Linear Time Membership in a Class of Regular Expressions with Counting, Interleaving and Unordered Concatenation

DARIO COLAZZO, Université Paris-Dauphine - PSL Research University - CNRS, France
GIORGIO GHELLI, Università di Pisa, Italy
CARLO SARTIANI, Università della Basilicata, Italy

Regular Expressions (REs) are ubiquitous in database and programming languages. While many applications make use of REs extended with *interleaving* (*shuffle*) and *unordered concatenation* operators, this extension badly affects the complexity of basic operations, and, especially, makes *membership* NP-hard, which is unacceptable in most practical scenarios.

In this paper we study the problem of membership checking for a restricted class of these extended REs, called *conflict-free REs*, which are expressive enough to cover the vast majority of real-world applications. We present several polynomial algorithms for membership checking over conflict-free REs. The algorithms are all polynomial and differ in terms of adopted optimization techniques, and in the kind of supported operators. As a particular application, we generalize the approach in order to check membership of XML trees into a class of EDTDs which models the crucial aspects of DTDs and XSD schemas.

Results about an extensive experimental analysis validate the efficiency of the presented membership checking techniques.

CCS Concepts: • **Information systems** → *XQuery*; Extensible Markup Language (XML); • **Theory of computation** → *Regular languages*;

Additional Key Words and Phrases: Regular expressions, word membership, XML

ACM Reference Format:
Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Linear Time Membership in a Class of Regular Expressions with Counting, Interleaving and Unordered Concatenation. *ACM Trans. Datab. Syst.* 1, 1 (November 2017), 45 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Regular expressions are ubiquitous in database and programming languages. They represent a fundamental programming tool for identifying and extracting data from text, and are the basis over which mainstream schema and query languages for semistructured data are built. Given the important role they have always played in computer science, regular expressions have been the subject of research activities for decades. More recently, the

Authors' addresses: Dario Colazzo, Université Paris-Dauphine - PSL Research University - CNRS , LAMSADE, Place du Maréchal de Lattre de Tassigny, Paris, 75016, France, dario.colazzo@dauphine.fr; Giorgio Ghelli, Università di Pisa, Dipartimento di Informatica, Largo Bruno Pontecorvo, 3, Pisa, 56127, Italy, ghelli@di.unipi.it; Carlo Sartiani, Università della Basilicata, Dipartimento di Matematica, Informatica ed Economia, Via dell'Ateneo Lucano, 10, Potenza, 85100, Italy, sartiani@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
0362-5915/2017/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

research community has focused renewed efforts in studying families of regular expressions extended with counting and interleaving operators, that find many applications in processing and querying textual as well as tree- and graph-shaped data [1, 9, 11, 29, 30, 34, 43]. A counting operator $T\,[m..n]$ specifies a minimum (m) and a maximum (n) number of repetitions of a given regular expression T, while an interleaving operator $T_1\&T_2$ shuffles the words of T_1 and T_2. The addition of these operators to plain regular expressions enhances the flexibility of the language, as it allows one to define exponentially more succinct regular expressions, but also has an impact on the complexity of decision problems. One problem whose complexity is strongly affected by the presence of these operators is *membership checking*, that is checking whether a word can be generated by a given regular expression.

More precisely, given a class $\mathcal{R}$ of regular expressions, the membership problem for $\mathcal{R}$ is that of checking whether a word w belongs to the language generated by an expression e in $\mathcal{R}$. The problem is polynomial in the size of w and of e when $\mathcal{R}$ is just the class of standard regular expressions with union, concatenation, and Kleene star, which we denote as RE [31], and is still polynomial when intersection $\cap$ is added to the operators. When interleaving comes into play, however, membership becomes at least NP-hard: indeed, as proved by Mayer and Stockmeyer in [36], membership is NP-complete for RE(&), i.e., RE extended with interleaving, and remains NP-hard when only the combinations of & and concatenation, & and union, & and Kleene star are considered. This shows that the NP-hardness of membership with interleaving is quite robust.

Another operator that has been studied in the literature is *unordered concatenation* (%). It has been first used in SGML (Standard Generalized Markup Language) [28], and later in a limited form in XML Schema [22], the main XML schema language. Unordered concatenation can be seen as a restricted form of interleaving: while the expression $T_1\&T_2$ contains all the words obtained by *shuffling* words in T_1 and T_2, the expression $T_1\%T_2$ is equivalent to $T_1 \cdot T_2 + T_2 \cdot T_1$, where + and $\cdot$ denote, respectively, union and concatenation. Of course, this rewriting is not of practical interest in the n-ary case ($T_1\% \ldots \%T_n$ produces an exponential explosion of the corresponding union type). Another difference from & is that % is not associative. As shown by Hovland in [30], membership is NP-complete for regular expressions using counting and unordered concatenation.

Our contribution. In this paper we study the membership problem for a restricted class of regular expressions with counting, interleaving and unordered concatenation, and show that this class admits membership checking in linear time. The family of regular expressions that we consider contains those expressions where no symbol appears twice (*single-occurrence*), and where repetition is only applied to single symbols; Kleene star is generalized to counting constraints such as $a\,[1..*]$ and $a\,[2..5]$. We introduced this class of regular expressions, called *conflict-free*, in [20, 25] and exploited it to lower the complexity of inclusion [16–19]. As shown in [8, 15], the two main restrictions behind conflict-freedom (single-occurrence and single-symbol repetition) characterize the vast majority of regular expressions defined by users in practice.

Our approach is based on the translation of each conflict-free regular expression into a set of constraints through a linear-time translation algorithm. By relying on this translation, our membership checking algorithms verify if a word belongs to the language generated by a regular expression by verifying if it satisfies the constraints describing the expression. These algorithms are based on the implicit representation of the constraints using a tree structure, and on the parallel verification of all constraints, using a *residuation* technique. The residuation technique transforms each constraint into the constraint that has to be

verified on the rest of the word after a symbol has been read; this *residual constraint* is computed in constant time. The notion of *residuation* is strongly reminiscent of Brzozowski's derivative of REs [14].

In this paper we present four membership algorithms for conflict-free regular expressions. The first one, called MEMBER, assumes that operators of regular expressions are all binary, and excludes unordered concatenation (which needs an n-ary representation as it is not associative), hence applying to conflict-free expressions with interleaving (&) and counting (#) only.[1] For each regular expression T the algorithm builds in linear time a binary tree reflecting its parse tree and whose nodes are labeled with constraints associated to T to be residuated, while leaves correspond to unique symbols occurring in it. The algorithm requires a bottom up visit of the tree for each symbol in a word w checked for membership, and has $O(|T| + |w| * depth(T))$ time complexity. The design and study of this algorithm revealed that a kind of *stability* property holds: once a given symbol a has been parsed, there is no need to redo a bottom-up visit of the tree when the symbol is met again, except for one specific case where the new occurrence of the symbol makes membership checking fail. We prove stability for the binary algorithm and exploit it to develop a new linear membership algorithm MEMBERSTAB that runs in $O(|T| + |w|)$.

We then consider the case where operators in regular expressions are n-ary operators, and this time we include unordered concatenation, hence considering conflict-free expressions in RE(#, &, %). In this setting of *flat* regular expressions, we define new constraints, prove their soundness and completeness, and define the associated residuation technique. We used these constraints to define the membership algorithm MEMBERFLAT. This algorithm can deal with unordered concatenation, and runs in $O(|T| + |w| * depth(T))$ time, which is formally the same as the complexity of the base MEMBER algorithm, but it is in practice much better, since the factor $depth(T)$ is usually much lower for flat expressions than for binary expressions. We then identify a stability property for the new family of *flat* constraints, and use it to define a membership algorithm MEMBERFLATSTAB with linear complexity $O(|T| + |w|)$.

We then extend our approach to the validation of XML trees against XML schemas (Extended DTDs) that use conflict-free regular expressions only, producing an algorithm that runs in time $O(|T| + |w| * depth(T))$, where w is the XML tree, T is the XML schema, and $depth(T)$ is the maximal depth of all the regular expressions that appear in the schema.

Finally, we perform an experimental study of our algorithms. We would have liked to compare them with standard automata-based approaches, but this was not possible since we found no tools supporting both counting and interleaving, with the notable exception of Anders Møller's automaton library [37]. However, this library maps regular expressions with counting and interleaving into DFAs incurring in an exponential blow up of the size of automaton, hence it cannot deal with the size of our test suite. Hence, we compare our algorithm with an alternative baseline approach based on Brzozowski's derivatives [14] for expressions in cf-RE(#, &) and cf-RE(#, &, %). The results of our experiments are coherent with the theoretical analysis, showing that the MEMBERFLATSTAB is indeed superior to the simpler variants, and order of magnitudes faster than the baseline.

Paper Outline. The paper is organized as follows. In Section 2 we describe the type and constraint languages we are using, and show how types can be represented in terms of constraints. In Section 3, then, we illustrate the basic principles of our residuation technique.

[1] We use RE(#, &) to denote the class of unrestricted regular expressions with interleaving and counting; furthermore, given a class of regular expressions $\mathcal{R}$, cf-$\mathcal{R}$ will denote the subclass of conflict-free regular expressions in $\mathcal{R}$.

In Sections 4 and 5 we describe four algorithms based on the residuation technique of Section 3. In Section 6 we show how our algorithms can be applied to XSD schemas. In Section 7, then, we validate our algorithms with an extensive experimental analysis. In Section 8 we describe related work. In Section 9, finally, we draw our conclusions.

2 TYPE AND CONSTRAINT LANGUAGE

This section offers preliminary definitions and related properties that we introduced in [20] and form the basis of our membership algorithms.

2.1 The Type Language

We follow the terminology of [20], and use the term *type* instead of *regular expression*. We consider the following type language with counting, disjunction, concatenation, and interleaving, for words over a finite alphabet Σ; we use ϵ for the empty word and for the expression whose language is $\{\epsilon\}$, while $a\,[m..n]$, with $a \in \Sigma$, contains the words composed by j repetitions of a, with $m \leq j \leq n$.

$$T ::= \quad \epsilon \;\mid\; a\,[m..n] \;\mid\; T + T \;\mid\; T \cdot T \;\mid\; T \& T$$

More precisely, we define $\mathbb{N}_* = \mathbb{N} \cup \{*\}$, and extend the standard order among naturals with $n \leq *$ for each $n \in \mathbb{N}_*$. In every type expression $a\,[m..n]$ we have that $m \in (\mathbb{N} \setminus \{0\})$, $n \in (\mathbb{N}_* \setminus \{0\})$, and $n \geq m$. Specifically, $a\,[0..n]$ is not part of the language, but we use it to abbreviate $(\epsilon + a\,[1..n])$. The operator $_\,[m..n]$ generalizes Kleene star, but can only be applied to symbols. Unbounded repetition of a disjunction of symbols, i.e., $(a_1 + \ldots + a_n)*$, can be expressed as $((a_1\,[0..*]) \& \ldots \& (a_n\,[0..*]))$.

Word concatenation $w_1 \cdot w_2$ and language concatenation $L_1 \cdot L_2$ are standard. The *shuffle*, or *interleaving*, operator $w_1 \& w_2$ is defined as follows.

Definition 2.1 ($v\&w,\ L_1\&L_2$). The shuffle set of two words $v, w \in \Sigma^*$, or two languages $L_1, L_2 \subseteq \Sigma^*$, is defined as follows; notice that each v_i or w_i may be the empty word ϵ.

$$\begin{array}{lll} v\&w & =_{def} & \{\ v_1 \cdot w_1 \cdot \ldots \cdot v_n \cdot w_n \ \mid\ v_1 \cdot \ldots \cdot v_n = v,\ w_1 \cdot \ldots \cdot w_n = w, \\ & & \qquad\qquad\qquad\qquad\quad v_i \in \Sigma^*,\ w_i \in \Sigma^*,\ n > 0\ \} \\ L_1\&L_2 & =_{def} & \bigcup_{w_1 \in L_1,\ w_2 \in L_2} w_1\&w_2 \end{array}$$

Example 2.2. $(ab)\&(XY)$ contains the permutations of $abXY$ where a comes before b and X comes before Y:

$$(ab)\&(XY) = \{abXY, aXbY, aXYb, XabY, XaYb, XYab\}$$

Definition 2.3 ($S(w), S(T), Atoms(T)$). For any word w and for any type T:

(1) $S(w)$ is the set of all symbols appearing in w;
(2) $Atoms(T)$ is the set of all atoms $a\,[m..n]$ appearing in T;
(3) and $S(T)$ is the set of all alphabet symbols appearing in T.

Semantics of types is defined as follows.

$$\begin{array}{rcl} [\![\epsilon]\!] & = & \{\epsilon\} \\ [\![a\,[m..n]]\!] & = & \{w \mid S(w) = \{a\},\ |w| \geq m,\ |w| \leq n\} \\ [\![T_1 + T_2]\!] & = & [\![T_1]\!] \cup [\![T_2]\!] \\ [\![T_1 \cdot T_2]\!] & = & [\![T_1]\!] \cdot [\![T_2]\!] \\ [\![T_1 \& T_2]\!] & = & [\![T_1]\!] \& [\![T_2]\!] \end{array}$$

We will use $\otimes$ to range over $\cdot$ and & when we need to specify common properties, such as: $[\![T \otimes \epsilon]\!] = [\![\epsilon \otimes T]\!] = [\![T]\!]$.

In this system, no type is empty. Some types contain the empty word ϵ (are *nullable*), and are characterized as follows.

Definition 2.4 ($\mathrm{N}(T)$*).* $\mathrm{N}(T)$ is a predicate on types, defined as follows:

$$\begin{array}{rcl} \mathrm{N}(\epsilon) & = & \text{true} \\ \mathrm{N}(a\,[m..n]) & = & \text{false} \\ \mathrm{N}(T + T') & = & \mathrm{N}(T) \text{ or } \mathrm{N}(T') \\ \mathrm{N}(T \otimes T') & = & \mathrm{N}(T) \text{ and } \mathrm{N}(T') \end{array}$$

Lemma 2.5. *$\epsilon \in [\![T]\!]$ iff $\mathrm{N}(T)$.*

Proof. Trivial: it directly follows by induction from Definition 2.4. □

We define now the notion of *conflict-free* types.

Definition 2.6 (Conflict-free types). A type T is *conflict-free* if for each subexpression $(U + V)$ or $(U \otimes V)$: $S(U) \cap S(V) = \emptyset$.

Equivalently, a type T is conflict-free if, for any two distinct subterm occurrences $a\,[m..n]$ and $a'\,[m'..n']$ in T, a is different from a'.

Remark 1. *The class of grammars we study is quite restrictive, because of the* conflict-free *limitation and of the constraint on Kleene-star. However, similar, or stronger, constraints, have been widely studied in the context of DTDs [10] and XSD schemas, and it has been discovered that the vast majority of real-life expressions do respect them.*

Conflict-free REs have been studied, for example, as "duplicate-free" DTDs in [38, 45], as "Single Occurrence REs" (SOREs) in [6, 7], as "conflict-free DTDs" in [3, 4]. The specific limitation that we impose on Kleene-star is reminiscent of Chain REs (CHAREs), as defined by Bex et al. in [6], which are slightly more restrictive. That paper states that "an examination of the 819 DTDs and XSDs gathered from the Cover Pages (including many high-quality XML standards) as well as from the web at large, reveals that more than 99% of the REs occurring in practical schemas are CHAREs (and therefore also SOREs)". Barbosa et al., on the basis of a corpus of 26604 content models from `xml.org`*, measure that 97,7% are conflict-free, and 94% are conflict-free and* simple*, where* simple *is a restriction much stronger than our Kleene-star restriction [3]. Similar results about the prevalence of* simple *content models had been reported by Choi in [15].*

Hereafter, we will silently assume that every type is conflict-free, although some of the properties we specify are valid for any type.

We show now how the semantics of a type T can be expressed by a set of constraints. This alternative characterization of type semantics will then be used for membership checking.

2.2 The Constraint Language

The semantics of conflict-free types can be fully captured by a set of constraints on words. To illustrate the intuition behind this, consider the type $T = ((a\,[1..3] \cdot b\,[2..2]) + c\,[1..*])$, which can be represented by the following constraints: $abc^+ \wedge \mathrm{upper}(abc) \wedge a?[1..3] \wedge b?[2..2] \wedge c?[1..*] \wedge a^+ \Rightarrow b^+ \wedge b^+ \Rightarrow a^+ \wedge a \prec b \wedge a \prec c \wedge c \prec a \wedge b \prec c \wedge c \prec b$. Constraints express the following properties of each word w in $[\![T]\!]$: the constraint abc^+ indicates that w must have at least one symbol in $\{a, b, c\}$, while the upper bound constraint $\mathrm{upper}(abc)$ indicates that

each symbol of w must be in $\{a, b, c\}$. The constraint $a?[1..3]$ indicates that if a is in w then a must appear at least once and at most three times, and we have similar constraints for b and c. The constraint $a \prec b$ indicates that it is never the case that one occurrence of b appears after an occurrence of a. The couple $a \prec c$ and $c \prec a$ indicates mutual exclusion between a and c, since no order would be compatible with both constraints, and similarly for the pair $b \prec c$ and $c \prec b$. Finally, $a^+ \Mapsto b^+$ and $b^+ \Mapsto a^+$ respectively say that, if a occurs in w, then b is in w too and vice-versa. As shown in [20] these constraints exactly capture the semantics of T, and constraints can be extracted in polynomial time from the type.

Constraints are expressed using the following logic, where $a, b \in \Sigma$ and $A, B \subseteq \Sigma$, $m \in (\mathbb{N} \backslash \{0\})$, $n \in (\mathbb{N}_* \backslash \{0\})$, and $n \geq m$:

$$F \ ::= \quad A^+ \ \mid \ A^+ \Mapsto B^+ \ \mid \ a?[m..n] \ \mid \ \mathrm{upper}(A) \ \mid \ a \prec b \ \mid \ F \wedge F' \ \mid \ \mathbf{true}$$

Satisfaction of a constraint F by a word w, written $w \models F$, is defined as follows.

$$\begin{array}{rcl}
w \models a?[m..*] & \Leftrightarrow & \text{if } a \text{ appears in } w\text{, then it appears at least } m \text{ times} \\
w \models a?[m..n] & \Leftrightarrow & \text{if } a \text{ appears in } w\text{, then it appears at least } m \text{ times} \\
(n \neq *) & & \text{and at most } n \text{ times} \\
w \models a \prec b & \Leftrightarrow & \text{there is no occurrence of } a \text{ in } w \text{ that follows an occurrence} \\
 & & \text{of } b \text{ in } w \text{ (hence, both } a \text{ and } b \text{ may be missing)} \\
w \models A^+ & \Leftrightarrow & (S(w) \cap A) \neq \emptyset\text{, i.e., some } a \in A \text{ appears in } w \\
w \models A^+ \Mapsto B^+ & \Leftrightarrow & w \not\models A^+ \text{ or } w \models B^+ \\
w \models \mathrm{upper}(A) & \Leftrightarrow & S(w) \subseteq A \\
w \models F_1 \wedge F_2 & \Leftrightarrow & w \models F_1 \text{ and } w \models F_2 \\
w \models \mathbf{true} & \Leftrightarrow & \text{always}
\end{array}$$

We use the following abbreviations:

$$\begin{array}{rcl}
A^+ \Leftrightarrow B^+ & =_{def} & A^+ \Mapsto B^+ \wedge B^+ \Mapsto A^+ \\
a <> b & =_{def} & (a \prec b) \wedge (b \prec a) \\
A \prec B & =_{def} & \bigwedge_{a \in A,\, b \in B} a \prec b \\
A <> B & =_{def} & \bigwedge_{a \in A,\, b \in B} a <> b \\
\mathbf{false} & =_{def} & \emptyset^+ \\
A^- & =_{def} & A^+ \Mapsto \emptyset^+
\end{array}$$

The next propositions specify that $A <> B$ encodes mutual exclusion between sets of symbols, and that A^- denotes the absence of any symbol in A.

Proposition 2.7. $w \models a <> b \ \Leftrightarrow w \not\models (\{a\}^+ \wedge \{b\}^+)$, *which means that a and b are mutually exclusive in $S(w)$.*

Proof. Trivial. □

Proposition 2.8. $w \models A <> B \ \Leftrightarrow \ w \not\models (A^+ \wedge B^+)$

Proof. By definition of $A <> B$ and by Proposition 2.7, we observe that $w \models A <> B$ means that for each $a \in A$, $b \in B$ it is the case that $\{a, b\} \not\subseteq S(w)$; hence, if one element of A

is in w no element of B may be there, and vice versa. This is the same property expressed by $w \not\models A^+ \wedge B^+$. □

Proposition 2.9. $w \models A^- \;\Leftrightarrow\; w \not\models A^+$

Proof. Trivial. □

2.3 Constraint Extraction

We can now define the extraction of constraints from types.

We start with those constraints whose definition is *flat*, since they only depend on the leaves of the syntax tree of T. Flat constraints formalize the following observations:

- *lower-bound*: unless T is nullable (i.e., unless N(T)), w must include one symbol of $S(T)$;
- *upper-bound*: no symbol $a \notin S(T)$ may appear in w;
- *cardinality*: if a symbol in $S(T)$ appears in w, it must appear with the right cardinality.

Hereafter we abbreviate the constraint $(S(T))^+$, that requires the presence of one symbol from type T, with $S^+(T)$.

Definition 2.10 (Flat constraints).

$$\begin{array}{rrcl} \text{Lower-bound:} & SIf(T) & =_{def} & \begin{cases} S^+(T) & \text{if not N}(T) \\ \textbf{true} & \text{if N}(T) \end{cases} \\ \text{Cardinality:} & \text{ZeroMinMax}(T) & =_{def} & \bigwedge_{a[m..n]\in Atoms(T)} a?[m..n] \\ \text{Upper-bound:} & \text{upperS}(T) & =_{def} & \text{upper}(S(T)) \\ \text{Flat constraints:} & \mathcal{FC}(T) & =_{def} & SIf(T) \;\wedge\; \text{ZeroMinMax}(T) \wedge\; \text{upperS}(T) \end{array}$$

We add now the *nested constraints*, whose definition depends on the internal structure of T. Nested constraints formalize the properties that we list below. All nested constraints depend on the fact that T is conflict-free. The quantification "for any $w \in [\![C[T]]\!] \ldots$" that we use below means "for any $w \in [\![T']\!]$ where T' is any type with a subterm $T\ldots$". All properties are quite obvious if one ignores the "for any $C[_]$" quantification. This context quantification is a consequence of the fact that types are conflict-free, and expresses the fact that any nested constraint that holds for a subterm of the type also holds for the whole type. These are the properties:

- *co-occurrence*: for any $w \in [\![C[T_1 \otimes T_2]]\!]$:
  - unless T_2 is nullable, if a symbol in $S(T_1)$ is in w, then a symbol in $S(T_2)$ is in w as well;
  - unless T_1 is nullable, if a symbol in $S(T_2)$ is in w, then a symbol in $S(T_1)$ is in w as well;
- *order*: for any $w \in [\![C[T_1 \cdot T_2]]\!]$, no symbol in $S(T_1)$ may follow a symbol in $S(T_2)$;
- *exclusion*: for any $w \in [\![C[T_1 + T_2]]\!]$, it is not possible that w has a symbol that belongs to both $S(T_1)$ and $S(T_2)$.

Co-occurrence is not obvious because of the double condition 'unless T_2 is nullable' and 'if a symbol in $S(T_1)$ is in w'. Consider a word w in $[\![(T_1 \otimes T_2) + T_3]\!]$. If a symbol of T_1 is in w, then w has been generated by $T_1 \otimes T_2$, rather than by the context $_ + T_3$. Since it comes from $[\![T_1 \otimes T_2]\!]$, then a symbol of T_2 should appear in w, unless T_2 is nullable. This is captured by the co-occurrence constraint.

In the formal definition that follows, $\mathit{If}_{T_2}(S^+(T_1) \Rrightarrow S^+(T_2))$ denotes **true** when $\mathrm{N}(T_2)$, and $(S(T_1))^+ \Rrightarrow (S(T_2))^+$ otherwise, that is, if T_2 is not nullable then $(S(T_1))^+ \Rrightarrow (S(T_2))^+$. The relation '$T'$ subterm of T', used in the definition of $\mathcal{NC}(T)$, is the standard subterm relation and is reflexive. Observe that the exclusion constraints are actually encoded by means of order constraints.

Definition 2.11 (Nested constraints).

Co-occurrence:

$$\begin{array}{rll} \mathcal{CC}(T_1 \otimes T_2) & =_{def} & \mathit{If}_{T_2}(S^+(T_1) \Rrightarrow S^+(T_2)) \wedge \mathit{If}_{T_1}(S^+(T_2) \Rrightarrow S^+(T_1)) \\ \mathcal{CC}(T) & =_{def} & \mathbf{true} \qquad \text{in any other case} \end{array}$$

Order/exclusion:

$$\begin{array}{rll} \mathcal{OC}(T_1 + T_2) & =_{def} & S(T_1) <> S(T_2) \\ \mathcal{OC}(T_1 \cdot T_2) & =_{def} & S(T_1) \prec S(T_2) \\ \mathcal{OC}(T) & =_{def} & \mathbf{true} \qquad \text{in any other case} \end{array}$$

Nested constraints:

$$\mathcal{NC}(T) =_{def} \bigwedge_{T_i \text{ subterm of } T} (\mathcal{CC}(T_i) \wedge \mathcal{OC}(T_i))$$

As a consequence of the above definition, nested constraints have the following property, which may also be used as an alternative definition of nested constraints.

PROPOSITION 2.12 ($\mathcal{NC}(T)$).

$$\begin{array}{rll} (i)\ \mathcal{NC}(T_1 + T_2) & = & (S(T_1) <> S(T_2)) \wedge \mathcal{NC}(T_1) \wedge \mathcal{NC}(T_2) \\ (ii)\ \mathcal{NC}(T_1 \& T_2) & = & \mathit{If}_{T_2}(S^+(T_1) \Rrightarrow S^+(T_2)) \wedge \mathit{If}_{T_1}(S^+(T_2) \Rrightarrow S^+(T_1)) \wedge \mathcal{NC}(T_1) \wedge \mathcal{NC}(T_2) \\ (iii)\ \mathcal{NC}(T_1 \cdot T_2) & = & (S(T_1) \prec S(T_2)) \wedge \\ & & \mathit{If}_{T_2}(S^+(T_1) \Rrightarrow S^+(T_2)) \wedge \mathit{If}_{T_1}(S^+(T_2) \Rrightarrow S^+(T_1)) \wedge \mathcal{NC}(T_1) \wedge \mathcal{NC}(T_2) \end{array}$$

$$(iv)\ \mathcal{NC}(\epsilon) = \mathcal{NC}(a\,[m..n]) = \mathbf{true}$$

PROOF. Property (i) derives from $\mathcal{CC}(T_1 + T_2) = \mathbf{true}$, and (ii) from $\mathcal{OC}(T_1 \& T_2) = \mathbf{true}$. Properties (iii) and (iv) directly follow from Definition 2.11. □

By definition, when either A or B is "$\emptyset$", both $A \prec B$ and $A <> B$ are **true**, hence the order constraint associated to a node where one child has $S(T_i) = \emptyset$ is trivial; this typically happens with a subterm $T + \epsilon$.

Example 2.13. Consider the type $T = ((a + \epsilon)\&b\,[1..5]) \cdot (c + d\,[1..*])$, where we use a to abbreviate $a\,[1..1]$. The application of the extraction rules to this type yields the conjunction of the following constraints (see Figure 1):

- CC: $a^+ \Rrightarrow b^+ \ \wedge\ ab^+ \Leftrightarrow cd^+$, where ab^+ and cd^+ stand for $\{a, b\}^+$ and $\{c, d\}^+$, respectively
- OC: $c <> d \ \wedge\ ab \prec cd$
- Flat:

$$\begin{array}{ll} abcd^+ \wedge & \text{Lower-bound} \\ a?[1..1] \wedge b?[1..5] \wedge c?[1..1] \wedge d?[1..*] \wedge & \text{Cardinality} \\ \mathrm{upper}(abcd) & \text{Upper-bound} \end{array}$$

The following theorem (that we proved in [20]) states that constraints provide a sound and complete characterization of type semantics.

Table 1. Computing the residual of a nested co-occurrence constraint.

Condition	$a \in A$	$a \in B$	$a \in A$	$a \in B$	$a \in A$
Constraint	$A^+ \Rrightarrow B^+$	$A^+ \Rrightarrow B^+$	$A^+ \Leftrightarrow B^+$	$A^+ \Leftrightarrow B^+$	A^+
Residual	B^+	**true**	B^+	A^+	**true**

Table 2. Computing the residual of a nested order constraint.

Condition	$a \in A$	$a \in B$	$a \in A$	$a \in B$	$a \in A$
Constraint	$A \prec B$	$A \prec B$	$A <> B$	$A <> B$	A^-
Residual	$A \prec B$	A^-	B^-	A^-	**false**

Theorem 2.14. *Given a conflict-free type T, it holds that:*

$$w \in [\![T]\!] \quad \Leftrightarrow \quad w \models \mathcal{FC}(T) \wedge \mathcal{NC}(T)$$

Proof. See proof of Theorem 16 in [20]. □

This theorem allows us to reduce membership in T to the verification of $\mathcal{FC}(T) \wedge \mathcal{NC}(T)$.

3 BASIC RESIDUATION ALGORITHM

We first present an algorithm to decide membership of a word w in a type T in time $O(|w| * depth(T))$, where $depth(T)$ is defined as $depth(\epsilon) = depth(a\,[m..n]) = 1$, $depth(T_1 \circledast T_2) = 1 + \max(depth(T_1), depth(T_2))$, with $\circledast \in \{+, \cdot, \&\}$. The algorithm verifies whether the word satisfies all the constraints associated with T, through a linear scan of w. The basic observation is that every symbol a of w transforms each constraint F into a *residual constraint* F', to be satisfied by the subword w' that follows the symbol. We write $F \xrightarrow{a} F'$ to specify that F is transformed (or residuated) into F' by a. Residuation is defined in Tables 1 and 2 for co-occurrence and order constraints, respectively. In all the cases not covered by Tables 1 and 2, we have that $F \xrightarrow{a} F$. We apply residuation to the nested constraints only, since flat constraints can be checked in linear time by just counting the occurrences of each symbol in the word.

Residuation $F \xrightarrow{a} F'$ is extended from symbols to words $F \xrightarrow{w}{}^{*} F'$ in the obvious way:

$$F \xrightarrow{\epsilon}{}^{*} F$$
$$F \xrightarrow{aw}{}^{*} F'' \quad \Leftrightarrow_{\text{def}} \quad F \xrightarrow{a} F' \ \wedge \ F' \xrightarrow{w}{}^{*} F''$$

Observe that, by construction, residuation is a total function, that is, for every F and w, there always exists a unique residual constraint F' such that $F \xrightarrow{w}{}^{*} F'$.

When a word has been read up to the end, the residual constraint is satisfied iff it is satisfied by ϵ, that is, if it is different from A^+ or **false**. This is formalized by the relation $F \downarrow^{w} G$, defined below, with $G \in \{\textbf{true}, \textbf{false}\}$, where G, again, is uniquely determined by F and w.

Definition 3.1 ($F \downarrow^{w} G$).

$$F \downarrow^{w} \textbf{true} \quad \Leftrightarrow_{\text{def}} \quad (F \xrightarrow{w}{}^{*} F' \ \wedge \ \epsilon \models F')$$

$$F \downarrow^{w} \textbf{false} \quad \Leftrightarrow_{\text{def}} \quad (F \xrightarrow{w}{}^{*} F' \ \wedge \ \epsilon \not\models F')$$

The following lemma specifies that residuation corresponds to the semantics of our constraints.

Lemma 3.2 (Residuation). $w \models F$ *iff* $F \downarrow^w$ **true**.

Proof. We first prove that $aw \models F$ iff $F \xrightarrow{a} F' \ \wedge\ w \models F'$.
We reason by cases on F and a. Let us first consider $F = A^+ \mapsto B^+$.
By definition of $aw \models A^+ \mapsto B^+$, if a belongs to A, then $aw \models A^+ \mapsto B^+$ iff $w \models B^+$.
If a belongs to B, then $aw \models A^+ \mapsto B^+$ holds.
Finally, if a does not belong to either A or B, then $aw \models A^+ \mapsto B^+$ iff $w \models A^+ \mapsto B^+$.
These are exactly the residuation rules for $A^+ \mapsto B^+$ in Table 1 - the identity rule for the case when a does not belong to either A or B is implicit, since $F \xrightarrow{a} F$ holds for any case not explicitly covered in the table.
The analysis of all the other constraints is equally trivial.
By induction, property $aw \models F \ \Leftrightarrow\ (F \xrightarrow{a} F' \ \wedge\ w \models F')$ and Definition 3.1 imply that

$$w \models F \ \Leftrightarrow\ \ (F \xrightarrow{w}{}^{*} F' \ \wedge\ \epsilon \models F')$$

□

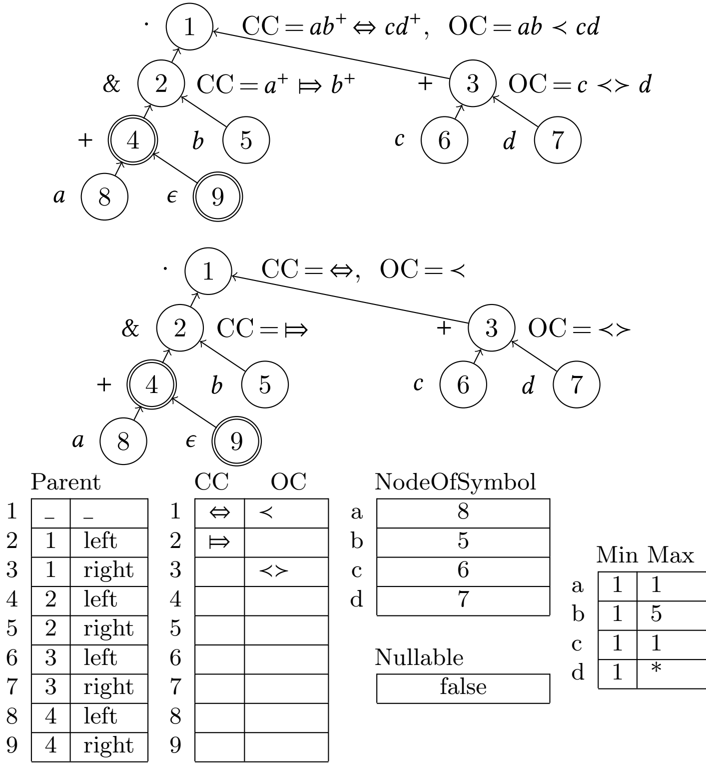
Residuation yields a membership algorithm of complexity $O(|w| * |\mathcal{NC}(T)|)$: for each symbol a in w, and for each constraint F in $\mathcal{NC}(T)$, we read a and substitute F with the residual F' such that $F \xrightarrow{a} F'$. As a consequence of the previous lemma, a word w is in T iff no **false** or A^+ is in the final set of constraints.

However, we can do much better than $O(|w| * |\mathcal{NC}(T)|)$. First of all, we do not build the constraints, but we keep them, and their residuals, implicit in a tree-shaped data structure with size $O(|T|)$. The structure initially corresponds to the syntax tree of T, that is, the tree that can be built by parsing T, encoded as a set of nodes and a *Parent*[] array, such that, for each node n, Parent[n] is either null (for the root) or a pair (n_p,*direction*), where n_p is the parent of n, while *direction* is *left* if n is the left child of n_p, and is *right* if it is the right child (Figure 1).

The constraints, and their residuals, are encoded using the following arrays, defined on the nodes of T:

- CC[]: for each node n of T, such that A_1 and A_2 are, respectively, the symbols in the left and right descendants of n, CC[n] is a symbol in the finite set $\{\Leftrightarrow, \mapsto, \Leftarrow, L^+, R^+, \mathbf{true}\}$ that specifies the status of the co-occurrence constraint for n, as follows:
  - $\Leftrightarrow$: encodes ${A_1}^+ \Leftrightarrow {A_2}^+$;
  - $\mapsto$: encodes ${A_1}^+ \mapsto {A_2}^+$;
  - $\Leftarrow$: encodes ${A_2}^+ \mapsto {A_1}^+$;
  - L^+: encodes the residual constraint A_1^+;
  - R^+: encodes the residual constraint A_2^+;
  - **true**: encodes the constraint **true**.
- OC[]: for each node n, such that A_1 and A_2 are, respectively, the symbols in the left and right descendants of n, OC[n] specifies the status of the order constraint for n, and may assume the following values:
  - $<$: encodes $A_1 < A_2$;
  - $<>$: encodes $A_1 <> A_2$;
  - L^-: encodes the residual constraint ${A_1}^-$;
  - R^-: encodes the residual constraint ${A_2}^-$;
  - **false**/**true**: encodes the constraint **false** or **true**.

We also need the following arrays defined on the symbols in Σ:



	Parent	
1	_	_
2	1	left
3	1	right
4	2	left
5	2	right
6	3	left
7	3	right
8	4	left
9	4	right

	CC	OC
1	$\Leftrightarrow$	$\prec$
2	$\Mapsto$	
3		<>
4		
5		
6		
7		
8		
9		

	NodeOfSymbol
a	8
b	5
c	6
d	7

Nullable
false

	Min	Max
a	1	1
b	1	5
c	1	1
d	1	*

Fig. 1. Syntax tree for $T = ((a + \epsilon)\&b\,[1..5]) \cdot (c + d^+)$, and the corresponding algorithmic representation; the two nullable nodes have a double line in the picture.

- NodeOfSymbol[]: NodeOfSymbol[a] is the only node n_a associated with a type $a\,[m..n]$, for some m and n, and is null if no such node exists[2];
- Min[]/Max[]: Min[a] and Max[a], when different from null, encode a constraint $a?$[Min[a]..Max[a]], and are used for flat constraints only;
- Nullable: Nullable is not an array, but is just a boolean that is *true* iff T is nullable.

Table 3 reports the constraint symbols that are initially associated with a node n that corresponds to a subterm T' of the input type T. The co-occurrence constraint depends on the nullability of T_1 and T_2. The last line is just a small optimization, where we directly write **true** rather than $A^+ \Leftrightarrow \emptyset^+$, and is typically applied to nodes $T_1 + \epsilon$.

After the constraint representation has been built, the MEMBER algorithm (Figure 3) reads each character a from the input word w, scans the ancestors of $a\,[m..n]$ in the constraint tree, residuates all the constraints in this branch, and keeps track of all the resulting A^+ constraints. At the end of w, it checks that all the A^+ constraints have been further residuated into **true** – the generation of a **false** causes an immediate failure. It also verifies that each symbol respects its cardinality constraints, using the *Count*[] array to record the cardinalities, and the *CardinalityOK* function to verify the constraints. This final check is clearly linear in w.

[2]Recall that a symbol appears at most once in each conflict-free type.

Table 3. Initialization of constraint annotations.

T'	$\mathrm{N}(T_1)$	$\mathrm{N}(T_2)$	$CC[n]$	$OC[n]$
$T1 \cdot T2$	true	true	**true**	$\prec$
$T1 \cdot T2$	true	false	$\Mapsto$	$\prec$
$T1 \cdot T2$	false	true	$\Mapsfrom$	$\prec$
$T1 \cdot T2$	false	false	$\Leftrightarrow$	$\prec$
$T1 \& T2$	true	true	**true**	**true**
$T1 \& T2$	true	false	$\Mapsto$	**true**
$T1 \& T2$	false	true	$\Mapsfrom$	**true**
$T1 \& T2$	false	false	$\Leftrightarrow$	**true**
$T1 + T2$	$S(T_1) \neq \emptyset \ \wedge \ S(T_2) \neq \emptyset$		**true**	$\prec\succ$
$T1 + T2$	$S(T_1) = \emptyset \ \vee \ S(T_2) = \emptyset$		**true**	**true**

Example 3.3. Consider again the type $T = ((a + \epsilon)\&b\,[1..5]) \cdot (c + d^+)$ from Example 2.13. Its syntax tree is reported in Figure 1.

Assume we read a word $bbac$. Figure 2 illustrates the evolution of tree constraints, together with evolutions of T nested constraints (indicated in Example 2.13), while the word is parsed. When b is read, the value of $Count[b]$ is set to 1. The node 5 is retrieved from $NodeOfSymbol[b]$, and its ancestors (*2,right*) and (*1,left*) are visited. The constraint $CC[2]$ is $\Mapsto$, the direction of b is *right*, hence the constraint becomes **true**. The constraint $CC[1]$ is $\Leftrightarrow$, the direction is *left*, hence it becomes R^+. Node 1 is also pushed into *ToCheck*, since the algorithm must eventually check that every R^+ or L^+ has been residuated to **true**. Finally, $OC[1]$ is not affected. When the next b is read, $Count[b]$ becomes 2 (we will ignore $Count[]$ for the next letters), and its ancestors are visited again, but this time none of them is changed. When a is read, $CC[2]$ is **true**, hence is not affected, $CC[1]$ is R^+, hence is not affected, and $OC[1]$ is also not affected. When c is read, its ancestors (*3,left*) and (*1,right*) are visited. $OC[3]$ becomes R^- and $OC[1]$ becomes L^-, so that the tree is now the one represented in the bottom (on the right hand-side) in Figure 2.

The algorithm now verifies that $Count[]$ respects the cardinality constraints and that every A^+ node pushed into *ToCheck* has been residuated to **true**; since both checks succeed, it returns *true*. If the word had been $bbacb\ldots$ instead, the algorithm would now find a b, visit nodes (*2,right*) and (*1,left*), and, finding a L^- in node 1, would return *false* immediately.

THEOREM 3.4 (COMPLEXITY). Member(w,T) *runs in time* $O(|T| + |w| * \mathit{depth}(T))$.

PROOF. The constraint tree can be built in time $O(|T|)$.

For every symbol a in w, the algorithm executes an inner loop, where only the nodes in the path from the node of a to the root of the type are visited. For each node in this path, a constant number of unit time operations is executed. As the length of the path is at most $\mathit{depth}(T)$, the body of the algorithm has $O(|w| * \mathit{depth}(T))$ time complexity.

The final check on *ToCheck* scans at most $|T|$ nodes, while *CardinalityOK* can be evaluated in linear time. The overall time complexity of the algorithm, hence, is $O(|T|+|w|*\mathit{depth}(T))$. □

THEOREM 3.5 (SOUNDNESS). Member(w,T) *yields* true *iff* $w \in [\![T]\!]$.

PROOF. Since, by Theorem 2.14, $w \in [\![T]\!] \Leftrightarrow w \models \mathcal{FC}(T) \wedge \mathcal{NC}(T)$, to show that Member$(w,T)$ is sound, it suffices to prove that Member(w,T) returns **true** if and only if w satisfies both the flat and the nested constraints of T. A constraint F is affected by a symbol

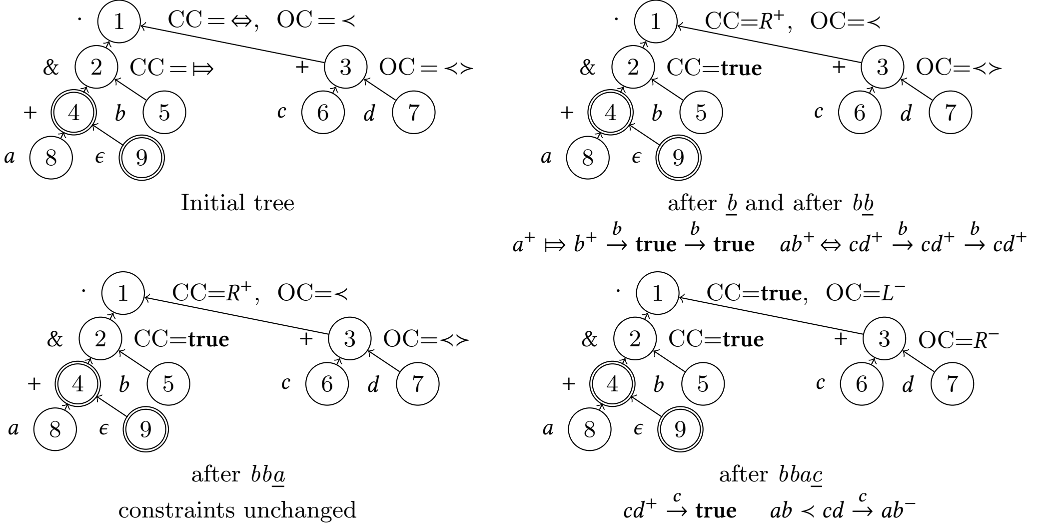


Fig. 2. The evolution of the tree for $T = ((a + \epsilon)\&b\,[1..5]) \cdot (c + d^+)$ while parsing $bbac$.

a only if a appears in F, hence, only if the node of T that corresponds to F has an $a\,[m..n]$ descendant. Hence, for each symbol of w, our algorithm residuates all the nested constraints that are affected. The final test ***exists*** *n* ***in*** *ToCheck* ***with*** *(CC[n] ≠ True)* verifies whether any A^+ constraint remains at the end, while every **false** residual causes the algorithm to immediately return *false*. For the flat constraints, the test in line 6 (**if** *NodeOfSymbol*[a] *is null*) excludes that $w \not\models \text{upperS}(T)$. When $w \models \text{upperS}(T)$, we have that $w \not\models \mathit{SIf}(T)$ iff w is empty and $\text{N}(w)$ is *false*; this is checked in line 3 (**if** *IsEmpty(w) and not Nullable*). We exclude $w \not\models \text{ZeroMinMax}(T)$ in line 30 — **if** *not CardinalityOK(Count[],Min[],Max[])*. □

4 STABILITY

We are now ready to present our linear algorithm.

The MEMBER algorithm visits all ancestors of *NodeOfSymbol*[a] every time a is found in w, which is redundant. Indeed, consider a node n such that A_1 and A_2 are, respectively, the symbols in its left and right descendants. Whenever the constraint of n has been residuated because a symbol of A_1 has been met, there is *almost* no reason to visit n again because of a symbol from A_1 (and the same holds for A_2). For example, a constraint A_1^+ is satisfied by the first $a \in A_1$. A constraint ${A_1}^+ \mapsto {A_2}^+$ or ${A_1}^+ \Leftrightarrow {A_2}^+$ is residuated to A_2^+. A constraint $A_1 <> A_2$ is residuated to ${A_2}^-$. None of them would be affected by a second symbol from A_1. There is only one exception: if the constraint is $A_1 \prec A_2$, then, even after a symbol of A_1 has already been seen, a symbol from A_2 transforms the constraint into ${A_1}^-$, and a further symbol from A_1 cannot be ignored, as it will cause the algorithm to yield “false”.

Formally, we have the following *stability* property.

LEMMA 4.1 (STABILITY). *Let F be a constraint of shape ${A_1}^+ \mapsto {A_2}^+$, ${A_1}^+ \Leftrightarrow {A_2}^+$, $A_1 <> A_2$, or $A_1 \prec A_2$, with A_1 disjoint from A_2. For each symbol a and words w and w' the following implications hold:*

A_1-stability

$$F \xrightarrow{waw'}{}^{*} F' \;\wedge \neg(F = A_1 \prec A_2 \;\wedge\; F' = {A_1}^-) \;\wedge\; a \in A_1 \quad\Rightarrow\quad \forall a' \in A_1.\; F' \xrightarrow{a'} F'$$

```
MEMBER(w, T)
 1  (Min[],Max[],NodeOfSymbol[],Parent[],CC[],OC[],Nullable) := EncodeType(T);
 2  SetToZero(Count[]);
 3  if (IsEmpty(w) and not Nullable)
 4      return (false)
 5  for a in w
 6      if (NodeOfSymbol[a] is null)
 7          return (false);
 8      Count[a] := Count[a]+1;
 9      for (n,direction) in Ancestors(NodeOfSymbol[a])
10          case (CC[n], direction)
11              when (⤇ or ⇔, left)
12                  then  CC[n] := R⁺;
13                        push(n,ToCheck);
14              when (⤆ or ⇔, right)
15                  then  CC[n] := L⁺;
16                        push(n,ToCheck);
17              when (⤆ or L⁺, left) or (⤇ or R⁺, right)
18                  then CC[n] := True;
19              else ;
20          case (OC[n], direction)
21              when (<> or <, right)
22                  then OC[n] := L⁻;
23              when (<>, left)
24                  then OC[n] := R⁻;
25              when (L⁻, left) or (R⁻, right)
26                  then return(false);
27              else ;
28  if (exists n in ToCheck with (CC[n] ≠ True))
29      return (false);
30  if (not CardinalityOK(Count[],Min[],Max[]))
31      return (false);
32  return (true);

ANCESTORS(n)
1  if (Parent[n] is null)
2      return (emptylist);
3  else return (Parent[n] ++ Ancestors([Parent[n]]));
```

Fig. 3. The basic residuation algorithm.

A_2-stability

$$F \overset{waw'}{\rightarrow}{}^{*} F' \;\wedge\; a \in A_2 \quad\Rightarrow\quad \forall a' \in A_2.\ F' \overset{a'}{\rightarrow} F'$$

PROOF. By cases on the shape of F.

Assume F has the form ${A_1}^+ \Mapsto {A_2}^+$ or ${A_1}^+ \Leftrightarrow {A_2}^+$. To prove A_1-stability, we first observe that, if $a \in A_1$, then $F \overset{wa}{\rightarrow} F'$ implies that either $F' = {A_2}^+$ or $F' = \mathbf{true}$. So, A_1-stability follows in the first case since $A_1 \cap A_2 = \emptyset$, while it trivially follows in the other case. A_2-stability follows by a similar reasoning.

Assume F has the form $A_1 <> A_2$. To prove A_1-stability, we first observe that, if $a \in A_1$, then $F \stackrel{wa}{\to} F'$ implies that either $F' = A_2{}^-$ or $F' =$ **false**. So, A_1-stability follows in the first case since $A_1 \cap A_2 = \emptyset$, while it trivially follows in the second case. A_2-stability for $A_1 <> A_2$ and $A_1 \prec A_2$ follows by a similar reasoning.

The remaining case is that F has the form $A_1 \prec A_2$ and $F' \neq A_1{}^-$. We can have that either $F' = A_1 \prec A_2$ or $F' =$ **false**. In both cases A_1-stability follows.

□

The algorithm MemberStab, described in Figure 5, exploits the above property. In a nutshell, it 'deactivates' the edges along the paths that are visited, since they do not need to be visited again, unless an $A_1 \prec A_2$ node is residuated to $A_1{}^-$, in which case the paths below A_1 are re-activated. In greater detail, when we move the first time from a node n to $Parent[n]$, the link from n to its parent is set to **null**, so that, when we arrive to n for the second time, its ancestors are not visited again. We say that the uplink from n to its parent is deleted. However, the child n is saved in $ToRestore[Parent[n]]$, so that the $ToRestore$ array contains, reversed, all the links that have been deleted. It is worth noticing that, for each node m, $ToRestore[m]$ includes at most a left child m_1 and a right child m_2 with parent links set to **null**, and that $RestoreUpLinks(m)$ undoes these uplink deletions, and is eventually recursively called on m_1 and m_2.

The $ToRestore$ structure is needed because a constraint $A_1 \prec A_2$ may be affected by a second symbol from A_1, after it has been residuated into $A_1{}^-$ by a symbol from A_2. Hence, when $A_1 \prec A_2$ becomes $A_1{}^-$, we undo any uplink deletion that took place inside A_1. Observe that all the restored uplinks now converge to the first child of a $A_1{}^-$ constraint; hence, if any of them is traversed again, a failure will immediately follow.

To illustrate, consider Figure 4 showing the evolution of the tree for $T = a \cdot (b \cdot c)$ while parsing $bbcb$. When the first b is read, the edge $(4,3)$ is traversed, the constraints associated to node 3 are residuated, and the edge is deactivated and saved in $ToRestore[3]$. Then $(3,1)$ is traversed, the constraints of 1 are residuated, and, since an L^- constraint is generated for node 1, $RestoreUpLinks(1)$ is evaluated to restore the edges in the left-hand side subtree of node 1, but no edge is actually restored as that subtree has not been visited yet; hence, after $RestoreUpLinks(1)$ has been completed, the edge $(3,1)$ is still deactivated. When the second b is read, no edge is traversed since $(4,3)$ is not active. When c is read, the traversal stops at node 3, the constraints at 3 are residuated, and $RestoreUpLinks(3)$ is evaluated, thus restoring the edge $(4,3)$; then $(5,3)$ is deactivated and saved into $ToRestore[3]$. Now, when the third b is read, the just restored edge $(4,3)$ is traversed, and the algorithm returns **false**, since node 3 is marked with L^-. The edge $(4,3)$ has been traversed three times: with the first b (deactivation), when c has been read (reactivation by $RestoreUpLinks(3)$), and finally when the last b has been read. After the third traversal, the algorithm returned **false**.

Theorem 4.2 (soundness). MemberStab(w,T) *yields* true *iff* $w \in [\![T]\!]$.

Proof. The algorithm dynamically splits the edges of the parse tree of T in two sets: the *active* edges that are stored in the *Parent[]* array and the *inactive* edges that are stored in the *ToRestore[]* array. Any time a link goes out the *Parent[]* array (line 14) it moves into the *ToRestore[]* array (line 34), and vice versa (function *RestoreUpLinks[]*). We must hence prove that any time a symbol does not affect a constraint because of an inactive link, that constraint was actually stable with respect to that symbol.

The path up from a symbol a to a constraint F crosses an inactive link only at a node n that is above a and is below F, and only if n has already been reached, meaning that another

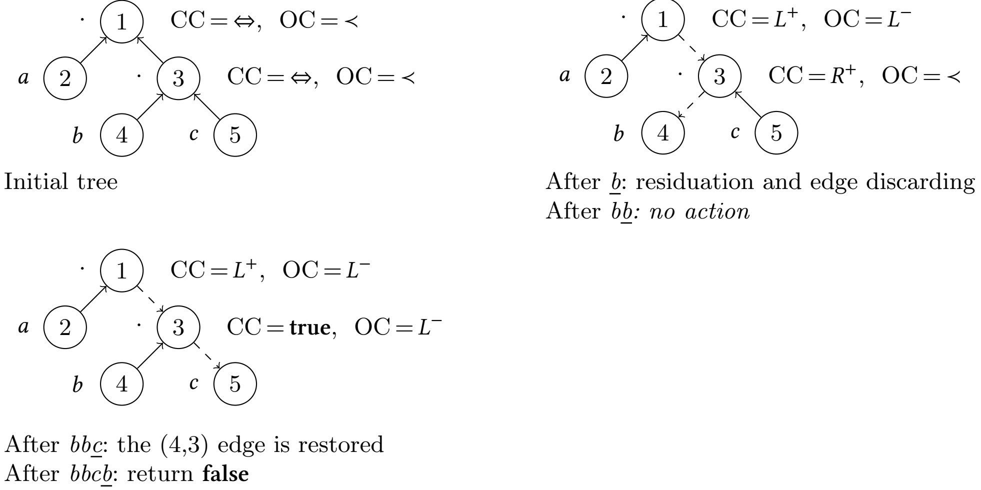


Fig. 4. The evolution of the tree for $T = a \cdot (b \cdot c)$ while parsing $bbcb$.

symbol in $S(n)$ has been used, hence all the constraints above F are now stable with respect to any symbol in $S(n)$, unless F derives from $A_1 \prec A_2$. Consider now any F' that derives from $A_1 \prec A_2$ and has a shape ${A_1}^-$. This is the only constraint that is not stable with respect to a symbol that comes from A_1 after another symbol from A_1 has been met: indeed, any symbol from A_1 must be processed, and it will cause the algorithm to return **false**. This happens since any time a constraint $A_1 \prec A_2$ is residuated to ${A_1}^-$, the algorithm restores all the deleted uplinks that are found inside A_1 (line 26). □

Proposition 4.3 (Linear complexity). *MemberStab(w,T) has $O(|T| + |w|)$ complexity.*

Proof. The algorithm has a set-up phase of cost $|T|$, as in the base algorithm. Then, for every symbol a of the word, we increment the counter of a and we follow the not-yet-deleted uplinks inside T, performing constant-time operations for each link, with the only exception of link restoration, that has constant cost for each restored link. Every link is traversed at most three times: at the first traversal it is deleted and saved; the second traversal is during a restoration phase, when it is traversed in the parent-to-child direction and restored, and in this phase, we only restore nodes that are below the left hand side of an L^- constraint. Hence, if one link is traversed again after these two visits, then a failure arises, so that no link can be traversed more than three times. Hence the phase of word-checking has a total $O(|w|)$ component related to character scanning and counter maintenance plus a total $O(|T|)$ component that is related to the less than 3 traversals of each link inside T. The final steps, where *ToCheck* is verified and *CardinalityOK* is invoked, are both linear in $\min(|T|, |w|)$ - they only involve a constant-time operation for each distinct character in $S(T) \cap S(w)$.

□

```
MemberStab(w, T)
 1  (Min[],Max[],NodeOfSymbol[],Parent[],CC[],OC[], Nullable, Symbol[]) := EncodeType(T);
 2  SetToZero(Count[]);
 3  SetToEmpty(ToRestore[]);
 4  if (IsEmpty(w) and not Nullable)
 5      return (false);
 6  for  a in w
 7      if (NodeOfSymbol[a] is null)
 8          return (false);
 9      Count[a] := Count[a]+1;
10      n:=(NodeOfSymbol[a]);
11      while (Parent[n] is not null)
12          child := n;
13          (n,direction) := Parent[n];
14          Parent[child] := null;
15          case (CC[n], direction)
16              when (⤇ or ⇔, left)
17                  then CC[n] := R^+; push(n,ToCheck);
18              when (⤆ or ⇔, right)
19                  then CC[n] := L^+; push(n,ToCheck);
20              when (⤆ or L^+, left) or (⤇ or R^+, right)
21                  then CC[n] := True;
22              else ;
23          case (OC[n], direction)
24              when (<, right)
25                  then  OC[n] := L^-;
26                        RestoreUpLinks(n)
27              when (<>, right)
28                  then  OC[n] := L^-;
29              when (<>, left)
30                  then  OC[n] := R^-;
31              when (L^-, left) or (R^-, right)
32                  then return (false);
33              else ;
34          ToRestore[n] := append((child,direction),ToRestore[n]);
35      endwhile
36  if (exists n in ToCheck with (CC[n] ≠ True))
37      return (false);
38  if (not CardinalityOK(Count[],Min[],Max[]))
39      return (false);
40  return (true);
```

```
RestoreUpLinks(n)
1  for (nc,direction) in ToRestore[n]
2      Parent[nc] := (n,direction);
3      RestoreUpLinks(nc);
4  ToRestore[n]:=();
```

Fig. 5. Basic residuation algorithm with stability.

5 FLAT MEMBERSHIP ALGORITHM

In the literature a third form of language product has been studied, apart from concatenation and interleaving. It is called *unordered concatenation*, denoted with %, whose semantics is defined as follows, where S_n is the set of all permutations of $\{1 \dots n\}$.

$$w \in [\![T_1 \% T_2 \% \dots \% T_n]\!] \iff \exists \pi \in S_n.\ w \in [\![T_{\pi(1)} \cdot T_{\pi(2)} \cdot \dots \cdot T_{\pi(n)}]\!]$$

Unordered concatenation has been studied by Hovland in [30], and differs from shuffling. As an example, $acb \in [\![(a \cdot b) \& c]\!]$, but $acb \notin [\![(a \cdot b) \% c]\!]$. The % operator is not associative, hence it is quite unnatural to combine it with binary type operators and binary constraints.

For this reason, in this section we introduce a 'flat' version of our system, that is, a version that includes % and is based on n-ary operators.

5.1 Flat constraints

We now define flat types and n-ary constraints. Flat types obey the following grammar:

$$T ::= \quad \epsilon \ \mid\ a[m..n] \ \mid\ T_1 \otimes \dots \otimes T_j \ \mid\ T_1 + \dots + T_j$$

where $\otimes$ ranges over $\{\cdot,\ \&,\ \%\}$, $m \in (\mathbb{N} \setminus \{0\})$, $n \in (\mathbb{N}_* \setminus \{0\})$, $n \geq m$ and $j \geq 2$.

The semantics for $T_1 \cdot \dots \cdot T_n$, $T_1 \& \dots \& T_n$ and $T_1 + \dots + T_n$ derives unambiguously from the binary case, since all binary operators are associative (Section 2), while semantics of $T_1 \% \dots \% T_n$ is defined as already indicated above.

Constraints are extended with a new n-ary constraint for % types. Hence, they are defined as follows, where $a \in \Sigma$ and $A, A_i \subseteq \Sigma$, $m, j \in (\mathbb{N} \setminus \{0\})$, $n \in (\mathbb{N}_* \setminus \{0\})$, and $n \geq m$:

$$F \ ::= \quad \%\{A_1, \dots, A_{j+1}\} \ \mid\ A^+ \ \mid\ A^+ \Rrightarrow B^+ \ \mid\ a?[m..n] \ \mid\ \text{upper}(A) \ \mid\ a \prec b \ \mid\ F \wedge F' \ \mid\ \mathbf{true}$$

Semantics of % constraints is defined as follows.

$$w \models \%\{A_1, \dots, A_n\} \quad \Leftrightarrow \quad \exists \pi \in S_n.\ w \models A_{\pi(1)} \prec \dots \prec A_{\pi(n)}$$

We now introduce some abbreviations, reported below. The constraint ${A_0}^+ \Rrightarrow \{{A_1}^+, \dots, {A_n}^+\}$ encodes a conjunction of n co-occurrence constraints with the same left hand side. The constraint $A_1 \prec \dots \prec A_n$ imposes an order on symbols from $A_1, \dots, A_n$ with universal quantification semantics: for each pair $i < j$, every symbol from A_i must precede every symbol from A_j, which holds trivially when the word contains no symbol from A_i or no symbol from A_j. The constraint $\prec\succ(A_1, \dots, A_m)$ encodes pairwise mutual exclusion: one symbol from A_i precludes presence of any symbol from any other set A_j, since the two symbols would violate one order constraint or the other one. The constraint $A_1 \prec \%\{A_2, \dots, A_m\}$ fixes the first set in the permutation, that is, it is satisfied when $A_1 \prec A_{\pi(2)} \prec \dots \prec A_{\pi(m)}$ is satisfied for at least one permutation π. Finally, we will abbreviate conjunctions as commas and we will re-use the previously defined abbreviations, such as A^-.

$$
\begin{array}{lll}
{A_0}^+ \Rrightarrow \{{A_1}^+, \dots, {A_n}^+\} & =_{def} & ({A_0}^+ \Rrightarrow {A_1}^+) \wedge \dots \wedge ({A_0}^+ \Rrightarrow {A_n}^+) \\
{A_0}^+ \Rrightarrow \{\} & =_{def} & \mathbf{true} \\
A_1 \prec \dots \prec A_n & =_{def} & \wedge_{1 \leq i < j \leq n} A_i \prec A_j \\
\prec\succ(A_1, \dots, A_m) & =_{def} & (A_1 \prec \dots \prec A_m) \wedge (A_m \prec \dots \prec A_1) \\
A_1 \prec \%\{A_2, \dots, A_m\} & =_{def} & (A_1 \prec (A_2 \cup \dots \cup A_m)) \ \wedge\ \%\{A_2, \dots, A_m\} \\
F_1, \dots, F_n & =_{def} & F_1 \wedge \dots \wedge F_n
\end{array}
$$

We can now generalize constraint extraction. A co-occurrence constraint expresses that, if one symbol from the product occurs, then any factor that is not nullable must contribute a symbol. The constraint is **true** when all factors are nullable. Order and exclusion constraints for concatenation and union are obvious. The % constraint for the % operator is also natural: since the unordered concatenation type is the union of the permuted ordered types, then the constraint is the disjunction of the corresponding permuted order constraints. While flat constraints $\mathcal{FC}(T)$ are not affected, since they only depend on atoms, nested constraints $\mathcal{NC}(T)$ need to be redefined. We still have

$$\mathcal{NC}(T) \ =_{def} \bigwedge_{T_i \text{ subterm of } T} (\, CC(T_i) \wedge OC(T_i)\,)$$

and:

$$\begin{array}{lll} CC(T_1 \otimes \ldots \otimes T_n) & =_{def} & (\cup_{i\in 1..n} S(T_i))^+ \Rightarrow \{\ S^+(T_i)\ \mid\ \text{not N}(T_i)\ \} \\ OC(T_1 \cdot \ldots \cdot T_n) & =_{def} & S(T_1) \prec \ldots \prec S(T_n) \\ OC(T_1 + \ldots + T_n) & =_{def} & \lt\gt(S(T_1), \ldots, S(T_n)) \\ OC(T_1 \% \ldots \% T_n) & =_{def} & \%\{S(T_1), \ldots, S(T_n)\} \end{array}$$

In order to prove soundness and completeness of constraint extraction we need a couple of easy properties. We use $\circledast$ to indicate any n-ary operator, that is, one of $\{\cdot, +, \&, \%\}$.

NOTATION 5.1 ($w_{|A}$). *We indicate with $w_{|A}$ the projection of w on symbols in A, that is, the word obtained by removing from w every symbol that is not in A.*

LEMMA 5.2 (PROJECTION). *For any constraint F not including any formula* upperS(A), *for any A:*

$$B \supseteq S(F) \ \Rightarrow\ (w \models F \Leftrightarrow w_{|B} \models F)$$

LEMMA 5.3 (BINARY). *For any flat types $T_1, \ldots, T_n$:*

$$w \models \mathcal{NC}(T_1 \circledast \ldots \circledast T_n) \quad\Rightarrow\quad (w \models \mathcal{NC}(T_1) \wedge w \models \mathcal{NC}(T_2 \circledast \ldots \circledast T_n))$$

$$w \models \mathcal{NC}(T_1 \circledast \ldots \circledast T_n) \quad\Rightarrow\quad (w_{|S(T_1)} \models \mathcal{NC}(T_1) \wedge w_{|S(T_2 \circledast \ldots \circledast T_n)} \models \mathcal{NC}(T_2 \circledast \ldots \circledast T_n))$$

We can now prove that flat constraints fully characterize flat types, as expressed by the following property.

LEMMA 5.4. *For any flat type T:*

$$w \in [\![T]\!] \ \Leftrightarrow\ w \models \mathcal{FC}(T) \wedge \mathcal{NC}(T)$$

PROOF. Throughout this proof, we will be using the projection Lemma 5.2 without mentioning it, and the following corollary:

a) $w \models \text{ZeroMinMax}(T_1 \circledast \ldots \circledast T_n) \Leftrightarrow w \models \text{ZeroMinMax}(T_1) \wedge \ldots \wedge \text{ZeroMinMax}(T_n)$

We first prove completeness: $w \in [\![T]\!] \ \Leftarrow w \models \mathcal{FC}(T) \wedge \mathcal{NC}(T)$. To this aim, we prove the following property

$$w_{|S(T)} \models \text{ZeroMinMax}(T) \wedge SIf(T) \wedge \mathcal{NC}(T) \ \Rightarrow\ w_{|S(T)} \in [\![T]\!]$$

which implies the desired entailment since by $w \models \text{upperS}(T)$ we have $w = w_{|S(T)}$.

We first consider the special case $w_{|S(T)} = \epsilon$ which easily follows: $\epsilon \models SIf(T)$ implies $N(T)$, hence $\epsilon \in [\![T]\!]$, that is $w_{|S(T)} \in [\![T]\!]$.

We now consider the case when $w_{|S(T)} \neq \epsilon$, reasoning by cases and by induction on the size of the type. We use Lemma 5.3 to reduce the complexity of the proofs, in order to treat

flat types as if they were binary: we will inductively reduce a type $T_1 \circledast \ldots \circledast T_n$ to a type $T_2 \circledast \ldots \circledast T_n$, which is not truly a subterm of the first one, since the operators are n-ary, but has a smaller size, hence can be used for an induction step.

Case $\mathbf{T} = \mathbf{T_1} \cdot \ldots \cdot \mathbf{T_n}$ with $n \geq 2$.

By $w_{|S(T)} \models \text{ZeroMinMax}(T_1 \cdot \ldots \cdot T_n)$ and a), we have $w_{|S(T)} \models \text{ZeroMinMax}(T_1)$ and $w_{|S(T)} \models \text{ZeroMinMax}(T_2 \cdot \ldots \cdot T_n)$. By conflict-freedom of T we have $w_{|S(T_1)} \models \text{ZeroMinMax}(T_1)$ and $w_{|S(T_2 \cdot \ldots \cdot T_n)} \models \text{ZeroMinMax}(T_2 \cdot \ldots \cdot T_n)$.

By $w_{|S(T)} \models \mathcal{NC}(T_1 \cdot \ldots \cdot T_n)$ and Lemma 5.3 we have $w_{|S(T_1)} \models \mathcal{NC}(T_1)$ and $w_{|S(T_2 \cdot \ldots \cdot T_n)} \models \mathcal{NC}(T_2 \cdot \ldots \cdot T_n)$.

By $w_{|S(T)} \models \mathcal{OC}(T_1 \cdot \ldots \cdot T_n)$ we have

$$w_{|S(T)} \models S(T_1) \prec (S(T_2) \cup \ldots \cup S(T_n)) \qquad (i)$$

while by Lemma 5.3 and $w_{|S(T_1 \cdot \ldots \cdot T_n)} \models \mathcal{CC}(T_1 \cdot \ldots \cdot T_n)$ we have

$$w_{|S(T_2 \cdot \ldots \cdot T_n)} \models \mathcal{CC}(T_2 \cdot \ldots \cdot T_n) \qquad (ii)$$

Now, let $w_{|S(T)} = a_1 \ldots a_n$ and distinguish the following two exhaustive, and mutually exclusive cases:

(1) $a_1 \in S(T_1)$
(2) $a_1 \in S(T_2 \cdot \ldots \cdot T_n)$

In case (1), $w_{|S(T_1)} \models \mathit{SIf}(T_1)$ is immediate. We have to prove that $w_{|S(T_2 \cdot \ldots \cdot T_n)} \models \mathit{SIf}(T_2 \cdot \ldots \cdot T_n)$. If $N(T_2 \cdot \ldots \cdot T_n)$, then we are done. If $\neg N(T_2 \cdot \ldots \cdot T_n)$ then there exists $2 \leq i \leq n$ such that $\neg N(T_i)$. By $w_{|S(T)} \models \mathcal{CC}(T)$ and by $a_1 \in S(T)$ we have that $w_{|S(T)} \models S(T_i)^+$, hence $w_{|S(T_i)} \models S(T_i)^+$, hence $w_{|S(T_2 \cdot \ldots \cdot T_n)} \models S(T_2 \cdot \ldots \cdot T_n)^+$, hence $w_{|S(T_2 \cdot \ldots \cdot T_n)} \models \mathit{SIf}(T_2 \cdot \ldots \cdot T_n)$.

Summing up, we have that $w_{|S(T_1)} \models \text{ZeroMinMax}(T_1) \wedge \mathit{SIf}(T_1) \wedge \mathcal{NC}(T_1)$, so by induction we have $w_{|S(T_1)} \in [\![T_1]\!]$. Similarly, we have $w_{|S(T_2 \cdot \ldots \cdot T_n)} \models \text{ZeroMinMax}(T_2 \cdot \ldots \cdot T_n) \wedge \mathit{SIf}(T_2 \cdot \ldots \cdot T_n) \wedge \mathcal{NC}(T_2 \cdot \ldots \cdot T_n)$, hence $w_{|S(T_2 \cdot \ldots \cdot T_n)} \in [\![T_2 \cdot \ldots \cdot T_n]\!]$.

From $w_{|S(T)} \models S(T_1) \prec (S(T_2) \cup \ldots \cup S(T_n))$ we deduce that $w_{|S(T_1 \cdot \ldots \cdot T_n)} = w_{|S(T_1)} \cdot w_{|S(T_2 \cdot \ldots \cdot T_n)}$, hence $w_{|S(T)} \in [\![T_1 \cdot \ldots \cdot T_n]\!]$.

In case (2) we observe that, by $w_{|S(T)} \models S(T_1) \prec (S(T_2) \cup \ldots \cup S(T_n))$, we have that w contains no symbol coming from T_1. This implies $N(T_1)$ (suppose $\neg N(T_1)$, then, by $w_{|S(T)} \models \mathcal{CC}(T_1 \cdot \ldots \cdot T_n)$, we obtain that $w \models S^+(T_1)$, which is absurd). So we have $w_{|S(T_1)} = \epsilon \in [\![T_1]\!]$, therefore we have $[\![T_2 \cdot \ldots \cdot T_n]\!] \subseteq [\![T_1 \cdot \ldots \cdot T_n]\!]$. By reasoning as for case (1) we obtain $w = w_{|S(T_2 \cdot \ldots \cdot T_n)} \in [\![T_2 \cdot \ldots \cdot T_n]\!] \subseteq [\![T_1 \cdot \ldots \cdot T_n]\!]$.

Case $\mathbf{T} = \mathbf{T_1} \% \ldots \% \mathbf{T_n}$ with $n \geq 2$.

By the definition of nested constraint extraction, $\mathcal{NC}(U) =_{def} \wedge_{U' \text{ subterm of } U} (\mathcal{CC}(U') \wedge \mathcal{OC}(U'))$, it follows that the hypothesis $w_{|S(T)} \models \text{ZeroMinMax}(T) \wedge \mathit{SIf}(T) \wedge \mathcal{NC}(T)$ can be rewritten into

$$w_{|S(T)} \models \text{ZeroMinMax}(T) \wedge \mathit{SIf}(T) \wedge \%\{S(T_1), \ldots, S(T_n)\} \wedge \mathcal{CC}(T_1 \% \ldots \% T_n) \wedge \mathcal{NC}(T_1) \wedge \ldots \wedge \mathcal{NC}(T_n)$$

We observe now that for any permutation π on $\{1, \ldots, n\}$, we have (*) $\mathcal{CC}(T_1 \% \ldots \% T_n) = \mathcal{CC}(T_{\pi(1)} \cdot \ldots \cdot T_{\pi(n)})$. This trivially follows from the fact that co-occurrence constraints are order-independent. By definition of $w_{|S(T)} \models \%\{S(T_1), \ldots, S(T_n)\}$, we have that $w_{|S(T)} \models S(T_{\pi(1)}) \prec \ldots \prec S(T_{\pi(n)})$ for a permutation π on $\{1, \ldots, n\}$. For that π, by (*) we have $w_{|S(T)} \models \mathcal{CC}(T_{\pi(1)} \cdot \ldots \cdot T_{\pi(n)})$. So by the initial hypothesis and by the definition of $\mathcal{NC}()$ previously recalled, we have $w_{|S(T)} \models \text{ZeroMinMax}(T) \wedge \mathit{SIf}(T) \wedge \mathcal{NC}(T_{\pi(1)} \cdot \ldots \cdot T_{\pi(n)})$. Now, by reasoning as in the previous case for $T = T_1 \cdot \ldots \cdot T_n$, we obtain that $w_{|S(T)} \in [\![T_{\pi(1)} \cdot \ldots \cdot T_{\pi(n)}]\!]$, which concludes the case.

Case $\mathbf{T} = \mathbf{T_1}\& \ldots \&\mathbf{T_n}$ with $n \geq 2$.

By reasoning as for the $\cdot$ case, we have that:

- $w_{|S(T_1)} \models \text{ZeroMinMax}(T_1)$ and $w_{|S(T_2\&\ldots\&T_n)} \models \text{ZeroMinMax}(T_2\& \ldots \&T_n)$;
- $w_{|S(T_1)} \models \mathcal{CC}(T_1)$ and $w_{|S(T_2\&\ldots\&T_n)} \models \mathcal{CC}(T_2\& \ldots \&T_n)$;
- $w_{|S(T_1)} \models \mathcal{NC}(T_1)$ and $w_{|S(T_2\&\ldots\&T_n)} \models \mathcal{NC}(T_2\& \ldots \&T_n)$.

Now, let $w_{|S(T_1)} = w_1$ and $w_{|S(T_2\&\ldots\&T_n)} = w_2$. As for the $\cdot$ case, we obtain $w_1 \models \mathit{SIf}(T_1)$ and $w_2 \models \mathit{SIf}(T_2\& \ldots \&T_n)$. So the case follows by induction as in the previous case, by observing that $w \in w_1\&w_2$.

Case $\mathbf{T} = \mathbf{T_1} + \ldots + \mathbf{T_n}$ with $n \geq 2$.

By $w_{|S(T)} \models <>(S(T_1), \ldots, S(T_n))$ we have that $w_{|S(T)} = w_{|S(T_i)}$ for an $i \in \{1, \ldots, n\}$. Since $w_{|S(T_i)}$ is not empty, we have that $w_{|S(T_i)} \models \mathit{SIf}(T_i)$. From $w_{|S(T)} \models \mathcal{NC}(T)$ we have $w_{|S(T_i)} \models \mathcal{NC}(T_i)$ by definition, and $w_{|S(T_i)} \models \text{ZeroMinMax}(T_i)$ by a), so by induction we can conclude $w_{|S(T_i)} \in [\![T_i]\!]$, which proves the case.

Case $\mathbf{T} = \mathbf{a}\,[\mathbf{m..n}]$.

The case is easy as $w \neq \epsilon$ and $w \models \mathit{SIf}(T)$ with $\neg N(T)$. So we have that $w_{|S(T)}$ is not empty and only contains the a symbol. By $w_{|S(T)} \models \text{ZeroMinMax}(T)$ we directly have that $w_{|S(T)} \in [\![a\,[m..n]]\!]$.

We now prove soundness:

$$w \in [\![T]\!] \Rightarrow w \models \mathcal{FC}(T) \wedge \mathcal{NC}(T)$$

We first prove $w \in [\![T]\!] \Rightarrow w \models \mathcal{FC}(T)$. Consider first the $\mathit{SIf}(T)$ constraint: if $w = \epsilon$, then $\text{N}(T)$, so $\mathit{SIf}(T)$ is **true**. Otherwise w contains at least one symbol in $S(T)$, hence $w \models S(T)^+$. The other flat constraints are immediate by induction.

To prove $w \in [\![T]\!] \Rightarrow w \models \mathcal{NC}(T)$ we proceed by structural induction and case distinction on T. The case $\mathbf{T} = \mathbf{a}\,[\mathbf{m..n}]$ is trivial. We also assume that $w \neq \epsilon$ as ϵ trivially satisfies any order and co-occurrence constraint.

$\mathbf{T} = \mathbf{T_1} \cdot \ldots \cdot \mathbf{T_n}$.

By hypothesis and conflict-freedom we have (*) $w = w_1 \cdot \ldots \cdot w_n$ with $w_i = w_{|S(T_i)}$ and $w_i \in [\![T_i]\!]$, with $i \in \{1, \ldots, n\}$. So by induction we can assume $w_i \models \mathcal{NC}(T_i)$. So we must only prove that $w \models \mathcal{OC}(T)$ and $w \models \mathcal{CC}(T)$. $w \models S(T_1) \prec \ldots \prec S(T_n)$ directly follows by (*) and conflict-freedom of T (sets $S(T_i)$ are pairwise disjoint). Concerning $w \models (\cup_{i\in 1..n} S(T_i))^+ \Mapsto \{\ S^+(T_i) \mid \text{not N}(T_i)\ \}$, for any $i \in \{1, \ldots, n\}$ such that $\text{N}(T)$ does not hold, by $w_i \in [\![T_i]\!]$ we have $w_i \models S(T_i)^+$. So the case is proved.

$\mathbf{T} = \mathbf{T_1}\% \ldots \%\mathbf{T_n}$ with $n > 2$.

By hypothesis we have that $w \in [\![T_{\pi(1)} \cdot \ldots \cdot T_{\pi(n)}]\!]$ for a permutation π of $\{1, \ldots, n\}$. So the case follows by proceeding as above, and by the definition of the semantics of % constraints.

$\mathbf{T} = \mathbf{T_1} + \ldots + \mathbf{T_n}$.

W.l.o.g. we can assume $w \in [\![T_j]\!]$. By conflict-freedom of T we have that $S(w) \cap S(T_i) = \emptyset$ for $i \neq j$. This implies $w \models <>(S(T_1), \ldots, S(T_n))$. Moreover, by induction we can assume $w \models \mathcal{NC}(T_j)$ as $w \in [\![T_j]\!]$. We also have $w \models \mathcal{NC}(T_i)$ for $i \neq j$, since $w \models \mathcal{NC}(U)$ always holds when $S(w) \cap S(U) = \emptyset$ (this property can be easily proved by induction).

$\mathbf{T} = \mathbf{T_1}\& \ldots \&\mathbf{T_n}$. By hypothesis and conflict-freedom we have $w \in w_1\& \ldots \&w_n$ with $w_i = w_{|S(T_i)}$ and $w_i \in [\![T_i]\!]$, with $i \in \{1, \ldots, n\}$. So by induction we can assume $w_i \models \mathcal{NC}(T_i)$. The

Table 4. Residuals of constraints for flat types.

Condition	**Constraint**	**Residual after** a
$a \in A \wedge a \in A_i$	$A^+ \Rrightarrow \{{A_1}^+, \ldots, {A_n}^+\}$	${A_1}^+, \ldots, {A_{i-1}}^+, {A_{i+1}}^+, \ldots, {A_n}^+$
$a \in A \wedge \forall i. a \notin A_i$	$A^+ \Rrightarrow \{{A_1}^+, \ldots, {A_n}^+\}$	${A_1}^+, \ldots, {A_n}^+$
$a \in A_i$	${A_1}^+, \ldots, {A_n}^+$	${A_1}^+, \ldots, {A_{i-1}}^+, {A_{i+1}}^+, \ldots, {A_n}^+$
$a \in A_j$	$<>(A_1, \ldots, A_n)$	${A_1}^-, \ldots, {A_{j-1}}^-, {A_{j+1}}^-, \ldots, {A_n}^-$
$a \in A_j$	${A_1}^-, \ldots, {A_n}^-$	**false**
$a \in A_j, j > i$	${A_1}^-, \ldots, {A_{i-1}}^-, A_i \prec \ldots \prec A_n$	${A_1}^-, \ldots, {A_{j-1}}^-, A_j \prec \ldots \prec A_n$
$a \in A_j, j < i$	${A_1}^-, \ldots, {A_{i-1}}^-, A_i \prec \ldots \prec A_n$	**false**
$a \in A_j$	$\%_{k \in \{1\ldots n\}}\{A_k\}$	$A_j \prec \%_{k \in \{1\ldots n\} \setminus \{j\}}\{A_k\}$
$a \in A_j$, $j \notin (S \cup \{i\})$	$\wedge_{k \in S} {A_k}^-, A_i \prec \%_{k \notin (S \cup \{i\})}\{A_k\}$	$\wedge_{k \in (S \cup \{i\})} {A_k}^-$, $A_j \prec \%_{k \notin (S \cup \{i,j\})}\{A_k\}$
$a \in A_j, j \in S$	$\wedge_{k \in S} {A_k}^-, A_i \prec \%_{k \notin (S \cup \{i\})}\{A_k\}$	**false**

case is proved by reasoning as in the $\cdot$ case, with the exception that only $w \models (\cup_{i \in 1..n} S(T_i))^+ \Rrightarrow \{\ S^+(T_i)\ \mid\ \text{not } \mathrm{N}(T_i)\ \}$ needs to be proved. □

Constraints for flat types are residuated as follows (Table 4). A co-occurrence constraint ${A_0}^+ \Rrightarrow \{{A_1}^+, \ldots, {A_n}^+\}$ reduces to ${A_1}^+, \ldots, {A_n}^+$ when a symbol from A_0 is read which is not in any A_i: by definition, if a symbol in A_0 is found, any implied set must be satisfied as well. This happens when the node is reached through a child that is nullable. If the symbol is both in A_0 and in A_j, then the residual is ${A_1}^+, \ldots, {A_{j-1}}^+, {A_{j+1}}^+, \ldots, {A_n}^+$, since ${A_j}^+$ has been satisfied already. This is the case when the symbol is reached from a non-nullable child. A constraint ${A_1}^+, \ldots, {A_n}^+$ is residuated by deleting a set A_i whenever a symbol from that set is read.

Mutual exclusion constraint $<>(A_1, \ldots, A_n)$ becomes ${A_1}^-, \ldots, {A_{i-1}}^-, {A_{i+1}}^-, \ldots, {A_n}^-$ after a symbol in A_i is read.

Order constraint $A_1 \prec \ldots \prec A_n$ specifies that, when a symbol from A_i is read, with $i > 1$, no symbol from $A_1 \ldots A_{i-1}$ can be met anymore, and the order for $A_i \ldots A_n$ must be respected, hence it generates a conjunctive constraint ${A_1}^-, \ldots, {A_{i-1}}^-, A_i \prec \ldots \prec A_n$. This residuated constraint is in turn residuated in a similar way when a symbol from A_j with $j > i$ is read: more sets are moved in the forbidden prefix, and j becomes the first of the still allowed sets. When a symbol from $A_1 \ldots A_{i-1}$ is read, the constraint is violated. When a symbol from A_i is read, nothing happens. In Table 4 we only give the rules for the exclusion plus order constraint, since the pure order constraint $A_1 \prec \ldots \prec A_n$ is just the special case when the prefix set is empty.

The residuation of % constraints is tightly related to that of order constraints. When a symbol from A_i is read, $\%\{A_1, \ldots, A_n\}$ is residuated to $A_i \prec \%\{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n\}$. If we keep reading symbols from A_i, the constraint is left unchanged; however, when we first meet a symbol from A_j, $j \neq i$, then we obtain a constraint of the form ${A_i}^-, A_j \prec \%_{k \in \{1,\ldots,n\}, k \neq i, k \neq j}\{A_k\}$. This constraint explicitly forbids any other symbol from A_i, so that, if we read any of them, the residuation returns **false**; symbols from A_j are, instead, irrelevant, while symbols from A_k ($k \neq i$, $k \neq j$) move A_j to the forbidden part of the constraint.

As with binary constraints, residuation of n-ary constraints yields an algorithm to verify whether a word satisfies a constraint.

Lemma 5.5 (n-ary Residuation). *For any n-ary constraint F: $w \models F$ iff $F \downarrow^w$* **true**.

PROOF. We first observe that $aw \models F$ iff $F \xrightarrow{a} F' \wedge w \models F'$, reasoning by cases on F and a, along the lines discussed in the text. Then we conclude that $w \models F$ iff $(F \xrightarrow{w}^{*} F' \wedge \epsilon \models F')$ by induction on the length of w.

□

Observe that, by definition, the only constraints that are not satisfied by an empty word are conjunctions that include some A^+ (we do not need to explicitly consider **false**, since **false** is defined as $\emptyset^+$).

LEMMA 5.6.

$$F_1, \ldots, F_n \downarrow^{\epsilon} \textbf{ false} \quad \Leftrightarrow \quad \exists i \in \{1, \ldots, n\}\ \exists A \quad F_i = A^+$$
$$F_1, \ldots, F_n \downarrow^{\epsilon} \textbf{ true} \quad \Leftrightarrow \quad \forall i \in \{1, \ldots, n\}\ \forall A \quad F_i \neq A^+$$

5.2 The flat algorithm

In order to devise a membership algorithm based on n-ary constraints, we refine the data structures of the binary residuation algorithm, as follows. Co-occurrence constraints are represented by an array $CC[]$ of records with the following fields: $CC[n].kind \in \{\Rrightarrow, A^+, \textbf{true}\}$, $CC[n].needed[]$ is an array of boolean values $\{\textbf{true}, \textbf{false}\}$, one for each child of n, and $CC[n].neededCount \in \mathbb{N}$. Informally, if we have a type $T_1 \otimes T_2 \otimes T_3$, where T_1 is the only nullable child, we have a constraint $S^+(T_1 \otimes T_2 \otimes T_3) \Rrightarrow (S^+(T_2), S^+(T_3))$, and we represent it through a record cc[3] with "$cc.kind$=($\Rrightarrow$)", with $cc.needed$[]= [$false, true, true$], specifying that $S^+(T_1)$ is "not needed" (since T_1 is nullable), while $S^+(T_2)$ and $S^+(T_3)$ are "needed"; we also have $cc.neededCount = 2$.

The first time we meet any child i of a type $T_1 \otimes \ldots \otimes T_i \otimes \ldots \otimes T_n$, we switch the kind of cc from ($\Rrightarrow$) to (A^+), and we set $cc.needed[i]$ to *false*. For any other child i' we meet, we will also set its $cc.needed[i']$ value to *false*. Every time we switch a *needed*[] entry from *true* to *false*, we also decrease the value of *cc.neededCount*, so that any constraint of kind (A^+) is satisfied when its *cc.neededCount* is down to zero.

Order constraints for a node n with m children are represented by the record $OC[n]$ with fields $kind \in \{^{-}\prec, \lt\gt, \%, ^{-}\%, A^{-}, \textbf{true}\}$, $current \in \{1 \ldots m\}$, and *forbidden*, which is an array of m booleans, one for each child; the boolean array is only used for residuated % constraints, and initially only contains **false**.

If we have a type $T_1 \cdot \ldots \cdot T_m$, the corresponding record oc[4] has $oc.kind$=($^{-}\prec$) and $oc.current$=1, which corresponds to a constraint $S(T_1) \prec \ldots \prec S(T_m)$. More generally, $oc.kind$=($^{-}\prec$) with $oc.current$=i, represents a constraint $A_1{}^{-}, \ldots, A_{i-1}{}^{-}, A_i \prec \ldots \prec A_m$, hence, when we meet a symbol in $S(T_j)$, if $j < oc.current$ we return **false**, and otherwise we just update *oc.current* to j.

A type $T_1 + \cdots + T_m$ is represented by a record oc with $oc.kind$=($\lt\gt$), which is residuated into $oc.kind$=(A^{-}) and $oc.current$=i when a symbol in $S(T_i)$ is met, and yields **false** if, later on, a symbol in $S(T_{i'})$ is met with $i' \neq i$.

A type $T_1\% \ldots \%T_m$ is represented by a record oc with $oc.kind$=(%), which is residuated into $oc.kind$=($^{-}\%$), $oc.current$= i when a symbol in $S(T_i)$ is met. If in a subsequent step a T_j symbol is met, with $i \neq j$, then we set $oc.current$= j and *oc.forbidden[i]* takes **true**. In general, if a T_h symbol is met: if $oc.current$= h nothing happens, if *oc.forbidden[h]*= **true** then **false** is returned, otherwise *oc.forbidden[oc.current]* is set to **true** and *oc.current* is set to h.

[3] cc denotes a generic record of $CC[]$, e.g., $CC[j]$.
[4] oc denotes a generic record of $OC[]$, e.g., $OC[j]$.

To sum up, we represent constraints, and their residuals, through the following two arrays, where we assume that n is the node that corresponds to either a type $T = T_1 \otimes \ldots \otimes T_m$ or a type $T = T_1 + \ldots + T_m$.

- CC[]: CC[n] is a record with fields *kind*, *needed*[], and *neededCount*. The meaning of CC[] depends on the value of $CC[n].kind$, as follows:
  - $\Rrightarrow$: $CC[n]$ represents a constraint

$$(\cup_{i\in 1..m} S(T_i))^+ \Rrightarrow \{S(T_{k(1)})^+, \ldots, S(T_{k(j)})^+\}$$

    where j is $CC[n].neededCount$, and where $\{k(1), \ldots, k(j)\}$ enumerates the indexes i such that $CC[n].needed[i]{=}true$.
  - A^+: $CC[n]$ represents a constraint

$$S(T_{k(1)})^+, \ldots, S(T_{k(j)})^+$$

    where j is $CC[n].neededCount$, and where $\{k(1), \ldots, k(j)\}$ enumerates the indexes i such that $CC[n].needed[i]{=}true$.
- OC[]: OC[n] is a record with fields *kind*, *current*, and, if n is a % type, *forbidden*[]. The meaning of OC[] depends on the *kind* field, as follows (j is assumed to range in $1 \ldots m$ with $m \geq 1$):
  - $^-\!\prec$: this is a constraint

$$A_1{}^-, \ldots, A_{i-1}{}^-, A_i \prec \ldots \prec A_m$$

    where $A_j = S(T_j)$, and $i = CC[n].current$. When $i = 1$, this constraint is just $A_1 \prec \ldots \prec A_m$.
  - $\prec\succ$: this is a constraint $\prec\succ(A_1, \ldots, A_m)$, where $A_j = S(T_j)$;
  - A^-: this is a residual constraint

$$A_1{}^-, \ldots, A_{i-1}{}^-, A_{i+1}{}^-, \ldots, A_m{}^-$$

    where $A_j = S(T_j)$ and $i = OC[n].current$;
  - %: this is a constraint $\%\{A_1, \ldots, A_m\}$ where $A_j = S(T_j)$;
  - $^-\%$: this is a residual constraint

$$\wedge_{k\in S} A_k{}^-, A_i \prec \%_{k\notin(S\cup\{i\})}\{A_k\}$$

    where $A_j = S(T_j)$, $i = OC[n].current$, and $S = \{i \mid OC[n].forbidden[i] = \mathbf{true}\}$.

We present the MemberFlat algorithm in Figure 7. For simplicity, we omit parts similar to those of previously presented algorithms. In this algorithm *Ancestors(n)* returns a list of pairs *(father, pos)*, where *father* is the father node of n and *pos* is the relative position of n in the list of children of *father*; this function is an n-ary generalization of the *Ancestors(n)* function used in the binary algorithm.

The next example illustrates its behaviour.

Example 5.7. Consider the type $T = (a\%(b\&c))\cdot(d^*)\cdot(e + f + g)$ (Figure 6), where we use x to abbreviate $x\,[1..1]$ and x^* to abbreviate $x\,[1..*] + \epsilon$.

The record $OC[3]$ is null since only one child of node 3 has a non-empty set of symbols. Assume we read a word *cbadddgdga*. When c is read, nodes 6, 2, and 1 are visited (line 7), and their corresponding constraints are residuated to: $CC[6]{=}(A^+,[t,f],1)$ (lines 9 - 12), $OC[2]{=}(^-\%,2,[f,f])$ (lines 26 - 28), $CC[2]{=}(A^+,[t,f],1)$ (lines 9 - 12), $CC[1]{=}(A^+,[f,f,t],1)$ (lines 9 - 12), $OC[1]{=}(^-\prec,1)$ (lines 17 - 20). The constraint $OC[1]$ is actually unaffected, because both *current* and the child position *pos* are equal to 1 (lines 18 - 19). Nodes 6, 2 and 1 are inserted in *ToCheck*, since they now have an A^+ kind (line 11). When b is read, $CC[6]$

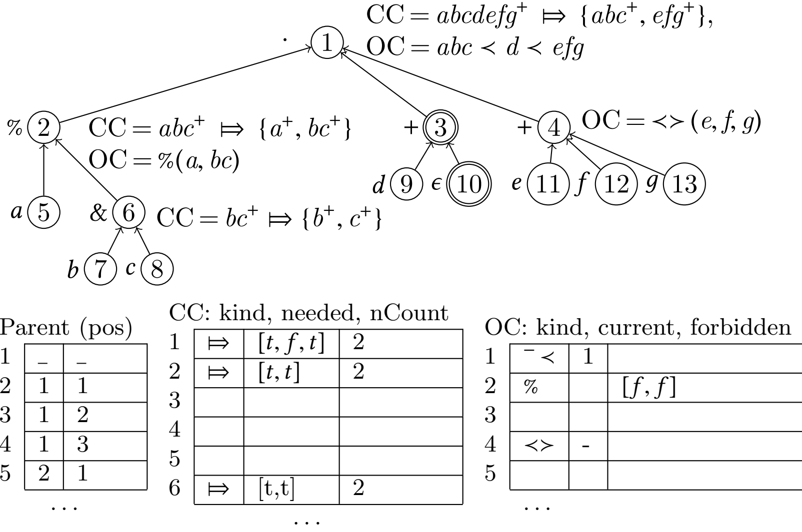


Fig. 6. Representation of $T = (a\%(b\&c))\cdot(d^*)\cdot(e+f+g)$.

becomes (**true**,$[f,f]$,0) (line 14), while other constraints in the path up to the root remain unchanged (as we will see in the next section, due to stability). When a is read, $CC[2]$ becomes (**true**,$[f,f]$,0) (line 14), since its *neededCount* is 0, and $OC[2]$=($^-\%$,1,$[f,t]$) (lines 29 - 34). When d is read, 3 and 1 are visited; $CC[1]$ is unaffected, since $CC[1]$.*needed*[2] is already *false*, while $OC[1]$ becomes ($^-\prec$,2) (lines 17 - 19), which means that symbols from the first subtree are now forbidden. The next two d's require two new visits to 3 and 1, which have no effect. When g is read, $OC[4]$ becomes (A^-,3) (lines 21 - 23), $CC[1]$ becomes (**true**, $[f,f,f]$,0) (line 14), and $OC[1]$ becomes ($^-\prec$,3) (lines 17 - 19). If the word ended here, the algorithm would return *true*, since all nodes in *ToCheck* have now kind **true**. But now a new d is read, which makes the algorithm stop with *false* (lines 17 - 20), because *pos* (= 2) < $OC[1]$.*current* (= 3).

THEOREM 5.8 (SOUNDNESS). MemberFlat*(w,T) yields* true *iff* $w \in [\![T]\!]$.

PROOF. We first prove the left-to-right implication. As for MEMBER, each time a symbol a in w is met, all the nested T constraints including a are residuated according to flat constraints residuation rules (Table 4). If the main loop consumes all the symbols of w, then we can conclude that all ordered constraints have been residuated to **true**. Then the check of absence of nodes with C[n].kind ≠ **true** ensures that also co-occurrence constraints have been residuated to **true**. So by Lemma 5.5 we can conclude that w meets all T nested constraints. Concerning flat constraints the reasoning is similar to that for the MEMBER algorithm (Theorem 3.5). So we can conclude that if MEMBERFLAT(w,T) yields *true* then all T constraints are met by w, and therefore $w \in [\![T]\!]$.

Now the right-to-left implication. If $w \in [\![T]\!]$, then w satisfies all T constraints. By Lemma 5.5 we have that any nested T constraint residuates to **true**, hence the algorithm successfully parses all w symbols and successfully passes the final check for co-occurrence constraints (**if** (**exists** n **in** ToCheck **with** (CC[n].kind ≠ **true**))). Also, the algorithm successfully checks other flat constraints. Hence we conclude that MEMBERFLAT(w,T) yields *true*.

□

```
MemberFlat(w, T)
 1  (Min[],Max[],NodeOfSymbol[],Parent[],CC[],OC[],Nullable) := EncodeType(T);
 2  SetToZero(Count[]);
 3  if (IsEmpty(w) and not Nullable) return (false);
 4  for a in w
 5      if (NodeOfSymbol[a] is null) return (false);
 6      Count[a] := Count[a]+1;
 7      for (n,pos) in Ancestors(NodeOfSymbol[a])
 8          case CC[n].kind
 9              when (⤇)
10                  then  CC[n].kind := A⁺;
11                        push(n,ToCheck);
12                        ResiduatePlus(CC[n],pos);
13              when (A⁺)
14                  then  ResiduatePlus(CC[n],pos);
15              else ;
16          case OC[n].kind
17              when (⁻<)
18                  then  if (pos >= OC[n].current)
19                            OC[n].current := pos;
20                        else return (false);
21              when (<>)
22                  then  OC[n].kind := A⁻;
23                        OC[n].current := pos;
24              when (A⁻)
25                  then  if (pos ≠ OC[n].current) return (false);
26              when (%)
27                  then  OC[n].kind := ⁻%;
28                        OC[n].current := pos;
29              when (⁻%)
30                  then  if (OC[n].forbidden[pos])
31                            return (false);
32                        elseif (pos≠OC[n].current)
33                            OC[n].forbidden[OC[n].current]:=true;
34                            OC[n].current:=pos;
35              else ;
36  if (exists n in ToCheck with (CC[n].kind ≠ true)) return (false);
37  if (not CardinalityOK(Count[],Min[],Max[])) return (false);
38  return (true);

ResiduatePlus(ccn, childPos)
1  if (ccn.needed[childPos])
2      ccn.needed[childPos] := false;
3      ccn.neededCount := ccn.neededCount-1;
4      if  (ccn.neededCount =0) ccn.kind := true;
```

Fig. 7. Flat membership algorithm.

This version of the algorithm has $O(|T| + |w| * \mathit{flatdepth}(T))$ time complexity, where $\mathit{flatdepth}(T)$ is the depth of the type after all operators have been flattened. In practice, $\mathit{flatdepth}(T)$ is almost invariably smaller than three, even in the biggest XML types, since

types usually get big because of a very wide choice of alternatives or a very long concatenation of symbols, while a deep alternation of different operators is extremely unusual (see [5]). Hence, this algorithm is in practice "almost linear" even before stability is considered.

THEOREM 5.9 (COMPLEXITY). MemberFlat*(w,T) runs in time* $O(|T| + |w| * \mathit{flatdepth}(T))$.

PROOF. The algorithm comprises two nested **for** loops and a final scan on *ToCheck*. The outer loop is executed once for each symbol in the word w; for each symbol a in w, the inner loop explores the path from the node containing a to the root of the parse tree, and executes a fixed number of unit time operations. As a consequence, the two nested loops require $O(|w| * \mathit{flatdepth}(T))$ operations.

The control on *ToCheck* requires to scan at most $|T|$ nodes, while *CardinalityOK* can be evaluated in $O(|T|)$ time. The encoding of T can be built in $O(|T|)$. As a consequence, the *MemberFlat* algorithm has $O(|T| + |w| * \mathit{flatdepth}(T))$ time complexity.

□

5.3 Exploiting stability

The algorithm MEMBERFLAT can be further optimized by exploiting stability, which still holds for n-ary constraints. The resulting algorithm is based on the following stability lemma, and is presented in Figure 8.

LEMMA 5.10 (N-ARY STABILITY). *Let F be one of the following constraints*

$$A^+ \Mapsto \{{A_1}^+, \ldots, {A_n}^+\}$$
$$<>(A_1, \ldots, A_n)$$

with $A \supseteq (A_1 \cup \ldots \cup A_n)$, and A_k and A_j pairwise disjoint when $k \neq j$.

For each symbol a, words w and w', and i such that $1 \leq i \leq n$, the following implication holds:

$$F \xrightarrow{waw'}{}^{*} F' \wedge a \in A_i \quad \Rightarrow \quad \forall a' \in A_i.\ F' \xrightarrow{a'} F'$$

PROOF. By cases on the shape of F.

Assume F has the form $A^+ \Mapsto \{{A_1}^+, \ldots, {A_n}^+\}$. To prove A_i-stability, with $1 \leq i \leq n$, we first observe that, if $a \in A_i$, then $F \xrightarrow{waw'} F'$ (recall that $A_i \subseteq A$) implies that either $F' = {A_{i_1}}^+, \ldots, {A_{i_k}}^+$, with $\{A_{i_1}, \ldots, A_{i_k}\} \subseteq \{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n\}$, or $F' = \textbf{true}$. So, A_i-stability follows in the first case since $A_{i_h} \cap A_i = \emptyset$, while is trivial in the second case.

Assume F has the form $<>(A_1, \ldots, A_n)$. Then $F \xrightarrow{waw'} F'$ implies that either $F' = {A_1}^-, \ldots, {A_{i-1}}^-$, ${A_{i+1}}^-, \ldots, {A_n}^-$ or $F' = \textbf{false}$. So, A_i-stability follows by definition of residuation in the first case since $A_i \cap A_k = \emptyset$ with $i \neq k$, while is trivial in the other case.

□

The algorithm MEMBERFLATSTAB extends MEMBERFLAT with stability and is listed in Figure 8. Every time the visit moves from a node c to its parent p, the pointer Parent[c] is deleted. In this way no node is reached twice from the same child, which, by stability, would be useless, apart from the non-stable constraints. However, stability does not hold for the ${A_1}^-, \ldots, {A_{i-1}}^-, A_i \prec \ldots \prec A_m$ and ${A_{j_1}}^-, \ldots, {A_{j_i}}^-, A_{j_{i+1}} \prec \%\{A_{j_{i+2}}, \ldots, A_{j_n}\}$ constraints. For these constraints the link from the children that correspond to the forbidden part are reactivated every time the current focus is moved. As in the binary case, the reactivation of the up-links must be recursively executed on all the descendants. To allow this reactivation, every time the link from c to p is deactivated while moving from c to p, the pair (c, pos), where pos is the position of c among the children of p, is stored in ToRestore[p]. Hence, when

p is required to restore its up-links, it can broadcast the same request to all the children that deleted their up-links.

```
MemberFlatStab(w, T)
     ...;
 1   if (IsEmpty(w) and not Nullable) return (false);
 2   for a in w
 3       if (NodeOfSymbol[a] is null) return (false);
 4       Count[a] := Count[a]+1;
 5       n:=(NodeOfSymbol[a]);
 6       while (Parent[n] is not null)
 7           child := n;
 8           (n,pos) := Parent[n];
 9           Parent[child] := null;
10           case CC[n].kind
             ...;
11           case OC[n].kind
12                when (⁻<)
13                     then  if (pos < OC[n].current)
14                                return (false)
15                           elseif (pos > OC[n].current)
16                                restoreUpLinks(n);
17                                OC[n].current := pos;
18                when (<>) then ...;
19                when (A⁻) then ..;
20                when (%) then...;
21                when (⁻%)
22                     then  if (OC[n].forbidden[pos])
23                                return (false);
24                           elseif (pos≠OC[n].current)
25                                restoreUpLinks(n);
26                                OC[n].forbidden[OC[n].current]:=true;
27                                OC[n].current:=pos;
28                else ;
29           ToRestore[n] := append((child,pos),ToRestore[n]);
30       endwhile
         ..;
```

```
RestoreUpLinks(n)
1   for (nc,pos) in ToRestore[n]
2        Parent[nc] := (n,pos);
3        RestoreUpLinks(nc)
4   ToRestore[n]:=();
```

Fig. 8. Flat membership algorithm with stability.

Proposition 5.11 (Linear complexity). MemberFlatStab*(w,T) runs in time* $O(|T| + |w|)$.

Proof. We reason as in Proposition 4.3. The algorithm has a set-up phase of cost $O(|T|)$. Then, for every symbol a of the word, we increment the counter of a and we follow the

not-yet-deleted uplinks inside T, performing constant-time operations for each link, with the only exception of link restoration, that has constant cost for each restored link. As in Proposition 4.3, every link is traversed or restored at most three times, hence the total cost of link traversal is $O(|T|)$. The final steps are in $O(\min(|T|, |w|))$ as in Proposition 4.3. □

6 MEMBERSHIP FOR XSD SCHEMAS

6.1 Multi-Words Checking

A single type T is often used to check m words $w_1, \ldots, w_m$ - one-time use of a type is not the standard case. In this case, the repeated application of MemberStab gives us an upper bound of $O(m * (|T| + |w|))$, where $|w|$ is the average length of the words. This bound is not linear, in general, in the input size $|T| + (m * |w|)$, due to the $m * |T|$ component. If types are always smaller than words, this is not a problem, since $m * (|T| + |w|)$ is in this case bounded by $m * (|w|)$, hence the algorithm is indeed linear. In the general case, where $|T|$ may be bigger than $|w|$, the non-linearity of the theoretical bound reflects the practical observation that we should not needlessly rebuild a type, which may be big, for every word that we read.

This problem can be easily solved by building the CC[] and OC[] structures once, plus two copies *CCSave*[] and *OCSave*[]. Every time a word is checked, rather than rebuilding the type from scratch, we use *ToRestore*[] in order to only rebuild the part of the type that has been modified, by applying the following *RestoreType* procedure, which trivially extends *RestoreUpLinks*, to the root of the type. The same technique can be of course used with the flat algorithm MemberFlatStab.

```
RestoreType(n)
1  OC[n] := OCSave[n]
2  CC[n] := CCSave[n]
3  for (nc,pos) in ToRestore[n]
4       Parent[nc] := (n,pos);
5       if Symbol[nc] is not null
6            Count[Symbol[nc]]:=0
7       else RestoreType(nc)
8  ToRestore[n]:=();
```

This *restoring* phase does not visit the whole T but only the modified part, which is never bigger than $|w| * depth(T)$, which gives the whole algorithm a complexity $O(|T| + m * |w| * depth(T))$. When types are flattened, $flatdepth(T)$ is typically smaller than 3, hence this algorithm is 'quasi-linear'.

In the next section, we will use *MultiMemberStab* as a name for this optimized algorithm.

6.2 XSD Schemas

We are now ready to extend our techniques from words to trees. For the purpose of this discussion, we focus on XML trees where every node is an element node, hence on documents generated by the following grammar:

$$x ::= \epsilon \quad | \quad \langle a \rangle x \langle /a \rangle x$$

Following a long tradition, (see [24], for example), we model an XSD schema as an extended DTD, that is, as a quintuple $(\Sigma, \Delta, \tau, \mu, \rho)$, where Σ is a set of labels, Δ is a set of *type-names*, τ is a function mapping each type-name to a content-model, which is a type expressed on the alphabet Δ, μ is a function from Δ to Σ, and $\rho \in \Delta$ is the root type-name. Although μ is not injective in general, the *Element Declarations Consistent* (EDC) constraint specifies that μ must be injective when restricted to a specific content model (see [43]).

As a consequence, it is possible to check membership of an XML tree x into an XSD schema as follows. Membership checking happens in the context of a specific type-name β, which is initially the root type-name of the schema, hence of a specific content-model $T = \tau(\beta)$. To check whether $\langle a_1 \rangle x_1 \langle /a_1 \rangle \ldots \langle a_n \rangle x_n \langle /a_n \rangle$ satisfies T, we retrieve the content model $T_i = \tau(\mu_\beta^{-1}(a_i))$ of each subelement, check that each x_i matches T_i, and check that the sequence $w = \mu_\beta^{-1}(a_1) \ldots \mu_\beta^{-1}(a_n)$ matches T. Here, $\mu_\beta^{-1}(_)$ is the inverse of μ restricted to the type-names appearing in the content model of β; this inverse function is well-defined thanks to the EDC constraint.

We assume here that each content model is expressed in our type language and satisfies the conflict-freedom constraint. The cost of verifying whether x satisfies (τ, μ, ρ) depends on the cost of checking whether a word belongs to a content model $\tau(\alpha)$, as follows. We assume that the XSD schema contains $|J|$ content models $\{\tau(\alpha_j)\}^{j \in J}$, each of size $|\tau(\alpha_j)|$, that x contains (immediately or recursively) $|I|$ elements $\{e_i\}^{i \in I}$, and that w_i is the sequence of the labels of the children of e_i. We assume that *MultiMemberStab* is used for word-membership. We have a set-up phase with cost $O(\sum_{j \in J} |\tau(\alpha_j)|) = O(|\tau|)$. We have a checking and type-restoration phase with cost $O(|w_i| * depth(\tau(\alpha_i)))$ for each $w_i \in [\![\tau(\alpha_i)]\!]$ test.[5] Hence, the total cost of XSD checking is in $O(|T| + |w_i| * depth(T))$. Since DTDs can be modeled as a special case of EDTDs, this result holds for DTDs as well.

7 EXPERIMENTAL EVALUATION

In the previous sections we presented two different families of membership checking algorithms for regular expressions in cf-RE(#, &) and cf-RE(#, &, %), respectively. In this section we analyze the performance and scalability of these algorithms, compared with a baseline algorithm based on Brzozowski's derivatives [14] for expressions in cf-RE(#, &) and cf-RE(#, &, %). We would have liked to also compare our algorithms with some of the many automata-based systems, but, to the best of our knowledge, while many tools can deal with regular expressions with counting, there are no tools supporting both counting and interleaving, with the notable exception of Anders Møller's automaton library [37]. However, this library maps regular expressions with interleaving and counting into DFAs incurring in an exponential blow up of the size of automaton, hence it cannot deal with the size of our test suite. Finally, we were not able to find any tool or automaton library supporting unordered concatenation.

7.1 Derivative-based Membership Algorithm

As said above, in our experiments we compare the proposed algorithms with a competitor based on Brzozowski's derivatives. There are several reasons behind this choice. First of all, to the best of our knowledge, there are no tools for regular expressions offering unordered concatenation. Furthermore, while counting is supported by many tools, the only known implementation of interleaving is represented by Anders Møller's automaton library; this library, while very efficient for standard regular-expression operators, features a trivial implementation of counting, which leads to an exponential explosion of automaton size and is hence unable to support regular expressions with multiple counting operators and large upper bounds; interleaving is implemented through a product automaton construction (see [36]), and also leads to an exponential explosion of the number of states.

[5] Although XSD-checking uses top-down recursion, its total run-time can be still evaluated by just adding the time needed to verify that the w_i label sequence of each element, at any depth level in the document, matches the element content model

Hence, lacking an established alternative, we decided to explore a competitor algorithm based on Brzozowski's derivatives, which are very well behaved for deterministic types, so that the technique is well suited to work with conflict-free types. The algorithm is an original adaptation to our operators of a well known approach. The notion of Brzozowski's derivative is standard, and is defined as follows (see [13]).

Definition 7.1. The *derivative* of a regular language L over a finite alphabet Σ with respect to a symbol $a \in \Sigma$ is defined as $\delta_a(L) = \{v \mid a \cdot v \in L\}$. Such a derivative is, by definition, unique.

A derivative of a regular expression E whose language is L, with respect to a, is any regular expression E' that generates the language $\delta_a(L)$. The Brzozowski's derivative $d_a(E)$ of an expression E with respect to a is a specific derivative expression of E with respect to a that is computed according to the rules described by Brzozowski in [14].

To define a derivative-based membership algorithm for conflict-free types, we extended Brzozowski's derivation rules to conflict-free types with interleaving, counting, and unordered concatenation (see also [42]). To this aim, we first extend our type language with the *empty* expression.

Definition 7.2. $\emptyset$ denotes the *empty* regular expression, that is, $[\![\emptyset]\!] =_{def} \emptyset$. $\emptyset$ satisfies the following properties:

$$T + \emptyset \;=\; \emptyset + T \;=\; T \qquad T \cdot \emptyset \;=\; \emptyset \cdot T \;=\; \emptyset \qquad T \& \emptyset \;=\; \emptyset \& T \;=\; \emptyset$$

Moreover, we relax the constraints of Section 2.1 on the bounds of $a\,[m..n]$ types: we allow 0 to appear in both the m and n positions of $a\,[m..n]$, with the obvious semantics (specifically, $[\![a\,[0..0]]\!] = [\![\epsilon]\!]$ and $[\![a\,[0..n]]\!] = [\![\epsilon + a\,[1..n]]\!]$).

Hereafter, we will use cf-RE$(\#, \&, 0)$ to denote the class of conflict-free expressions extended with the $\emptyset$ type and 0 bounds. We extend cf-RE$(\#, \&, \%)$ to cf-RE$(\#, \&, \%, 0)$ in a similar way. We use these wider classes since they greatly simplify the definition of derivatives, and a membership algorithm for these classes can of course be used for cf-RE$(\#, \&)$ and cf-RE$(\#, \&, \%)$.

We can now give a function that returns a derivative for a conflict-free expression in cf-RE$(\#, \&, 0)$ and in cf-RE$(\#, \&, \%, 0)$.

Since we extend Brzozowski rules, we will use the same notation $d_a(T)$ to denote the derivative of a conflict-free extended regular expression T according to a symbol $a \in \Sigma$.

Notation 7.3 $(m^-, *-1)$. *In the following definitions, we use m^- to denote $max(m-1, 0)$, and assume that $* - 1 = *$.*

Definition 7.4. $first(T)$ is a function on regular expressions, defined as follows:

$$\begin{array}{lcl}
first(\emptyset) & = & \emptyset \\
first(\epsilon) & = & \emptyset \\
first(a\,[m..n]) & = & \{a\} \\
first(T_1 + T_2) & = & first(T_1) \cup first(T_2) \\
first(T_1 \cdot T_2) & = & \begin{cases} first(T_1) \cup first(T_2) & \text{if } \mathrm{N}(T_1) \\ first(T_1) & \text{otherwise} \end{cases} \\
first(T_1 \& T_2) & = & first(T_1) \cup first(T_2) \\
first(\%(T_1, \ldots, T_n)) & = & \bigcup_{i=1}^{n} first(T_i)
\end{array}$$

Definition 7.5. The function $d_a(T)$, where T is in cf-RE(#, &, %, 0) and a is a symbol, is defined as follows:

$$
\begin{array}{lll}
d_a(\epsilon) & =_{def} & \emptyset \\
d_a(\emptyset) & =_{def} & \emptyset \\
d_a(b\,[m..n]) & =_{def} & \begin{cases} \emptyset & \text{if } a \neq b \text{ or } n = 0 \\ b\,[m^{-}..n-1] & \text{otherwise} \end{cases} \\
d_a(T_1 + T_2) & =_{def} & \begin{cases} d_a(T_1) & \text{if } a \in first(T_1) \\ d_a(T_2) & \text{if } a \in first(T_2) \\ \emptyset & \text{otherwise} \end{cases} \\
d_a(T_1 \cdot T_2) & =_{def} & \begin{cases} d_a(T_1) \cdot T_2 & \text{if } a \in first(T_1) \\ d_a(T_2) & \text{if } a \in first(T_2) \text{ and } \mathrm{N}(T_1) \\ \emptyset & \text{otherwise} \end{cases} \\
d_a(T_1 \& T_2) & =_{def} & \begin{cases} d_a(T_1) \& T_2 & \text{if } a \in S(T_1) \\ d_a(T_2) \& T_1 & \text{if } a \in S(T_2) \\ \emptyset & \text{otherwise} \end{cases} \\
d_a(\%(T_1, \ldots, T_n)) & =_{def} & \begin{cases} d_a(T_i) \cdot \%(T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_n) & \text{if } \exists i.\ a \in first(T_i) \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

Note that for % types the function is defined on n-ary expressions, as the % operator is not associative. Furthermore, it can be observed that, thanks to conflict-freedom, all the cases in the definitions of $d_a(T_1 + T_2)$, $d_a(T_1 \cdot T_2)$, $d_a(T_1 \& T_2)$, and $d_a(\%(T_1, \ldots, T_n))$ are mutually exclusive.

The function $d_a(U)$ can be lifted to words in the following way: $d_\epsilon(U) = U$, $d_{aw}(U) = d_w(d_a(U))$.

It is easy to see that $d_a(U)$ is a Brzozowski's derivative, and that such derivative of a conflict-free expression is still a conflict-free expression.

We use $D(U)$ to denote the set of all $d_w(U)$-derivatives of U: $D(U) =_{def} \{d_w(U) \mid w \in \Sigma^*\}$. As it can be easily observed, $D(U)$ is finite. Unfortunately, $D(U\,[m..n])$ has a size that grows linearly with m, hence is exponential in $|U\,[m..n]\,|$, which makes $D(U)$ exponential in size with respect to U, in the worst case.

The membership algorithm based on Brzozowski's derivatives is reported in Figure 9. Given an input word w and a conflict-free type T, the algorithm scans w and, for each symbol a in w, it derives the current type according to a. If the obtained derivative is the empty expression, then the algorithm halts since w does not belong to $[\![T]\!]$; otherwise, the algorithm keeps iterating on the symbols in w until the last symbol has been read; if the last derivative is nullable, then $w \in [\![T]\!]$. The algorithm is quite natural and the proof of its correctness and completeness follows by a simple induction. Since DerivMember(w, T) computes a new derivative for each symbol being read, and the simplest algorithm to compute a derivative is in $O(|T|^2)$, its complexity is trivially in $O(|w| * |T|^2)$.

To improve the efficiency of the derivation process, each derivative being generated is simplified according to the following rules:

- $\epsilon \otimes T \rightarrow T, \quad \emptyset \otimes T \rightarrow \emptyset, \quad$ where $\otimes \in \{\cdot, \&\}$.

```
DerivMember(w, T)
1  Type U := T
2  for each symbol a in w
3      U := d_a(U)
4      if (U == ∅)
5          return false
6  if N(U)
7      return true
8  else return false
```

Fig. 9. Derivative-based membership algorithm.

Since the derivative of a conflict-free expression is still conflict-free, no non-trivial subexpression can appear more than once; hence, in our implementation there is no need to memoize the derivation process, as would happen in the case of unrestricted regular expressions.

7.2 Experimental Setup

We implemented the algorithms being tested in Java 1.8, and evaluated their performance on a 2.4 GHz Intel Core i7 machine with 8 GB of main memory and running Mac OSX 10.12.4. To avoid the perturbations introduced by system activity, we ran each experiment ten times, discarded the best and the worst performance, and computed the average of the remaining times: this means that, for each dataset, we evaluated the membership of each word ten times.

7.3 Datasets

In this experimental evaluation we analyze the performance and scalability of our algorithms when used for checking the membership of a word in the language generated by a regular expression in cf-RE(#, &) or cf-RE(#, &, %). Our experiments are based on several different datasets, generated as follows.

We perform distinct tests for expressions in cf-RE(#, &) and in cf-RE(#, &, %). In the cf-RE(#, &) case we evaluate the performance of all the binary and flat algorithms described in this paper (binary, binary with stability, flat, and flat with stability), and compare them with the baseline algorithm based on Brzozowski's derivatives, described in the previous section. In the cf-RE(#, &, %) case, instead, we limit our comparison to the flat algorithms, as unordered concatenation is not supported by the binary ones.

For both classes of regular expressions, we use positive as well as negative datasets. The first step in the creation of positive and negative datasets is the generation of a random regular expression T. Our generator essentially builds an AST of n-ary operators, and, then, transforms this AST into a type with unary and binary operators. The choices of the generator are driven by the following parameters:

- the expected depth of the n-ary AST (d_e);
- the expected number of children per operator (c_e);
- the probability of generating a union, a concatenation, an interleaving, or an unordered concatenation operator (p_u, p_c, p_i, p_{uc});
- the probability of generating an ϵ operator on a leaf (p_ϵ).

The generation algorithm uses these parameters to create an n-ary AST of depth $d_e + 1$. For each intermediate node in the AST, a union, concatenation, interleaving, or unordered

concatenation operator is generated according to probabilities p_u, p_c, p_i, p_{uc} by using the uniform distribution; these operators have a number of children randomly chosen according to the Poisson distribution with variance $\lambda = c_e$. When the generator reaches level $d_e - 1$ in the AST, if no interleaving and unordered concatenation operator has been previously generated, then it enforces their presence in the nodes of level $d_e - 1$. For each counting atom $a\,[m..n]$, furthermore, m and n values are randomly generated according to the uniform distribution, and n is set to $*$ with probability n_*.

For each leaf node, p_ϵ is used to choose between ϵ or an atom. To satisfy conflict-freedom, leaf symbols are chosen by assigning to each newly created atom a new symbol: in particular, the symbol generator exploits an integer counter initialized to 0 and incremented for each new generated symbol. Therefore, if T contains n atoms, then $S(T) = \{\text{“}0\text{”}, \ldots, \text{“}n-1\text{”}\}$.

Once an AST is generated, it is transformed into an actual type in the following way:

- n-ary union, concatenation, and interleaving operators are replaced with their binary counterparts in the obvious way;
- unordered concatenation operators, that are not supported by binary algorithms, are left untouched;
- if the type being generated is used for assessing the efficiency of flat algorithms, then union, concatenation, and interleaving operators are flattened again.

Our types were generated by using the above described scheme and the following parameter values:

- cf-RE(#, &): $d_e = 3$, $c_e = 8$, $p_u = 0.33$, $p_c = 0.33$, $p_i = 0.34$, $p_{uc} = 0$, $p_\epsilon = 0.25$, and $n_* = 0.01$;
- cf-RE(#, &, %): $d_e = 3$, $c_e = 8$, $p_u = 0.25$, $p_c = 0.25$, $p_i = 0.25$, $p_{uc} = 0.25$, $p_\epsilon = 0.25$, and $n_* = 0.01$.

Positive datasets were created by randomly generating a regular expression T, as described above, and, then, by using this expression to produce a dataset of 30000 words in $[\![T]\!]$ with minimum and maximum size ranging from 1000 to 5000 symbols. These words were generated from T by applying the following rules:

- for each atom $a\,[m..n]$, we use the uniform distribution to choose an integer $p \in [m, n]$ and output p occurrences of a; if $n = *$, then we choose p in the interval $[m, 2^{31} - 1]$ (Java maximum value for 32-bit integers);
- for each union type $T_1 + T_2$, we choose between T_1 and T_2 by generating a random boolean (uniform distribution);
- for each type $\%(T_1, \ldots, T_n)$, we generate words $w_1 \in [\![T_1]\!]$, $w_2 \in [\![T_2]\!]$, $\ldots$, $w_n \in [\![T_n]\!]$, and randomly choose an order for them;
- finally, for each type $T_1\&T_2$, we generate a word $w_1 \in [\![T_1]\!]$, a word $w_2 \in [\![T_2]\!]$, and shuffle them by performing a parallel scan and randomly choosing, at each step, the word from which the next symbol must be extracted.

For what concerns negative tests, we created two distinct datasets for each class of regular expressions, i.e., cf-RE(#, &) and cf-RE(#, &, %). The first dataset, for the given T, contains words that are very similar to words in T, while the second contains random words. In greater detail, the first dataset comprises 30000 negative words, with length ranging from 1000 to 5000 symbols, obtained by modifying positive words with a few wrong or misplaced symbols. We first generate a random regular expression T; this expression is then used to produce a random word $w \in [\![T]\!]$, as in the case of positive tests. w is, then, transformed into

Table 5. Datasets.

Dataset	Number of types	Number of words	Frequency (%)
Pos cf-RE(#, &)	1	30000	N/A
Pos cf-RE(#, &, %)	1	30000	N/A
Neg cf-RE(#, &)	1	30000	4.13
Neg cf-RE(#, &, %)	1	30000	7.78
Neg Random cf-RE(#, &)	1	30000	100
Neg Random cf-RE(#, &, %)	1	30000	100

Table 6. Types.

Dataset	Type size	Alphabet size	ϵ	Atoms	+	·	&	%
Pos cf-RE(#, &)	237	94	25	94	61	30	27	N/A
Pos cf-RE(#, &, %)	94	88	N/A	88	1	2	1	2
Neg cf-RE(#, &)	273	109	28	109	92	16	28	N/A
Neg cf-RE(#, &, %)	138	123	N/A	123	4	6	2	3
Neg Random cf-RE(#, &)	255	101	27	101	61	35	31	N/A
Neg Random cf-RE(#, &, %)	120	108	N/A	108	1	4	4	3

a negative word $\overline{w} \notin [\![T]\!]$, that is added to the dataset. The word transformation procedure works as follows.

(1) We fix to 10 the number of "violations" to be introduced in w.
(2) For each violation, a random and distinct position $p \in [1, length(w)]$ is chosen, and the symbol $w(p)$ is then replaced with a different symbol in $S(T) \cup \{x\}$, randomly chosen according to the uniform distribution, where $x \notin S(T)$; hence, in each word we introduce 10 misplaced or extraneous symbols.
(3) If $\overline{w} \in [\![T]\!]$, we apply again steps (1) and (2) until we get a word $\overline{w} \notin [\![T]\!]$.

The second negative dataset comprises again 30000 negative words, with length ranging from 1000 to 5000 symbols; in this case, however, after generating a random type T in cf-RE(#, &) or cf-RE(#, &, %), respectively, we created each word by randomly choosing a word length in $[1000, 5000]$ and by randomly picking, for each word position, a symbol in $S(T) \cup \{x\}$, where $x \notin S(T)$; any word that, by any chance, belongs to $[\![T]\!]$ is filtered out and replaced with a new word. As a consequence, this dataset contains 30000 words whose symbol set is $S(T) \cup \{x\}$ but are otherwise unrelated to T.

In Tables 5 and 6 we overview the characteristics of these datasets. In particular, Table 5 recalls the very basic information about the datasets (i.e., number of types, number of words, frequency of words with extraneous symbols), while Table 6 details, for each dataset, the features of the regular expressions being used (i.e., type size, alphabet size, number of occurrences for each operator).

7.4 Regular Expression Experiments

For the sake of clarity, in all the figures we present in this section, we divided the x-axis in disjoint buckets of 100 symbols (e.g., from 1001-1100 to 4901-5000), assigned words to buckets according to their size, and plotted for each bucket the average running time.

Positive Experiments. In our first experiment we compare the performance of our algorithms with that of the derivative-based algorithm on a positive sample of 30000 random words generated from a random type in cf-RE(#, &). This sample was generated as discussed in Section 7.3 and comprises words of size between 1000 and 5000 symbols. We use as input size the word length and measure the time required for completing the membership checking. The results we obtained are shown in Figures 10 and 11.

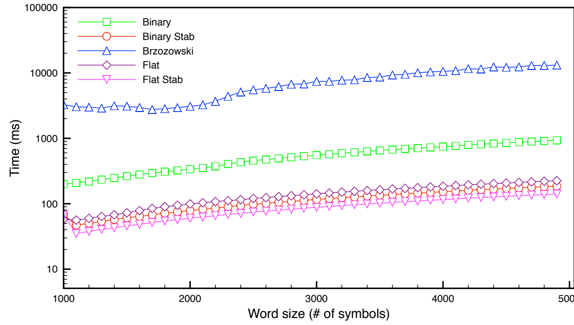


Fig. 10. Positive experiments for cf-RE(#, &): logarithmic scale.

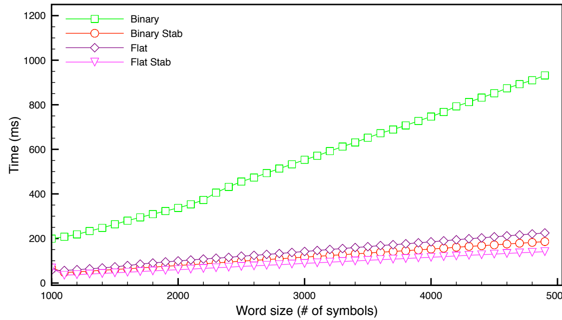


Fig. 11. Positive experiments for cf-RE(#, &): binary and flat algorithms.

In Figure 10 we plot our algorithms together with the derivative-based one. The figure shows that our algorithms are orders of magnitude faster than the derivative-based one, which is the reason why we had to use a logarithmic scale for the time axis.

In Figure 11 we excluded the data about the derivative-based algorithm and focused our attention on the binary and flat ones. As it can be observed, they are all very scalable, and the flat algorithm with stability is slightly faster than the binary one with stability and the flat one without stability; the binary algorithm without stability, instead, is the slowest constraint-based algorithm, but it still orders of magnitude faster than the derivative-based one. The huge performance gap between the binary algorithm without stability and the flat one, again without stability, is mostly due to the depth of the parse tree of T. Indeed, in most cases $flatdepth(T)$ is significantly lower than $depth(T)$: in particular, for the random type T used here, $flatdepth(T) = 4$ and $depth(T) = 15$, which implies that the binary algorithm must traverse in the parse tree of T a path of 15 nodes for each symbol being read, while the flat one traverses a path of only 4 nodes. The benefits provided by stability and by flattening constraints confirm what was indicated by the computational complexity analysis.

In our second experiment we move from cf-RE(#, &) to cf-RE(#, &, %), and compare the performance of our flat algorithms with that of the derivative-based one on a positive sample of 30000 random words generated from a random type in cf-RE(#, &, %). We restrict ourselves to the flat algorithms because of the presence of the % operator. As in the previous

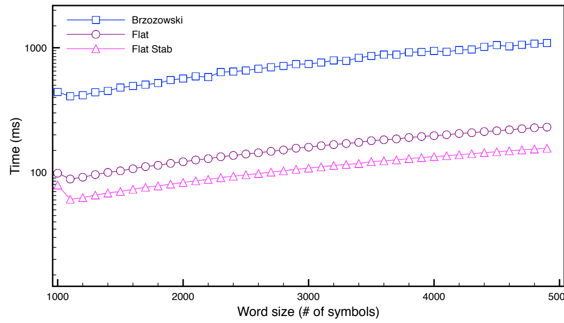


Fig. 12. Positive experiments for cf-RE(#, &, %): logarithmic scale.

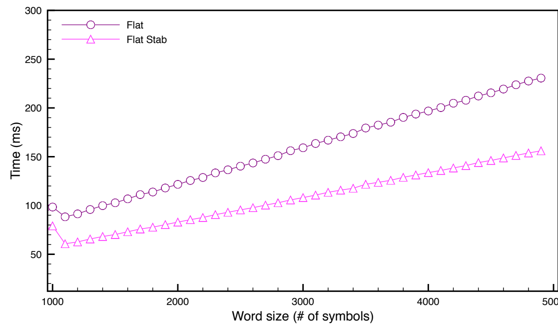


Fig. 13. Positive experiments for cf-RE(#, &, %): flat algorithms.

experiment, this sample was generated as discussed in Section 7.3 and comprises words of size ranging from 1000 to 5000 symbols. The results we obtained are shown in Figures 12 and 13, where we report, respectively, the results using a logarithmic scale on the y-axis, and the results for flat algorithms only.

As in the case of cf-RE(#, &), our algorithms are much faster than the derivative-based one, and there are several orders of magnitude of performance difference between the two classes of algorithms. In Figure 13, finally, we focus our attention on flat algorithms only. Both algorithms show a linear behaviour and, also in this case, stability seems to have a positive impact on performance.

To summarize, experiments on positive datasets show that our algorithms are much faster than the derivative-based competitor, and that, in particular, the flat algorithm with stability seems to represent the best option.

Negative Experiments. In these tests we evaluate the performance of our algorithms on two kinds of negative datasets, obtained, respectively, by introducing a few wrong or misplaced symbols in positive words, or by randomly generating unrelated words.

In both cases, we created datasets of 30000 words whose length ranges from 1000 to 5000 symbols. As in positive tests, for expressions in cf-RE(#, &) this comparison covers all the algorithms proposed here and the derivative-based one, while for expressions in cf-RE(#, &, %) it drops the binary ones.

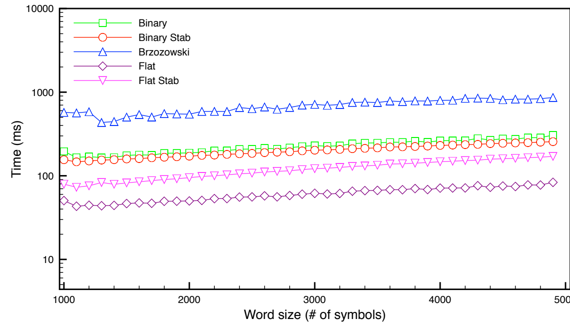


Fig. 14. Negative experiments for cf-RE(#, &): logarithmic scale on y-axis.

In Figure 14 we report the results obtained for regular expressions in cf-RE(#, &) and the first kind of datasets. As shown in Figure 14, where we used a logarithmic scale for the y-axis, the flat algorithm without stability outperforms the other ones, while the derivative-based algorithm appears to be the slowest one. The relative ineffectiveness of stability in negative tests for flat algorithms is not a surprise: indeed, stability is an optimization that optimistically assumes that the word being processed is in the semantics of the type, hence it pays a high cost the first time a symbol is met that is amortized when the same symbol is met again and again. However, the analysis of wrong words would typically stop much earlier than the end of the word, hence there is usually no time to recover the higher set-up cost. The problem is less serious in the binary case, where the gain offered by stability is higher, hence we need very few repetitions of a symbol in order to match the initial set-up cost.

In the case of regular expressions in cf-RE(#, &, %) the results we obtained, shown in Figure 15, are slightly different. We still observe that the flat algorithm without stability outperforms the others, and, specifically, is faster than the one with stability. However, in this case the Brzozowski algorithm is comparable to ours and is even faster than the flat algorithm with stability. Again, our algorithms are optimized for the positive case, since they only check bounds introduced by $a\,[m..n]$ operators once at the end of the word rather than at each step of the word analysis. We believe that this optimization is useful in the typical case when positive cases are the most common ones. In an applicative scenario where negative cases are common we could easily change the balance by checking for the violation of an $a\,[m..n]$ constraint every time the symbol a is met.

In Figures 16 and 17 we report the results obtained on negative random words for regular expressions in cf-RE(#, &) and cf-RE(#, &, %), respectively. Unlike what happens in all other tests, the derivative-based algorithm is now competitive with the flat one without stability, and both are definitely much faster than the other constraint-based algorithms. This is coherent with the fact that the derivative-based algorithm checks repetition bounds for each character being read while our algorithms wait until the end of the word. As in the other negative cases, stability seems to introduce a performance penalty rather than an advantage.

To summarize, experiments on negative samples show that the flat algorithm without stability should be taken into consideration when negative cases are common, and the derivative-based one may be considered if checks for fully random words dominate the workload.

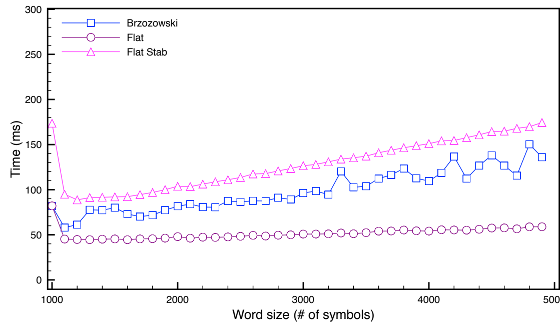


Fig. 15. Negative experiments for cf-RE(#, &, %).

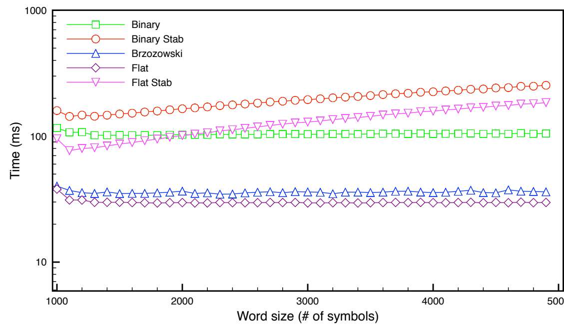


Fig. 16. Negative experiments for cf-RE(#, &): random words.

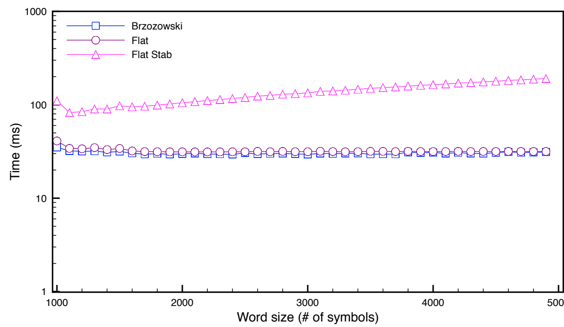


Fig. 17. Negative experiments for cf-RE(#, &, %): random words.

7.5 Schema Experiments

In this section we evaluate the performance of our algorithms when used to validate an XML document against an XML schema, as described in Section 6. We base our analysis on a dataset comprising 10 instances of the XMark benchmark, whose size ranges from 110 MB to 1100 MB and compare the performance of validators using our algorithms with that of a validator built around the derivative-based algorithm, as well as with the validator of Relax

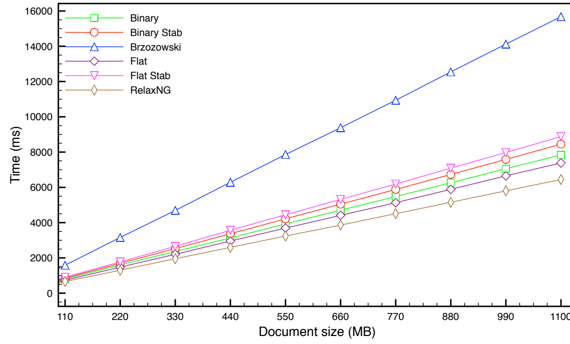


Fig. 18. Schema experiments: no caching.

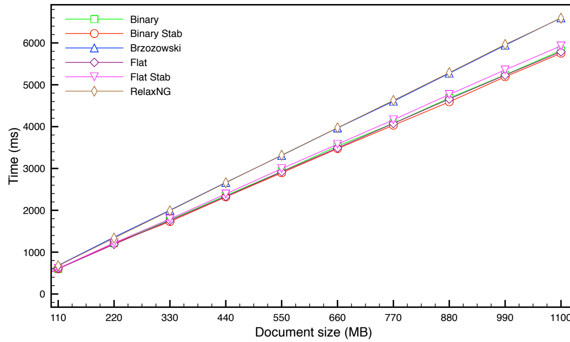


Fig. 19. Schema experiments: caching.

NG. The inclusion of this validator in the comparison is motivated by the fact that the Relax NG schema language supports interleaving as well as a constraint on symbol occurrences that is close to conflict-freedom.

In Figure 18 we show the results we got in our experiment. As can be observed, the validator of Relax NG slightly outperforms our algorithms, and its performance is very close to that of the flat algorithm without stability. This could be explained by the fact that Relax NG internally uses a cache, while our implementation has no caching.

As an XML document may contain several fragments having the same structure, we modified our implementation by means of a form of *memoization* through a fully associative LRU cache of 200 entries. In particular, we memoized the results of the word-membership algorithm and used memoization in the `endElement` method of the SAX handler. The results we obtained are shown in Figure 19.

As can be observed, our algorithms are now faster than the validator of Relax NG, which shows a performance very close to that of the Brzozowski algorithm with caching.

To summarize, the algorithms described here can represent a viable option for validating an XML document against a schema with and without some form of memoization.

8 RELATED WORK

Membership checking is one of the basic problems involving regular expressions. For plain REs membership testing is in PTIME (see [31]) and membership can be checked just by

translating the input regular expression into an equivalent NFA. For regular expressions with interleaving, counting, and/or unordered concatenation, however, the membership problem becomes much more complex. In the following, hence, we will first review the most prominent complexity results for the membership checking problem, discuss alternative restricted classes of regular expressions and constraint-based inclusion algorithms, and, then, describe a few evaluation techniques proposed in the past.

8.1 Complexity Results

In [35] Mansfield, as well as Warmuth and Hausler in [44], proved that the membership problem for RE(&) is NP-hard, even when union, Kleene star, and counting are not present. In [36] Mayer and Stockmeyer improved these results by showing that this problem is NP-complete and that it remains NP-hard even if & appears just once in the regular expression. They also proved that, just by combining interleaving with union or Kleene star alone, the problem remains intractable.

In [32] Kilpeläinen and Tuhkanen focused on RE(#) and showed that, despite the fact that regular expressions in RE(#) are exponentially more succinct than plain REs, their membership problem is still tractable.

In [30] Hovland analyzed the complexity of membership for RE(#, %) and proved that this problem is NP-complete and it remains NP-complete even if counting is omitted and REs are limited to those with a single occurrence of % at the top level.

In [24] Gelade et al. analyzed the interaction between interleaving and counting. They focused their research on inclusion and equivalence, but also proved the membership for RE(#, &) is in PSPACE by defining a special kind of automaton for which membership is in PSPACE.

8.2 Restricted Classes of Regular Expressions

As briefly pointed out in the Introduction and in Remark 1, conflict-freedom has already been used in the past to design type inclusion algorithms [16–20, 25], and, furthermore, it is not the only restriction that has been proposed in the literature with the aim of lowering the cost or the computational complexity of a given class of operations. Here we will first discuss constraint-based type inclusion algorithms, and, then, survey alternative restrictions for regular expressions, even though none of them has been applied to regular expressions with interleaving.

In [16–20, 25] we first proposed conflict-freedom as a convenient restriction to identify a class of regular expressions with interleaving and counting for which inclusion can be decided in polynomial time. We used essentially the same set of constraints we exploited here, but our inclusion algorithms are not based on constraint residuation but, rather, on constraint implication. Starting from [20, 25], where we proposed a cubic algorithm, in [18, 19] we lowered the complexity to quadratic time and extended our approach to asymmetric inclusions of the form $T_1 <: T_2$, where T_2 must satisfy conflict-freedom and T_1 can be any type; finally, we described in [16, 17] an optimized algorithm that, by means of syntax-driven inclusion rules, has an almost linear running time. While the asymmetric algorithms of [16–19] can also be used for membership checking, as membership can always be seen as a special kind of asymmetric inclusion, the algorithms we are proposing here have a better time complexity and have been fine-tuned and optimized for membership.

Conflict-free DTDs [3, 4] as well as duplicate-free DTDs [38, 45] are plain DTDs where each element type is described by a single-occurrence plain regular expression, i.e., a regular expression when each alphabet symbol may appear only once. These DTDs have been

proposed for lowering the cost of incremental validation and query satisfiability checking, respectively; compared to conflict-free regular expressions, they do not impose any constraint on the use of Kleene star, while they do not support interleaving, counting, and unordered concatenation.

Single Occurrence Regular Expressions (SORE) have been introduced by Bex et al. in [6, 7] in their studies about the inference of concise DTDs and XSDs. In these expressions no symbol can appear twice, even though there is no restriction on the use of the Kleene star.

CHain Regular Expressions (CHARE) are a refinement of SORE with further constraints and have been introduced in [6]. In essence, a CHARE r is a single-occurrence expression consisting of the concatenation of factors of the form $(a_1^{p_1} + \ldots + a_n^{p_n})^q$, where $p_1, \ldots, p_n, q \in \{*, +, ?\}$ are optional quantifiers. The ability to add an external quantifier makes CHAREs slightly more expressive than the conflict-free types that we presented here. For example, the CHARE $(a_1 + \ldots + a_n)^*$ can be expressed by the conflict-free type $(a_1^* \& \ldots \& a_n^*)$, but $(a_1 + \ldots + a_n)^+$ requires one more type operator $T!$ that deletes the empty string from a type. The extension to conflict-free types with $T!$ is trivial, makes conflict-free types more expressive than CHARES, and is discussed in [20]. In [24] Gelade et al. extended CHAREs with addends of the form $a\,[m..n]$. Nesting of counting quantifiers makes these extended CHAREs much more expressive that conflict-free types.

8.3 Membership Checking Approaches

Most approaches for checking the membership of a word in the language generated by a regular expression are based on the use of finite automata and differ in the kind of automaton being used and in the way the automaton is built.

For plain regular expressions, membership is usually checked by creating an equivalent NFA with the usual construction process, and by verifying whether the NFA accepts the input word. In the case of 1-unambiguous regular expressions [12] it is also possible to create an equivalent DFA in polynomial time by relying on the Glushkov construction process [27] or on Brzozowski automata [14]. Brzozowski's derivatives can also be directly used to check for membership without building an automaton, as detailed in the previous section.

For regular expressions in RE(#), Gelade et al. in [23] introduced Counter Automata (CNFA). A counter automaton is an NFA enriched with a set of *counter variables* C and a set of *guards* (boolean predicates) over counter variables. In a CNFA A, any transition is labeled with a symbol from a finite alphabet, a set of guards over counter variables, and a set of basic updates on counter variables: a transition from state q_i to state q_j, hence, can be fired if A, when in state q_i, reads the symbol labeling the transition and guards are satisfied by the current assignment of values to counter variables; after firing the transition, A moves to state q_j and updates counter variables. Counter automata can be built from regular expressions in RE(#) in polynomial time by applying the extended Glushkov construction described by Sperberg-McQueen in [41]. In [8] Björklund et al. used CNFA to address the traditional membership problem for RE(#) as well as its incremental variant, where words are subject to updates, insertions, and deletions [2, 39], by proposing an approach based on the construction of a tree intermediate data structure; an extensive experimental evaluation proved that the authors' approach is very efficient, while it is not clear whether the intermediate data structure can be built in polynomial time for non-trivial expressions in RE(#).

Similar automata have also been proposed by Kilpeläinen and Tuhkanen in [33]. The same authors proposed in [32] a radically different approach for testing the membership of a word in the language generated by an expression r in RE(#): this approach is based on a

dynamic programming algorithm that traverses bottom up the abstract syntax tree of r and decorates each node of the AST with relations over states. Automata with counters have also been studied by Smith et al. in [40] in the context of deep packet inspection for network appliances.

In [24] Gelade et al. described a kind of automata suitable for modeling regular expressions in RE(#, &). These automata are essentially NFAs extended with *split* and *merge* transitions, needed for translating interleaving operators, as well as counters on states and *counting* transitions, necessary for counting operators. Differently from CNFA, here counters are directly associated to automaton states and there are no counter variables. An alternative kind of automata for expressions in RE(#, &) has been described by Dal-Zilio and Lugiez in [21] and is based on the use of Presburger formulas for dealing with both interleaving and counting.

In [30] Hovland proposed *Finite Automata with Counters* (FACs) to model regular expressions in RE(#, %). The main idea of this new kind of automata is to associate a counter to each counting and unordered concatenation operator in the input regular expression. Counters, hence, are directly bound to the original regular expression and are not associated to states. As in [23], transitions may change the current state of an automaton, as well as update and reset counter values.

REMARK 2. *An extended abstract of this paper appeared in [26]. Compared to that conference version, there are four major additions:*

- *In [26] we presented a very naive form of stability applied to binary and flat algorithms. Here we fully develop stability and completely rewrite our algorithms to exploit stability without any complexity and performance penalty.*
- *We introduce the support for unordered concatenation, that was missing in [26].*
- *We provide here all the proofs that were omitted in the conference version; some of these proofs are definitely non-trivial.*
- *In [26] we presented a very partial experimental evaluation of our algorithms in the context of XSD schemas. Here, we provide a very detailed performance analysis of our algorithms, mostly focused on regular expressions, and study their behaviour on large positive and negative datasets.*

9 CONCLUSIONS

While membership checking is polynomial for plain regular expressions, it quickly becomes intractable when more flexible operators, like interleaving and unordered concatenation, are considered. In this paper we presented algorithms to verify whether a word w belongs to the language generated by a regular expression T with interleaving, counting, and unordered concatenation, provided that T satisfies a conflict-free restriction on the use of symbols and counting operators, a restriction that is quite severe but seems to be extremely common in practice. The fastest algorithm runs in linear time, and an extensive experimental evaluation proves that our algorithms perform very well on positive datasets as well as on negative datasets comprising words with wrong or misplaced symbols; given that cardinality constraints are checked at the end of word scan, they may experience performance issues on fully random negative datasets formed by words unrelated to the input type.

REFERENCES

[1] Carlos Buil Aranda, Marcelo Arenas, Óscar Corcho, and Axel Polleres. 2013. Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. *J. Web Sem.* 18, 1 (2013), 1–17.

[2] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. 2004. Incremental validation of XML documents. *ACM Trans. Database Syst.* 29, 4 (2004), 710–751.

[3] Denilson Barbosa, Gregory Leighton, and Andrew Smith. 2006. Efficient Incremental Validation of XML Documents After Composite Updates. In *XSym (LNCS)*, Vol. 4156. Springer, 107–121.

[4] Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. 2004. Efficient Incremental Validation of XML Documents.. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004*. IEEE Computer Society, 671–682.

[5] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. 2004. DTDs versus XML Schema: A Practical Study. In *Proceedings of the Seventh International Workshop on the Web and Databases, WebDB 2004, June 17-18, 2004, Colocated with ACM SIGMOD/PODS 2004*. 79–84.

[6] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. 2010. Inference of concise regular expressions and DTDs. *ACM Trans. Database Syst.* 35, 2 (2010), 11:1–11:47.

[7] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. 2007. Inferring XML Schema Definitions from XML Data. In *VLDB*. 998–1009.

[8] Henrik Björklund, Wim Martens, and Thomas Timm. 2015. Efficient Incremental Evaluation of Succinct Regular Expressions. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, CIKM 2015*. ACM, 1541–1550.

[9] Iovka Boneva, Radu Ciucanu, and Slawek Staworko. 2013. Simple Schemas for Unordered XML. In *Proceedings of the 16th International Workshop on the Web and Databases 2013, WebDB 2013*. 13–18.

[10] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. 2006. *Extensible Markup Language (XML) 1.1 (Second Edition)*. Technical Report. World Wide Web Consortium. W3C Recommendation.

[11] Anne Brüggemann-Klein. 1993. Unambiguity of Extended Regular Expressions in SGML Document Grammars. In *Algorithms - ESA '93, First Annual European Symposium, Bad Honnef, Germany, September 30 - October 2, 1993, Proceedings (Lecture Notes in Computer Science)*, Vol. 726. Springer, 73–84.

[12] Anne Brüggemann-Klein and Derick Wood. 1992. Deterministic Regular Languages. In *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings (Lecture Notes in Computer Science)*, Vol. 577. Springer, 173–184.

[13] Anne Brüggemann-Klein and Derick Wood. 1998. One-Unambiguous Regular Languages. *Inf. Comput.* 142, 2 (1998), 182–206.

[14] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494.

[15] Byron Choi. 2002. What are real DTDs like?. In *WebDB*. 43–48.

[16] Dario Colazzo, Giorgio Ghelli, Luca Pardini, and Carlo Sartiani. 2009. Linear inclusion for XML regular expression types. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*. ACM, 137–146.

[17] Dario Colazzo, Giorgio Ghelli, Luca Pardini, and Carlo Sartiani. 2013. Almost-linear inclusion for XML regular expression types. *ACM Trans. Database Syst.* 38, 3 (2013), 15.

[18] Dario Colazzo, Giorgio Ghelli, Luca Pardini, and Carlo Sartiani. 2013. Efficient asymmetric inclusion of regular expressions with interleaving and counting for XML type-checking. *Theor. Comput. Sci.* 492 (2013), 88–116.

[19] Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2009. Efficient asymmetric inclusion between regular expression types. In *ICDT (ACM International Conference Proceeding Series)*, Ronald Fagin (Ed.), Vol. 361. ACM, 174–182.

[20] Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2009. Efficient inclusion for a class of XML types with interleaving and counting. *Inf. Syst.* 34, 7 (2009), 643–656.

[21] Silvano Dal-Zilio and Denis Lugiez. 2003. XML Schema, Tree Logic and Sheaves Automata. In *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings (Lecture Notes in Computer Science)*, Vol. 2706. Springer, 246–263.

[22] David C. Fallside and Priscilla Walmsley. 2004. XML Schema Part 0: Primer – Second Edition. (Oct 2004). W3C Recommendation.

[23] Wouter Gelade, Marc Gyssens, and Wim Martens. 2012. Regular Expressions with Counting: Weak versus Strong Determinism. *SIAM J. Comput.* 41, 1 (2012), 160–190.

[24] Wouter Gelade, Wim Martens, and Frank Neven. 2009. Optimizing Schema Languages for XML: Numerical Constraints and Interleaving. *SIAM J. Comput.* 38, 5 (2009), 2021–2043.

[25] Giorgio Ghelli, Dario Colazzo, and Carlo Sartiani. 2007. Efficient Inclusion for a Class of XML Types with Interleaving and Counting. In *Database Programming Languages, 11th International Symposium,*

DBPL 2007, Vienna, Austria, September 23-24, 2007, Revised Selected Papers (Lecture Notes in Computer Science), Marcelo Arenas and Michael I. Schwartzbach (Eds.), Vol. 4797. Springer, 231–245.

[26] Giorgio Ghelli, Dario Colazzo, and Carlo Sartiani. 2008. Linear time membership in a class of regular expressions with interleaving and counting. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*. ACM, 389–398.

[27] V M Glushkov. 1961. The Abstract Theory of Automata. *Russian Mathematical Surveys* 16, 5 (1961), 1.

[28] Charles F. Goldfarb. 1990. *SGML handbook*. Clarendon Press.

[29] Steve Harris and Andy Seaborne. 2013. *SPARQL 1.1 Query Language*. Technical Report. World Wide Web Consortium. W3C Recommendation.

[30] Dag Hovland. 2012. The Membership Problem for Regular Expressions with Unordered Concatenation and Numerical Constraints. In *Language and Automata Theory and Applications - 6th International Conference, LATA 2012, A Coruña, Spain, March 5-9, 2012. Proceedings (Lecture Notes in Computer Science)*, Adrian Horia Dediu and Carlos Martín-Vide (Eds.), Vol. 7183. Springer, 313–324.

[31] J.E. Hopcroft and J.D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

[32] Pekka Kilpeläinen and Rauno Tuhkanen. 2003. Regular Expressions with Numerical Occurrence Indicators - preliminary results. In *Proceedings of the Eighth Symposium on Programming Languages and Software Tools, SPLST'03, Kuopio, Finland, June 17-18, 2003*, Pekka Kilpeläinen and Niina Päivinen (Eds.). University of Kuopio, Department of Computer Science, 163–173.

[33] Pekka Kilpeläinen and Rauno Tuhkanen. 2004. Towards efficient implementation of XML schema content models. In *Proceedings of the 2004 ACM Symposium on Document Engineering, Milwaukee, Wisconsin, USA, October 28-30, 2004*. ACM, 239–241.

[34] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. 2016. Querying Graphs with Data. *J. ACM* 63, 2 (2016), 14.

[35] Anthony Mansfield. 1983. On the computational complexity of a merge recognition problem. *Discrete Applied Mathematics* 5, 1 (1983), 119 – 122.

[36] Alain J. Mayer and Larry J. Stockmeyer. 1994. Word Problems — This Time with Interleaving. *Inf. Comput.* 115, 2 (1994), 293–311.

[37] Anders Møller. 2010. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. (2010). `http://www.brics.dk/automaton/`.

[38] Manizheh Montazerian, Peter T. Wood, and Seyed R. Mousavi. 2007. XPath Query Satisfiability is in PTIME for Real-World DTDs. In *XSym (LNCS)*, Vol. 4704. Springer, 17–30.

[39] Sushant Patnaik and Neil Immerman. 1997. Dyn-FO: A Parallel, Dynamic Complexity Class. *J. Comput. Syst. Sci.* 55, 2 (1997), 199–209.

[40] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. 2008. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008*. ACM, 207–218.

[41] C. M. Sperberg-McQueen. 2004. *Notes on finite state automata with counters*. Technical Report. Available at `http://www.w3.org/XML/2004/05/msm-cfa.html`.

[42] C. M. Sperberg-McQueen. 2005. Applications of Brzozowski derivatives to XML Schema processing. In *Proceedings of the Extreme Markup Languages® 2005 Conference*.

[43] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. 2004. *XML Schema Part 1: Structures Second Edition*. Technical Report. World Wide Web Consortium. W3C Recommendation.

[44] Manfred K. Warmuth and David Haussler. 1984. On the Complexity of Iterated Shuffle. *J. Comput. Syst. Sci.* 28, 3 (1984), 345–358.

[45] Peter T. Wood. 2003. Containment for XPath Fragments under DTD Constraints. In *Proceedings of the 9th International Conference on Database Theory - ICDT 2003, Siena, Italy, January 8-10, 2003 (Lecture Notes in Computer Science)*, Vol. 2572. Springer, 297–311.

......

Received ...