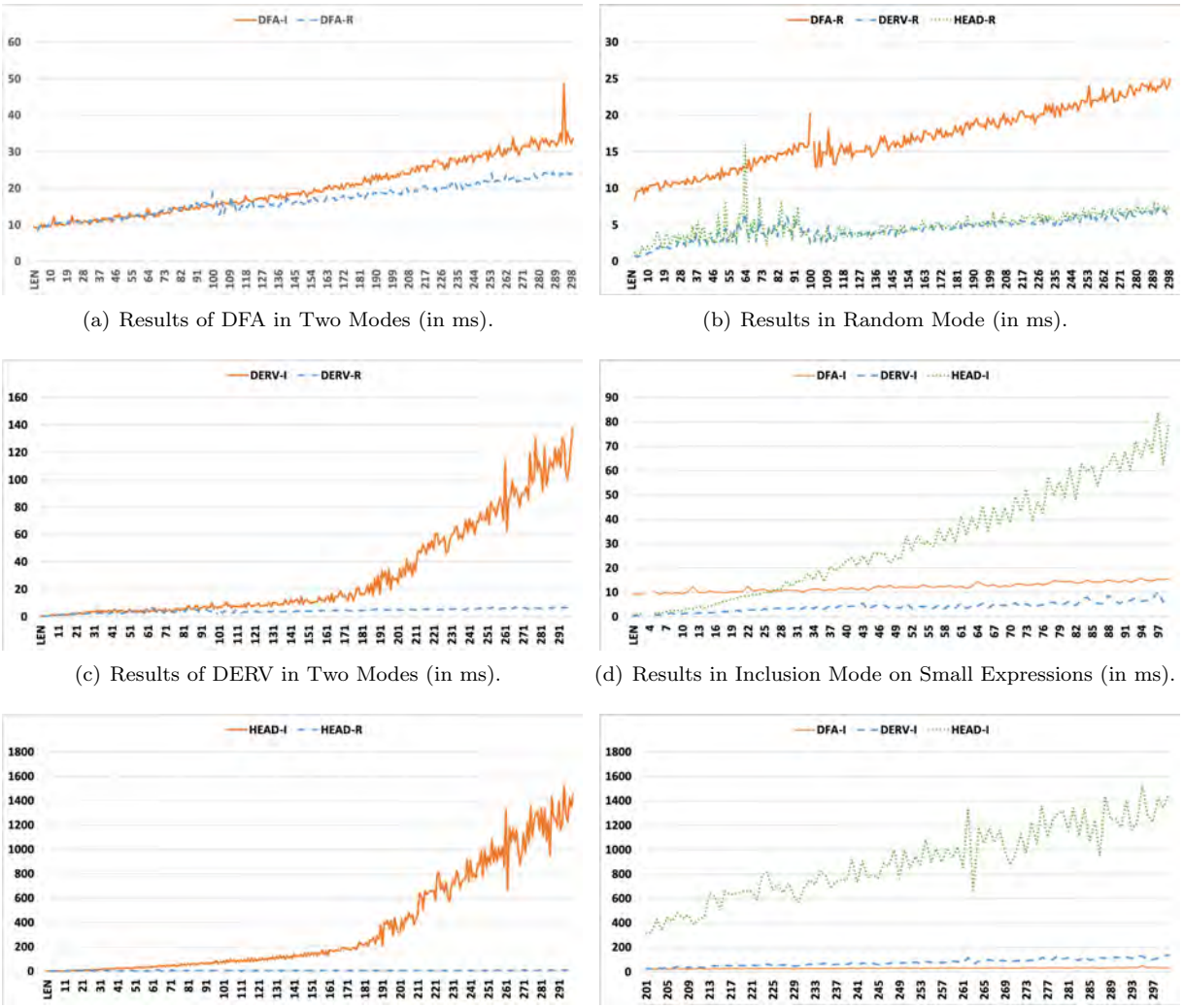
□

We used the generator in [30] to generate one-unambiguous regular expressions. This generator is based on the context-free grammars for one-unambiguous regular expressions and thus can generate one-unambiguous regular expressions directly. In detail, given an alphabet Σ and a length l, this generator starts from any nonterminal uniformly, as any nonterminal can be the start symbol for the grammar charactering the one-unambiguous regular expressions on Σ. Then during the generation, this generator selects a production for each nonterminal uniformly from the ones that satisfy some length-control conditions, which could lead the generation to terminating with an expression such that the length is as close to l as possible. There is an alternation to generate one-unambiguous regular expressions: first use a generator for regular expressions (or automata), then decide whether the generated expressions are deterministic. However, as demonstrated in [30], this solution is so inefficient (*e.g.*, to generate 100 one-unambiguous regular expressions with length 50 requires the generator to generate about 8618 regular expressions with a ratio 1.2%) that it is not feasible for generating one-unambiguous regular expressions. Therefore, only the generator in [30] is used here.

Experiments. We run each inclusion algorithm on the pairs generated by each mode, and account the runtime in milliseconds. To avoid the perturbations introduced by system activity, we ran each experiment 100 times, discarded the best and the worst performance, and computed the average of the remaining times. The experiments are run on a workstation with Intel Xeon Silver 4214 Processor, 64GB RAM, Ubuntu 18.04.

The experimental results for each algorithm under each mode are given in Figure 9, where the horizontal axis denote the sum of the lengths of expression pairs, and DFA, DERV, HEAD respectively denote our automata-based algorithm, our derivative-based algorithm, and Hovland's Header-form-based algorithm, with the suffix "-R" and "-I" respectively denoting the random mode (*i.e.*, the first mode) and the inclusion mode (*i.e.*, the second mode).

First, it can be seen from the results (Figures 9(a), 9(c) and 9(e)) that the performances in the random mode are much better than the ones in the inclusion mode, which are consistent with the discussion above. In detail, the derivative-based algorithm and Hovland's algorithm perform much more efficiently on the pairs generated in the random mode than on the ones generated in the inclusion mode, while the performances of the automata-based algorithm are almost the same for expressions with size smaller than 100 in both modes. The main reason is that the automata-based algorithm will construct two corresponding Glushkov DFAs for both expressions, which costs most of the runtime, regardless of whether the inclusion of the two expressions was satisfied or not. However, thanks to the improvements, the complement automata and intersection automata need not to be constructed fully, so the automata-based algorithm in the random mode performs



(a) Results of DFA in Two Modes (in ms).

(b) Results in Random Mode (in ms).

(c) Results of DERV in Two Modes (in ms).

(d) Results in Inclusion Mode on Small Expressions (in ms).

(e) Results of HEAD in Two Modes (in ms).

(f) Results in Inclusion Mode on Large Expressions (in ms).

Figure 9: Results for Different Algorithms.

a little more efficiently in the inclusion mode for expressions with size larger than 100. Moreover, in the inclusion mode, the runtime for all the algorithm increases, as the size increases. In particular, the runtime of the automata-based algorithm increases almost linearly, while both the runtimes of the derivative-based and Hovland's algorithm increase quadratically. In addition, from the results we can also see that the runtime of Hovland's algorithm grows much faster than those of both our algorithms.

Figure 9(b) shows the comparison of three algorithms in the random mode, from which we can see that the derivative-based algorithm performs the best (due to that looking for a reason to fail is easy to spot), while the automata-based algorithm does the worst (due to the constructing of Glushkov DFAs).

As Hovland's algorithm grows too fast, the comparison of three algorithms in the inclusion mode is shown in two subfigures: Figure 9(d) shows the results on sizes ranging from 1 to 100 (*i.e.*, small expressions) and Figure 9(f) gives the results on sizes ranging from 200 to 300 (*i.e.*, large expressions). The results show the derivative-based algorithm is more efficient on small expressions (*e.g.*, smaller than 100), while the automata-based algorithm performs better on large expressions (*e.g.*, larger than 200). This is because the derivative-based algorithm performs, in a sense, a breadth-first exploration of the two compared expressions,

and smaller expressions tend to have a smaller amount of such exploration. Moreover, compared with Hovland's algorithm, both of our algorithms are more efficient. Although Hovland's algorithm has a better performance on very small expressions (*e.g.*, smaller than 10), it becomes time-consuming very quickly as the size increases, while both of our algorithms can check the expression pairs with size no larger than 300 in 0.2 seconds. The reason is that Hovland's algorithm can deal with not only the one-unambiguous expressions, but also some one-ambiguous cases, thus it needs to consider more cases than ours. Indeed, our derivative-based algorithm can be viewed as a simple inference system of Hovland's consisting of a variant of Rule (Axm) (to check the inclusion of the First sets) and Rule (Letter). It is also easy to extend our algorithms to deal with general regular expressions, but the complexity will be increased as well.

To sum up, both our algorithms are more efficient than Hovland's algorithm on the whole. We suggest to use the automata-based algorithm in practice, since it performs well for both modes, that is, the runtime is acceptable no matter what is the relation between two expressions.

7. Experiments for XML Typechecking

In this section, we present some experiments to see the effectivity of our inclusion algorithms applied on XML typechecking and to examine the practical efficiency of our typechecking algorithms for DTD and for RRTG. For that, we conducted some preliminary experiments to compare our typechecking algorithms with XDuce, an XML processing language which supports regular tree languages as schemas. Comparisons with CDuce are also included.

As both XDuce and CDuce are implemented in OCaml, we also implement our typechecking algorithms for DTD and for RRTG in OCaml, where the automata-based inclusion algorithm is used and reimplemented in OCaml for a fair comparison. The experiments in this section are run under the environment: Intel Pentium 4 CPU 3.0GHz, 512MB RAM, Ubuntu.

7.1. Comparison with XDuce

In this section, we conducted some preliminary experiments to compare our typechecking algorithms with XDuce. To do that, we apply our typechecking algorithms and XDuce on the same test suite, which covers the actually used scenarios of XML processing, including fragment extraction, type conversion and so on. The test suite consists of the following examples, all of which, except the last one, are collected from XDuce [32]:

addrbook: defines a DTD of an address book file and extracts a phone book from a XML file.

bookmarks: converts a Netscape bookmark file (a subset of type HTML) into a file of full HTML type with an external DTD (xhtml1-transitional.dtd).

html2latex: converts a file (of type HTML) into a LATEX file (of type string) with an external DTD (xhtml1-transitional.dtd).

ns2xbel: converts a Netscape bookmark file into XBEL format with an external DTD (xbel-1.0.dtd).

rng2xduce: converts a file of type Relax NG into XDuce format.

polysample: is a simple polymorphic program.

dvdstore: extracts the dvd data from a input file with respect to the type definition of DVD store given in Section 5.2, which is a RRTG but not a DTD. This is a test case written by ourselves for testing our algorithm for RRTG. Detailed XDuce code for dvdstore is contained in Appendix A.

The size of these examples in the test suite are shown in Table 2, which are measured by line of code, where $size$ and $size_e$ respectively represents the size of the example and the size of the external DTD called by the example (the dash '-' means no external DTDs). Some examples are small in size (*e.g.*, 35), while some are not (*e.g.*, 451). Some examples call larger external DTDs (*e.g.*, $html2latex$), which will be converted to the corresponding type definitions of the program during processing, while some are not (*e.g.*, *bookmarks*). The content models in the examples are all one-unambiguous, which indicates the inclusion algorithms for one-unambiguous are helpful in practice.

We run each example in the test suite with our typechecking algorithm for DTD, and our algorithm for RRTG and XDuce respectively, and then collect the typechecking times in seconds. The experimental

Table 2: Sizes for examples.

	addrbook	*bookmarks*	*html2latex*	*ns2xbel*	*rng2xduce*	*polysample*	*dvdstore*
size	67	216	300	77	451	83	35
$size_e$	-	1198	1198	94	-	-	-

Table 3: Experimental results for XML typechecking (in s).

Example	*XDuce*	*DTD*	*RRTG*
addrbook	0.003188	0.001367	0.001532
bookmarks	0.581173	0.156781	0.163633
html2latex	1.204572	0.280307	0.297592
ns2xbel	0.002295	0.001122	0.001202
rng2xduce	0.594637	0.219868	0.223946
polysample	0.002016	0.001059	0.001157
dvdstore	0.532574	-	0.302241

results are shown in Table 3, where *XDuce* denotes the typechecking time needed by XDuce, *DTD* denotes the typechecking times needed by our typechecking algorithm for DTD (the dash '-' means the example is not performed), and *RRTG* denotes the typechecking times needed by our typechecking algorithm for RRTG.

The experimental results show that both our typechecking algorithms for DTD and RRTG are more efficient than XDuce. This is mainly because our typechecking algorithms make full use of the one-to-one correspondence for DTD and the disjoint condition for RRTG. Moreover, from Table 3, we can also see that the typechecking algorithm for DTD is more efficient than the one for RRTG. The reason is that the typechecking algorithm for RRTG needs to find an unique type for each bottom level expression, while the one for DTD needs not and thus is simpler.

7.2. Comparison with CDuce

In this section, we conducted some preliminary experiments to compare our typechecking algorithms with CDuce [18].

For the comparison, we collect the example *addrbook*, including XDuce and CDuce codes, from the benchmarks of CDuce (*i.e.*, CDuce vs. XDuce, Addrbook) [33]. Detailed XDuce code is given in Appendix B. Then we run the example *addrbook* for many times with XDuce, CDuce and our typechecking algorithm for RRTG. In detail, we let the major function *mekTelbook* of *addrbook* run and repeat n times during the experiments. This is twofold: one is to ensure the accuracy of time, the other is we found XDuce optimizes the typechecking algorithm by memoizing the intermediate results to avoid repeated checking. Finally, the typechecking times are collected in seconds.

The experimental results are shown in Figure 10, where the horizontal axis denotes the number of times n that *mekTelbook* is repeated and the vertical axis denotes the typechecking times that needed by XDuce, CDuce and our RRTG algorithm, respectively.

From the experimental results, we can see that CDuce is more efficient than XDuce and RRTG for a one-time typechecking (*i.e.*, when n is small). This is mainly because that CDuce makes several improvements on XDuce, including increased runtime efficiency and improved typechecking efficiency. In particular, CDuce adopts non-uniform automata [34], a merger between top-down deterministic automata and bottom-up deterministic automata, to improve typechecking algorihm. While both XDuce and ours work in a top-down way. Moreover, the experimental results also indicate that both XDuce and RRTG become more efficient than CDuce as n increases. This is due to the memoization of the intermediate results used in XDuce and RRTG. In our opinion, the memoization can highly improve efficiency in practice, as it is helpful to avoid the frequent and repeated subtype checking.

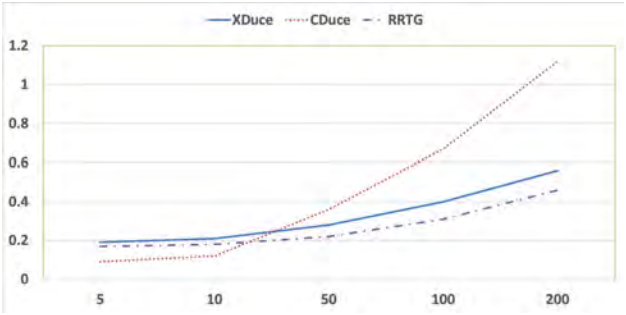


Figure 10: Comparison of XDuce, CDuce and RRTG.

8. Related Work

One-unambiguous regular expressions. One-unambiguous regular expressions have been a topic of research since they were formally defined by Brüggemann-Klein and Wood [9], also under the name of deterministic regular expressions or weakly deterministic regular expressions. Most of existing works are on determinism (*i.e.*, whether a regular expression possibly with some extensions is one-unambiguous) [9, 35, 36, 37, 38, 39, 40], generalization (*i.e.*, to extend the definition of one-unambiguous regular expressions) [41, 42, 43, 44, 45], definability (*i.e.*, where a regular language or expression can be defined by a one-unambiguous regular expressions) [46, 47, 48], or descriptional complexity (*i.e.*, the size complexity of the smallest representation for a one-unambiguous regular expression) [49]. However, there are few works on the algorithms for the inclusion problem of one-unambiguous regular expressions. In the paper, we focus on the inclusion and presents two algorithms, one is based on automata and the other one is based on derivatives. Hovland [16] gives another algorithm, which is similar to our derivative-based algorithm. Moreover, we also conduct some experiments to compare our algorithms with Hovland's, which shows that both of our algorithms are efficient than Hovland's.

Inclusion of XML schemas. A schema S_2 includes a schema S_1 if for any document d that is valid against S_1, d is valid against S_2. Martens et al. [1] study the relations between complexity for decision problems for DTD and XML Schema (single-type SDTD) and complexity for decision problems for the corresponding regular expressions, and show that for inclusion the complexity bounds for the regular expressions carry over to DTD and XML Schema, so it suffices to restrict attention to the complexity of regular expressions to derive complexity bounds for XML schema languages.

Martens et al. [1] give complexity of decision problems for several subclasses of regular expressions called CHAREs occurring in practice in XML schemas. Their results show that inclusion is already co-NP-complete for very innocent expressions such as expressions with factors of the form a or a^*. If the number of occurrences of the same symbol in expressions is bounded by some k ($\text{RE}^{\leq k}$), inclusion is in PTIME. Several authors give complexity for regular expressions with interleaving and/or numerical occurrence indicators [3, 4, 5]. These expressions are allowed in schema languages like XML Schema and Relax NG. Inclusion for these expressions is in EXPSPACE.

One-unambiguous regular expressions are used in DTD and XML Schema. Since one-unambiguous regular expressions can be transformed to DFAs in linear time, inclusion for this kind of expressions is in PTIME. However, there are few algorithms for the inclusion. Chen and Chen [19] give the first two algorithms, and Hovland [16] gives another algorithm.

For DTD inclusion, Suzuki [6] proposes a polynomial-time algorithm for solving a subproblem of the inclusion problem for regular expressions defined by edit operations. However, the algorithm does not cover the inclusion problem for one-unambiguous regular expressions.

Ghelli et al. [50] propose a restricted class of regular expressions with interleaving and counting, give the notion of conflict freedom, and a cubic algorithm for the symmetric inclusion of conflict-free types. This algorithm has been further refined in Colazzo et al. [51], where a quadratic-time algorithm has been defined. Colazzo et al. [29, 52] describe quadratic-time algorithm for asymmetric inclusion. The algorithm

requires the supertype to be conflict free, while the subtype can be any type. Colazzo et al. [53, 54] present an algorithm that is linear time in the common situations and resorts to the quadratic, constraints-based approach only for some special cases. The class also does not cover one-unambiguous regular expressions.

Amavi et al. [55] proposes a method to verify whether a possibly-recursive XML type is "approximatively" included in another XML type, where the approximation consists in weakening the father-children relationships. While our typechecking algorithms require "full" inclusion.

XML processing languages. Typechecking algorithms are proposed in XML processing languages like XDuce [17], CDuce [18], XHaskell [56], and so on. These algorithms solve the inclusion problem for regular tree languages. Inclusion is EXPTIME-complete in general, and there is no complexity results for the algorithms in the case of DTD or XML Schema. When considering polymorphic types, both polymorphic version of XDuce [57] and Vouillon's work [58] have to give up something. While XHaskell [56] mixes Haskell type classes with XDuce's regular expression types, it has two main drawbacks. First, every polymorphic variable must be annotated. Second, even without inference of explicit annotations (which they do not address), their system requires several restrictions for decidability. The work given in [59, 60, 61] provides the theoretical basis and the algorithmic tools needed to design and implement polymorphic functional languages for semi-structured data. In particular, the results pave the way to the polymorphic extension of CDuce. As a follow up of [59], Gesbert et al. [62] encode the subtyping relation of (polymorphic) XML types [59] into a tree logic, and use a satisfiability solver to efficiently decide it. How to extend our inclusion algorithms to polymorphic versions is a future work.

9. Concluding Remarks

We proposed two algorithms for checking inclusion of one-unambiguous regular expressions. One algorithm is based on automata and the other one is based on derivatives. Then we presented the application of the algorithms to XML typechecking, namely DTD typechecking and RRTG typechecking. Finally we gave experimental studies of the algorithms. We first compared the time efficiency of our algorithms together with Hovland's algorithm. The results show under the inclusion mode (wherein pairs satisfying the inclusion relation are generated) the derivative-based algorithm is more efficient than the automata-based one for small expressions (*i.e.*, the total length of two expressions is smaller than 100), and for large expressions the latter is more efficient. They also show that both of our algorithms are more efficient than Hovland's algorithm for one-unambiguous regular expressions. We also conducted some experiments by applying the algorithms to typechecking of XML, which show that our algorithms are helpful and quite efficient for XML typechecking.

Further experiments with more examples are still useful. Extending our algorithms to support interleaving and counting, and considering polymorphism are all important future works. The algorithms can also be used to other tasks of XML processing that require checking inclusion of schemas, which remain as future works.

Acknowledgment

We thank Lixiao Zheng for her careful reading and helpful comments on the writing of an earlier version of the paper. We also thank the editor and the anonymous reviewers for their constructive comments. This work was supported by National Natural Science Foundation of China under Grants Nos. 61872339, 61472405, 61972260 and 61502308, and Guangdong Basic and Applied Basic Research Foundation under Grant No. 2019A1515011577.

References

[1] Martens W, Neven F, Schwentick T. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM Journal on Computing*, 2009, 39(4):1486 – 1530.

[2] Stockmeyer L J, Meyer A R. Word problems requiring exponential time (preliminary report). In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, 1973, pp. 1 – 9.

[3] Mayer A J, Stockmeyer L J. Word problems - this time with interleaving. *Information and Computation*, 1994, 115(2):293 – 311.
[4] Kilpeläinen P, Tuhkanen R. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation*, 2007, 205:890–916.
[5] Gelade W, Martens W, Neven F. Optimizing schema languages for XML: numerical constraints and interleaving. *SIAM Journal on Computing*. 38(5): 2021-2043 (2009)
[6] Suzuki N. An edit operation-based approach to the inclusion problem for DTDs. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07)*, 2007, pp. 482 – 488.
[7] Bray T et al. *XML 1.1 (Second Edition)*. W3C Recommendation, 2006.
[8] Sperberg-McQueen C M, Thompson H. *XML Schema*, 2005. `http://www.w3.org/XML/Schema`.
[9] Bruggemann-Klein A, Wood D. One-unambiguous regular languages. *Information and Computation*, 1998, 142(2):182–206.
[10] Glushkov V M. The abstract theory of automata. *Russian Math, Surveys*, 1961, 16:1 – 53.
[11] McNaughton R, Yamada H. Regular expressions and state graphs for automata. *IEEE Trans. on Electronic Computers*, 1960, 9(1):39 – 47.
[12] Chen H. Finite automata of expressions in the case of star normal form and one-unambiguity. Technical report, ISCAS-LCS-10-11, June 2010.
[13] Antimirov V. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 1996, 155:291 – 319.
[14] Brzozowski J A. Derivatives of regular expressions. *J. ACM*, 1964, 11(4):481–494.
[15] Steven Grijzenhout, Maarten Marx. The quality of the XML Web. Web Semantics: Science, Services and Agents on the World Wide Web, 2013, 19:59-68.
[16] Hovland D. The inclusion problem for regular expressions. *Journal of Computer and System Sciences*, 2012, 78(6):1795–1813.
[17] Hosoya H, Pierce B. XDuce: a statically typed XML processing language. *ACM Transactions on Internet Technology*, 2003, 3(2):117–148.
[18] Benzaken V, Castagna G, Frisch A. CDuce: An XML centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2003, pp. 51 – 63.
[19] Chen H, Chen L. Inclusion test algorithms for one-unambiguous regular expressions. In *Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC'08)*, LNCS 51602008, 2008, pp. 96–110.
[20] Sheng Y. Regular languages. In Grzegorz R, Arto S, editors, *Handbook of Formal Languages: Volume 1 Word, Language, Grammar*, 1997, pp. 41–110.
[21] Bruggemann-Klein A. Regular expressions into finite automata. *Theoretical Computer Science*, 1993, 120:197 – 213.
[22] Cormen H, Leiserson C E, Rivest R L, Stein C. *Introduction to Algorithms*. The MIT Press, 2001.
[23] Filippo Bonchi, Damien Pous. Hacking Nondeterminism with Induction and Coinduction *Commun. ACM*, 2015, 58(2):87-95.
[24] Brzozowski J A. Roots of Star Events. *J. ACM*, 1967, 14(3):466-477.
[25] Antimirov V. Rewriting regular inequalities. In *Proc. of the 10th International Symposium on Fundamentals of Computation Theory*, volume 965, 1995, pp. 116–125.
[26] Champavère J, Gilleron R, Lemay A, Niehren J. Efficient inclusion checking for deterministic tree automata and DTDs. In *Proceedings of Second International Conference on Language and Automata Theory and Applications*, 2008, pp. 184–195.
[27] Martens W, Neven F, Schwentick T, Bex G J. Expressiveness and complexity of XML Schema. *ACM Trans. Database Syst.*, 2006, 31(3):770–813.
[28] Chen L, Chen H. Subtyping algorithm of regular tree grammars with disjoint production rules. In *Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC'10)*, LNCS 6255, 2010, pp. 45–59.
[29] Colazzo D, Ghelli G, Pardini L, Sartiani C. Efficient asymmetric inclusion of regular expressions with interleaving and counting for XML type-checking. *Theoretical Computer Science*, 2013, 492(24):88–116.
[30] H. Chen, P. Lu, Z. Xu. Towards an Effective Syntax for Deterministic Regular Expressions. Technical Report. Institute of Software Chinese Academy of Sciences, 2018.
[31] *XDuce Webpage*. `http://xduce.sourceforge.net/`.
[32] *CDuce Benchmarks*. `http://www.cduce.org/bench.html#xduce`.
[33] Frisch A. Regular Tree Language Recognition with Static Information. *Ifip Advances in Information and Communication Technology*, 2004, 155:661-674.
[34] Kilpeläinen P. Checking determinism of XML Schema content models in optimal time. *Information Systems*, 2011, 36(3):596 – 617.
[35] Groz B, Maneth S. Efficient testing and matching of deterministic regular expressions. *J. Comput. Syst. Sci.* 89: 372-399 (2017)
[36] Lu P, Bremer J, Chen H. Deciding determinism of regular languages. *Theory of Computing Systems*, 2015, 57(1):97–139.
[37] Chen H, Lu P. Checking determinism of regular expressions with counting. *Information and Computation*, 2015, 241:302 – 320.
[38] Lu P, Peng F, Chen H, Zheng L. Deciding determinism of unary languages. *Information and Computation*, 2015, 245:181 – 196.
[39] Peng F, Chen H, Mou X. Deterministic regular expressions with interleaving. In *Proceedings of the 12th International Colloquium on Theoretical Aspects of Computing (ICTAC 2015)*, 2015, pp. 203–220.
[40] Dora G, Rosa M, Derick W. Block-deterministic regular languages. In *Proceedings of the 7th Italian Conference on*

Theoretical Computer Science (ICTCS 2001), 2001, pp. 184–196.

[41] Han Y, Wood D. Generalizations of 1-deterministic regular languages. *Information and Computation*, 2008, 206(9):1117 – 1125.

[42] Caron P, Han Y, Mignot L. Generalized one-unambiguity. In *Proceedings of the 15th International Conference on Developments in Language Theory (DLT 2011)*, 2011, pp. 129–140.

[43] Caron P, Mignot L, Miklarz C. On the hierarchy of block deterministic languages. In *Proceedings of the 20th International Conference on Implementation and Application of Automata (CIAA 2015)*, 2015, pp. 63–75.

[44] Caron P, Mignot L, Miklarz C. On the hierarchy of generalizations of one-unambiguous regular languages. *Theoretical Computer Science*, 2016, (679):95–106

[45] Huang E, Wang D, Liou D. Development of a deterministic XML schema by resolving structure ambiguity of HL7 messages. *Computer Methods and Programs in Biomedicine*, 2005, 80(1):1 – 15.

[46] Wojciech C, Claire D, Katja L, Wim M. Deciding definability by deterministic regular expressions. *Journal of Computer and System Sciences*, 88 (2017): 75–89.

[47] Markus L, Matthias N. Definability by weakly deterministic regular expressions with counters is decidable. In *Mathematical Foundations of Computer Science 2015*, 2015, pp. 369–381.

[48] Katja L, Wim M, Matthias N. Closure properties and descriptional complexity of deterministic regular expressions. *Theoretical Computer Science*, 2016, 627:54 – 70.

[49] Ghelli G, Colazzo D, Sartiani C. Efficient inclusion for a class of XML types with interleaving and counting. In *Proceedings of the 11th International Symposium on Database Programming Languages (DBPL 2007)*, LNCS 4797, 2007, pp. 231 – 245.

[50] Colazzo D, Ghelli G, Sartiani C. Efficient inclusion for a class of XML types with interleaving and counting. *Inf. Syst*, 2009, 34(7):643–656.

[51] Colazzo D, Ghelli G, Sartiani C. Efficient asymmetric inclusion between regular expression types. In *Proceedings of the 12th International Conference on Database Theory (ICDT'09)*, 2009, pp. 174–182.

[52] Colazzo D, Ghelli G, Pardini L, Sartiani C. Almost-linear inclusion for XML regular expression types. *ACM Transactions on Database Systems (TODS)*, 2013, 38(3):15:1–15:45.

[53] 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 (CIKM09)*, 2009, pp. 137–146. ,.

[54] Joshua A, Jacques C, Mirian H, Pierre R. Weak inclusion for XML types. In *Proceedings of the 16th International Conference on Implementation and Application of Automata*, 2011, pp. 30–41.

[55] Sulzmann M, Lu K. XHaskell - adding regular expression types to Haskell. *Implementation and Application of Functional Languages*, 2007, pp. 75–92.

[56] Hosoya H, Frisch A, Castagna G. Parametric polymorphism for XML. *ACM TOPLAS*, 2009, 32(1):1–56.

[57] Vouillon J. Polymorphic regular tree types and patterns. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, 2006, pp. 103 – 114.

[58] Castagna G, Xu Z. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*, 2011, pp. 94–106.

[59] Castagna G, Nguyn K, Xu Z, Im H, Lenglet S, Padovani L. Polymorphic functions with set-theoretic types. part 1: Syntax, semantics, and evaluation. In *Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*, January 2014, pp. 5–17.

[60] Castagna G, Nguyn K, Xu Z, Abate P. Polymorphic functions with set-theoretic types. part 2: Local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, 2015, pp. 289–302.

[61] Gesbert N, Genevès P, Layaïda N. Parametric polymorphism and semantic subtyping: the logical connection. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*, 2011, pp. 107–116.

Appendix A. XDuce code for the example *dvdstore*

```
(* dvdstore.q *)
type Store = store[Regulars, Discounts]
type Regulars = Dvd1*
type Discounts = Dvd2, Dvd2*
type Dvd1 = dvd[(Title, Price)]
type Dvd2 = dvd[(Title, Price, Discount)]
type Title = title[String]
type Price = price[String]
type Discount = discount[String]
type Mydvds = Mydvd*
type Mydvd = mydvd[(Title, Price, Discount)]

fun main (val fn as String): Mydvds = match load_xml(fn) with
      store[val dvds as (Dvd1*,Dvd2,Dvd2* )] -> mkmyDvd1(dvds)
      | Any -> raise("Error")

fun mkmyDvd1 (val dvds as (Dvd1*,Dvd2,Dvd2* )): Mydvds =
  match dvds with
      dvd[title[val t], price[val p]], val rest
        -> mydvd[title[t], price[p],discount["0.5"]], mkmyDvd1(rest)
      | dvd[title[val t], price[val p], discount[val d]], val rest
        -> mydvd[title[t], price[p], discount[d]], mkmyDvd2(rest)

fun mkmyDvd2 (val dvd2s as Dvd2* ): Mydvds =
  match dvd2s with
      dvd[title[val t], price[val p], discount[val d]], val rest
        -> mydvd[title[t], price[p], discount[d]], mkmyDvd2(rest)
      | () -> ()

let val _ = main("dvdstore.xml")
```

Appendix B. XDuce code for the example *addrbook* (CDuce code is ommtited)

```
(* addrbook.q *)
type Addrbook = addrbook[Person*]
type Person = person[(Name,Tel?,Email* )]
type Name = name[String]
type Tel = tel[String]
type Email = email[String]
type Entry = entry[(Name,Tel)]*

fun main (val fn as String) : Entry =
  match load_xml(fn) with
    addrbook[val ps as Person*] -> mkTelbook(ps)
    | Any -> raise("Error")

fun mkTelbook (val ps as Person* ) : Entry =
  match ps with
    person[name[val n], tel[val t], val e], val rest
      -> entry[name[n], tel[t]], mkTelbook(rest)
    | person[name[val n], val e], val rest -> mkTelbook(rest)
    | () -> ()

let val _ = main("large-xduce.xml")
```