Linear Time Membership for a Class of XML Types with Interleaving and Counting

Giorgio Ghelli
Dipartimento di Informatica - Università di Pisa
ghelli@di.unipi.it

Dario Colazzo
LRI - Université Paris Sud
dario.colazzo@lri.fr

Carlo Sartiani
Dipartimento di Informatica - Università di Pisa
sartiani@di.unipi.it

Abstract

Regular Expressions (REs) form the basis of most XML type languages, such as DTDs, XML Schema types, and XDuce types (Thompson et al. 2004; Hosoya and Pierce 2003). In this context, the interleaving operator would be a natural addition to the language of REs, as witnessed by the presence of limited forms of interleaving in XSD (the *all* group), Relax-NG, and SGML. Unfortunately, membership checking for REs with interleaving is NP-hard in general. We present here a restricted class of REs with interleaving and counting which admits a linear membership algorithm. This restricted class is known to be expressive enough for the vast majority of the content models used in real-world DTDs and XSD schemas; moreover, we have proved in (Ghelli et al. 2007) that the same class admits a polynomial algorithm for subtyping and type-equivalence, problems which are EXPSPACE-complete for the full language of REs with interleaving.

We first present an algorithm for membership of a list of words into a RE with interleaving and counting, based on the translation of the RE into a set of constraints. We generalize the approach in order to check membership of XML trees into a class of EDTDs with interleaving and counting, which models the crucial aspects of DTDs and XSD schemas. Finally, we extend the approach to REs with intersection.

1. Introduction

Given a family L of extended, or restricted, regular expressions, membership for L is the problem of determining whether a word w belongs to the language generated by an expression e in L. The problem is polynomial (in $|w| + |e|$) when L is the set of standard regular expressions based on union, concatenation, and Kleene star, which we denote as $RE\{+,\cdot,*\}$ (J.E. Hopcroft and J.D. Ullman 1979), and is still polynomial when intersection $\cap$ is added to the operators. However, membership is NP-complete for $RE\{+,\cdot,*,\&\}$, the set of REs extended with an interleaving operator $\&$ (to be defined later) (Mayer and Stockmeyer 1994). The simpler combinations $RE\{\cdot,\&\}$, $RE\{+,\&\}$ and $RE\{*,\&\}$ are already NP-hard (Mayer and Stockmeyer 1994), showing that the NP-hardness of membership with interleaving is quite robust.

We present here a restricted set of regular expressions with interleaving which admits a linear-time membership algorithm (using the RAM model to assess time complexity). The set contains those regular expressions where no symbol appears twice (called *conflict-free* or *single-occurrence*) and where Kleene star is only applied to symbol disjunctions. We generalize Kleene star to counting constraints such as $a\,[2..5]$ or $a\,[1..*]$, and we also include intersection. We show how this approach can be generalized to check membership of XML trees into a class of EDTDs with interleaving and counting, obtaining an algorithm whose time complexity is linear in the product of the input size with the maximal alternation depth of all the content models in the schema. In this context, the restriction we are proposing is known to be met by the vast majority of schemas that are produced in practice (Ghelli et al. 2007).

Our approach is based on the translation of each regular expression into a set of constraints, as in (Ghelli et al. 2007). We define a linear-time translation algorithm, and we then define a linear-time algorithm to check that a word satisfies the resulting constraints. The algorithm is 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 the Brzozowski derivative of REs (Brzozowski 1964).

2. Type Language and Constraint Language

2.1 The Type Language

We follow the terminology of (Ghelli et al. 2007), and we use the term *type* instead of *regular expression*. We consider the following type language with counting, disjunction, concatenation, and interleaving, for strings 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 $a\,[m..n]$ 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 $((a_1\,[0..*])\& \ldots \&(a_n\,[0..*]))$.

To formally define the semantics of these types, we adopt the usual definitions for string concatenation $w_1 \cdot w_2$, and for the concatenation of two languages $L_1 \cdot L_2$. The *shuffle*, or *interleaving*, operator $w_1 \& w_2$ is also standard, and 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 string ϵ.*

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

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 string w, $S(w)$ is the set of all symbols appearing in w. For any type T, $Atoms(T)$ is the set of all atoms $a\,[m..n]$ appearing in T, and $S(T)$ is the set of all 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 string 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:*

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

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

We can now define 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 subterms $a\,[m..n]$ and $a'\,[m'..n']$ that occur in T, a is different from a'.

Remark 2.7 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 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 (Wood 2003; Montazerian et al. 2007), as "Single Occurrence REs" (SOREs) in (Bex et al. 2006, 2007), as "conflict-free DTDs" in (Barbosa et al. 2004, 2006). The specific limitation that we impose on Kleene-star is reminiscent of Chain REs (CHAREs), as defined in (Bex et al. 2006), 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 (Barbosa et al. 2006). Similar results about the prevalence of *simple* content models had been reported in (Choi 2002).

■

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

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

$$\begin{array}{rl} F\ ::= & A^+\ \mid\ A^+ \mapsto B^+\ \mid\ a?[m..n]\ \mid\ \mathrm{upper}(A) \\ & \mid\ a \prec b\ \mid\ F \wedge F'\ \mid\ \mathbf{true} \end{array}$$

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

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

We use the following abbreviations:

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

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

Proposition 2.8 $w \models a \prec\succ b \Leftrightarrow$ *a and b are not both in $S(w)$.*

Proposition 2.9 $w \models A \prec\succ B\ \Leftrightarrow\ w \not\models A^+ \wedge B^+$

Proposition 2.10 $w \models A^-\ \Leftrightarrow\ w \not\models A^+$

2.3 Constraint Extraction

We can now define the extraction of constraints from types.

To each type T, we associate a formula $S^+(T)$ that tests for the presence of one of its symbols; $S^+(T)$ is defined as $(S(T))^+$.

We can now endow a type T with five sets of constraints on every word $w \in [\![T]\!]$. We start with those constraints whose definition is *flat*, since they only depend on the leaves of the syntax tree of T:

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

Definition 2.11 (Flat constraints)

Lower-bound:

$$SIf(T) \quad =_{def} \quad \begin{cases} S^+(T) & \textit{if not } \mathrm{N}(T) \\ \mathbf{true} & \textit{if } \mathrm{N}(T) \end{cases}$$

Cardinality:

$$\mathrm{ZeroMinMax}(T) \quad =_{def} \quad \bigwedge_{a[m..n] \in Atoms(T)} a?[m..n]$$

Upper-bound:

$$\mathrm{upperS}(T) \quad =_{def} \quad \mathrm{upper}(S(T))$$

Flat constraints:

$$\mathcal{FC}(T) \quad =_{def} \quad SIf(T) \wedge \mathrm{ZeroMinMax}(T) \wedge \mathrm{upperS}(T)$$

We add now the *nested constraints*, whose definition depends on the internal structure of T; the quantification "for any $w \in [\![C[T']]\!]$" below means "for any $w \in T$ where T is any type with a subterm T'". All the nested constraints depend on the fact that T is conflict-free.

- *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 in $S(T_1)$ and also a symbol in $S(T_2)$.

In the formal definition below, $If_{T_2}(S^+(T_1) \Rrightarrow S^+(T_2))$ denotes, by definition, $\mathtt{true}$ when $\mathrm{N}(T_2)$, and $(S(T_1))^+ \Rrightarrow (S(T_2))^+$ otherwise. Observe that the exclusion constraints are actually encoded as order constraints.

Definition 2.12 (Nested constraints)

Co-occurrence:

$$\begin{aligned} \mathcal{CC}(T_1 \otimes T_2) \quad &=_{def} \quad If_{T_2}(S^+(T_1) \Rrightarrow S^+(T_2)) \\ &\qquad \wedge\, If_{T_1}(S^+(T_2) \Rrightarrow S^+(T_1)) \\ \mathcal{CC}(T) \quad &=_{def} \quad \mathbf{true} \qquad \textit{otherwise} \end{aligned}$$

Order/exclusion:

$$\begin{aligned} \mathcal{OC}(T_1 + T_2) \quad &=_{def} \quad S(T_1) \prec\succ S(T_2) \\ \mathcal{OC}(T_1 \cdot T_2) \quad &=_{def} \quad S(T_1) \prec S(T_2) \\ \mathcal{OC}(T) \quad &=_{def} \quad \mathbf{true} \qquad \textit{otherwise} \end{aligned}$$

Nested constraints:

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

By definition, when either A or B is "$\emptyset$", both $A \prec B$ and $A \prec\succ B$ are $\mathtt{true}$, hence the order constraint associated to a node where one child has $S(T_i) = \emptyset$ is trivial; this typically happens with a subterm $T + \epsilon$. For an example of nested constraint extraction, see the upper part of Figure 1.

The following theorem is proved in (Ghelli et al. 2007) and states that constraints provide a sound and complete characterization of type semantics.

Theorem 2.13 *Given a conflict-free type T, it holds that:*

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

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 $depth(\epsilon) = depth(a\,[m..n]) = 1$, $depth(T_1 \circledast T_2) = 1 + \max(depth(T_1), depth(T_2))$, where $\circledast$ stands for any binary operator. 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, according to Table 1. We write $F \xrightarrow{a} F'$ to specify that F is transformed into F' by a; in all the cases not covered by Table 1, 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 is extended from symbols to non-empty sequences of symbols in the obvious way:

$$\begin{array}{lllll} & F \xrightarrow{a} F' & & \Rightarrow & F \xrightarrow{a}{}^{+} F' \\ w \neq \epsilon: & F \xrightarrow{a} F' \ \wedge\ F' \xrightarrow{w}{}^{+} F'' & & \Rightarrow & F \xrightarrow{aw}{}^{+} F'' \end{array}$$

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 $\mathbf{false}$. This is formalized by the relations $F \rightarrow^{\epsilon} G$ and $F \xrightarrow{w}{}^{*} G$, defined below, with $G \in \{\mathbf{true}, \mathbf{false}\}$.

$$\begin{array}{llll} A^+ \Rrightarrow B^+ \rightarrow^{\epsilon} \mathbf{true} & & A^+ \Lleftarrow\!\!\Rrightarrow B^+ \rightarrow^{\epsilon} \mathbf{true} & \\ A^+ & \rightarrow^{\epsilon} \mathbf{false} & \mathbf{false} & \rightarrow^{\epsilon} \mathbf{false} \\ A \prec B & \rightarrow^{\epsilon} \mathbf{true} & A \prec\succ B & \rightarrow^{\epsilon} \mathbf{true} \\ A^- & \rightarrow^{\epsilon} \mathbf{true} & \mathbf{true} & \rightarrow^{\epsilon} \mathbf{true} \end{array}$$

$$\begin{array}{lll} F \rightarrow^{\epsilon} G & \Rightarrow & F \xrightarrow{\epsilon}{}^{*} G \\ F \xrightarrow{w}{}^{+} F' \ \wedge\ F' \rightarrow^{\epsilon} G & \Rightarrow & F \xrightarrow{w}{}^{*} G \end{array}$$

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

Lemma 3.1 (Residuation) $w \models F$ *iff* $F \xrightarrow{w}{}^{*} \mathbf{true}$.

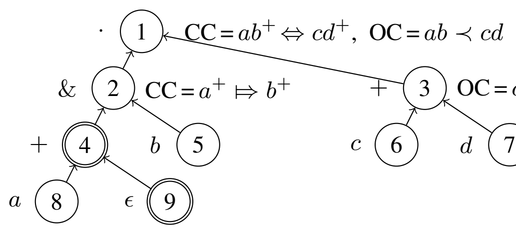
Residuation gives us immediately a membership algorithm of complexity $O(|w| * |\mathcal{NC}(T)|)$: for each symbol a in w, and for each constraint F in $\mathcal{NC}(T)$, we substitute F with the residual F' such that $F \xrightarrow{a} F'$. The word w is in T iff no $\mathbf{false}$ or A^+ is in the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

However, as it will be discussed later, 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, encoded as a set of nodes and a *Parent*[] array, such that, for each node n, Parent$[n]$ is either null or a pair (n_p,*direction*); 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 same nodes:

Co-occurrence	Condition	$a \in A$	$a \in B$	$a \in A$	$a \in B$	$a \in A$
	Constraint	$A^+ \Mapsto B^+$	$A^+ \Mapsto B^+$	$A^+ \Leftrightarrow B^+$	$A^+ \Leftrightarrow B^+$	A^+
	Residual	B^+	$\mathbf{true}$	B^+	A^+	$\mathbf{true}$
Order	Condition	$a \in A$	$a \in B$	$a \in A$	$a \in B$	$a \in A$
	Constraint	$A \prec B$	$A \prec B$	$A \prec\succ B$	$A \prec\succ B$	A^-
	Residual	$A \prec B$	A^-	B^-	A^-	$\mathbf{false}$

Table 1. Computing the residual of a nested constraint



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		$\prec\succ$
4		
5		
6		
7		
8		
9		

NodeOfSym

a	8
b	5
c	6
d	7

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

Nullable

false

Figure 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

- 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 $\{\Leftrightarrow, \Mapsto, \Mapsfrom, L^+, R^+, \mathbf{true}\}$ that specifies the status of the associated co-occurrence constraint, as follows:
  - $\Leftrightarrow$: encodes ${A_1}^+ \Leftrightarrow {A_2}^+$;
  - $\Mapsto$: encodes ${A_1}^+ \Mapsto {A_2}^+$;
  - $\Mapsfrom$: encodes ${A_2}^+ \Mapsto {A_1}^+$;
  - L^+: encodes the residual constraint A_1^+;
  - R^+: encodes the residual constraint A_2^+;
  - $\mathbf{true}$: encodes $\mathbf{true}$.
- OC[]: for each node n, OC[n] specifies the status of the associated order constraint, and may assume the following values:
  - $\prec$: encodes $A_1 \prec A_2$;
  - $\prec\succ$: encodes $A_1 \prec\succ A_2$;
  - L^-: encodes the residual constraint ${A_1}^-$;
  - R^-: encodes the residual constraint ${A_2}^-$;
  - $\mathbf{false}$: encodes the residual constraint $\mathbf{false}$.
- NodeOfSymbol[]: NodeOfSymbol[a] is the only node nd associated with a type $a\,[m..n]$, for some m and n, and is null if no such node exists[1];

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

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

Table 2. Initialization of constraint annotations.

- Min[]/Max[]: Min[a] and Max[a], when different from null, encode a constraint $a?$[Min[a]..Max[a]].
- Nullable: Nullable is not an array, but is just a boolean that is *true* iff T is nullable.

Table 2 reports the constraint symbols that are initially associated with a node n that corresponds to a subterm T' of the input type. The co-occurrence constraint depends on the nullability of T_1 and T_2.

After the constraint representation has been built, the 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 A^+ constraints that are so generated. At the end of w, it checks that all the A^+ constraints have been further residuated into $\mathbf{true}$. 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.

Example 3.2 Consider the type $T = ((a+\epsilon)\&b\,[1..5])\cdot(c+d^+)$, where we use a to abbreviate $a\,[1..1]$ and a^+ to abbreviate $a\,[1..*]$. Its syntax tree is reported in Figure 1.

Assume we read a word *bbac*. 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 $\mathbf{true}$. 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 $\mathbf{true}$. 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 $\mathbf{true}$, hence is not affected, and *CC*[1] is R^+, hence is 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 figure 2.

The algorithm now verifies that *Count*[] respects the cardinality constraints and that every A^+ node pushed into *ToCheck* has been

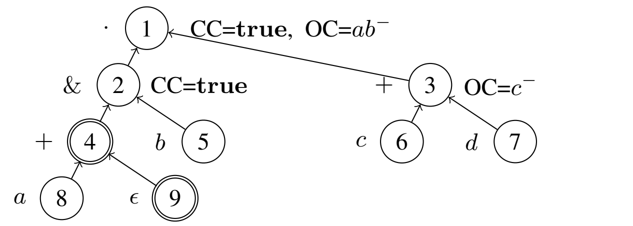


Figure 2. The tree for $T = ((a+\epsilon)\&b\,[1..5])\cdot(c+d^+)$ after word *bbac* has been read

residuated to $\mathbf{true}$; since both checks succeed, it returns *true*. If the word had been *bbacb*... 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. ∎

The constraint tree can be built in time $O(|T|)$. Moreover, for every symbol a in w, we only search and update the nodes which are ancestors of *NodeOfSymbol*$[a]$, and any such update has a constant cost, hence the resulting algorithm runs in $O(|T| + |w| * depth(T))$.

```
Member(w,T)
  (Min[],Max[],NodeOfSymbol[],Parent[],CC[],OC[],
    Nullable) := ReadType(T);
  SetToZero(Count[]);
  if (IsEmpty(w) and not Nullable) then return(false); fi;
  for a in w
  do  if (NodeOfSymbol[a] is null) then return(false); fi;
      Count[a] := Count[a]+1;
      for (n,direction) in Ancestors(NodeOfSymbol[a])
      do  case (CC[n], direction)
          when (⤇ or ⇔, left)
            then CC[n] := R+; push(n,ToCheck);
          when (⤆ or ⇔, right)
            then CC[n] := L+; push(n,ToCheck);
          when (⤆ or L+, left) or (⤇ or R+, right)
            then CC[n] := True;
          else ; esac;
          case (OC[n], direction)
          when (≺≻ or ≺, right)
            then OC[n] := L−;
          when (≺≻, left)
            then OC[n] := R−;
          when (L−, left) or (R−, right)
            then return(false);
          else ; esac;
      od;
  od;
  if (exists n in ToCheck with (CC[n] ≠ True))
      then return(false); fi;
  if (not CardinalityOK(Count[],Min[],Max[]))
      then return(false); fi;
  return(true);

Ancestors(n)
  if (Parent[n] is null) then return(emptylist);
  else return(Parent[n] ++ Ancestors([Parent[n]])); fi;
```

Figure 3. The basic residuation algorithm.

Theorem 3.3 (complexity) Member(w,T) *runs in time* $O(|T| + |w| * depth(T))$.

We can now prove that the algorithm is correct.

Theorem 3.4 (soundness) Member(w,T) *yields* true *iff* $w \in [\![T]\!]$.

Proof. A constraint F is affected by a symbol 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, our algorithm residuates all the nested constraints that are affected by every symbol of w. The final test ***exists*** *n* ***in*** *ToCheck* ***with*** *(CC[n] ≠ True)* verifies whether any A^+ constraint remains at the end, while every $\mathbf{false}$ residual causes the algorithm to immediately return *false*. For the flat constraints, the test in line 7 (**if** *NodeOfSymbol*$[a]$ *is null*) excludes that $w \not\models$ upperS(T). When $w \models$ upperS(T), we have that $w \not\models SIf(T)$ iff w is empty and N(w) is *false*; this is checked in line 5 (**if** *IsEmpty(w) and not Nullable*). We exclude $w \not\models$ ZeroMinMax(T) in line 30 — **if** *not CardinalityOK(Count[],Min[],Max[])*. ∎

4. The "Almost Linear" Version

The value of $depth(T)$ can be quite large, in practice, for example for types with many fields, such as $T_1 \cdot \ldots \cdot T_n$, or for types with many alternatives, such as $T_1 + \ldots + T_n$; in this last case, the type may be much larger than the word itself. This problem can be easily solved by flattening the constraints generated by such types, as follows ($SIf(T_1,\ldots,T_n)$ stands for $\{S^+(T_i) \mid \text{not } N(T_i)\}$; if $SIf(T_1,\ldots,T_n)=\emptyset$, then $\mathcal{CC}(T_1 \otimes \ldots \otimes T_n)$ is just $\mathbf{true}$).

$$\begin{array}{lcl} \mathcal{CC}(T_1 \otimes \ldots \otimes T_n) & = & (\cup_{i \in 1..n} S(T_i))^+ \\ & & \quad \Rightarrow SIf(T_1,\ldots,T_n) \\ \mathcal{OC}(T_1 \cdot \ldots \cdot T_n) & = & S(T_1) \prec \ldots \prec S(T_n) \\ \mathcal{OC}(T_1 + \ldots + T_n) & = & \prec\succ (S(T_1),\ldots,S(T_n)) \end{array}$$

Formally, we consider the three n-ary operators above in the syntax for types, and we generalize the syntax for constraints as specified below. The constraints ${A_1}^-,\ldots,{A_i}^-$ and $A_1 \prec \ldots \prec A_j$ are just two special cases of the constraint ${A_1}^-,\ldots,{A_i}^-, A_{i+1} \prec \ldots \prec A_{i+j}$, when $j=0$ or $i=0$ respectively; when both are zero, the constraint is written "$\mathbf{true}$". As usual, $\mathbf{false}$ abbreviates $(\emptyset)^+$. The syntactic forms below are defined under the condition that $i \geq 0$, $j \geq 0$, and $A_0 \supseteq (A_1 \cup \ldots \cup A_{i+1})$.

$$\begin{array}{rcl} F & ::= & {A_0}^+ \Rightarrow \{{A_1}^+,\ldots,{A_{i+1}}^+\} \\ & & \mid\ {A_1}^+,\ldots,{A_{i+1}}^+ \\ & & \mid\ \prec\succ (A_1,\ldots,A_{i+2}) \\ & & \mid\ {A_1}^-,\ldots,{A_i}^-, A_{i+1} \prec \ldots \prec A_{i+j} \end{array}$$

The semantics of these new constraints is fully defined by Table 3, plus the final conditions ${A_1}^+,\ldots,{A_n}^+ \ \to^{\epsilon} \mathbf{false}$ and $F \to^{\epsilon} \mathbf{true}$ otherwise.

We correspondingly refine the data structures of the residuation algorithm, as follows. Co-occurrence constraints are represented by an array *CC*[] of records with the following fields: *CC*$[n]$.*kind* $\in \{\Rightarrow, A^+, \mathbf{true}\}$, *CC*$[n]$.*needed*[], which is an array of booleans, 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 $S^+(T_1 \otimes T_2 \otimes T_3) \Rightarrow (S^+(T_2), S^+(T_3))$, and we represent it through a record *cc* with "*cc.kind*=($\Rightarrow$)", with *cc.neededCount=2*, and with a field *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".

The first time we meet any children i, we switch the kind of *cc* from ($\Rightarrow$) 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.

Condition	Constraint	Residual after a
$a \in A_i \subseteq A$	$A^+ \Rrightarrow \{A_1{}^+, \ldots, A_n{}^+\}$	$A_1{}^+, \ldots, A_{i-1}{}^+, A_{i+1}{}^+, \ldots, A_n{}^+$
$a \in A$	A^+	$\mathbf{true}$
$a \in A_i, n > 1$	$A_1{}^+, \ldots, A_n{}^+$	$A_1{}^+, \ldots, A_{i-1}{}^+, A_{i+1}{}^+, \ldots, A_n{}^+$
$a \in A_j$	$\prec\succ (A_1, \ldots, A_n)$	$A_1{}^-, \ldots, A_{j-1}{}^-, A_{j+1}{}^-, \ldots, A_n{}^-$
$a \in A_j, j \leq i$	$A_1{}^-, \ldots, A_i{}^-, A_{i+1} \prec \ldots \prec A_n$	**false**
$a \in A_j, j > i$	$A_1{}^-, \ldots, A_i{}^-, A_{i+1} \prec \ldots \prec A_n$	$A_1{}^-, \ldots, A_{j-1}{}^-, A_j \prec \ldots \prec A_n$

Table 3. Residuals of flattened constraints

Order constraints are represented by an array $OC[]$ of records with fields $kind \in \{^- \prec, \prec\succ, A^-, \mathbf{true}\}$, and $allowed \in \mathbb{N}$. If we have a type $T_1 \cdot \ldots \cdot T_m$, the corresponding oc record has *oc.kind*=($^- \prec$) and *oc.allowed*=1, which corresponds to a constraint $S(T_1) \prec \ldots \prec S(T_m)$. More generally, *oc.kind*=($^- \prec$) with *oc.allowed*=*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 $j < allowed$ we return **false**, and otherwise we just update *oc.allowed* to j.

Finally, a type $T_1 + \cdots + T_m$ is represented by oc with *oc.kind*=($\prec\succ$), which is residuated into *oc.kind*=(A^-) and *oc.allowed*=*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 $i' \neq i$.

To sum up, we represent these constraints, and their residuals, through the following two arrays; we assume that n is the node that corresponds to a type T whose children are $T_1, \ldots, T_m$.

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

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

    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[]: for each node n with m children, CC[n] is a record with two fields *kind* and *allowed*, whose meaning depends on the *kind* field, as follows:
  - $^- \prec$: this is a constraint

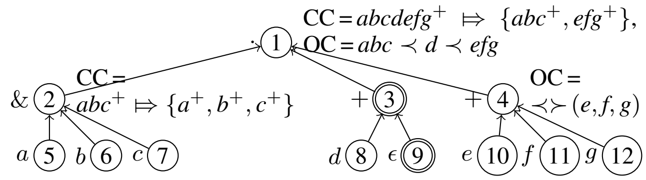
    $$A_1{}^-, \ldots, A_{i-1}{}^-, A_i \prec \ldots \prec A_m$$

    where $A_j = S(T_j)$, and i =*CC*[n].*allowed*.
  - $\prec\succ$: this is a constraint $A_1{}^-, \ldots, A_m{}^-$, 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 =*CC*[n].*allowed*.

We present the modified parts of the algorithm in Figure 5; a complete version, which includes all the optimizations that we are going to present, can be found in Appendix **??**.



Parent (pos)

1	-	-
2	1	1
3	1	2
4	1	3
5	2	1
...		

CC: kind, needed, nCount

1	$\Rrightarrow$	$[t, f, t]$	2
2	$\Rrightarrow$	$[t, t, t]$	3
3			
4			
5			
...			

OC: k., allowed

1	$^- \prec$	1
2		
3		
4	$\prec\succ$	-
5		
...		

Figure 4. Representation of $T = (a\&b\&c) \cdot (d^*) \cdot (e + f + g)$

Example 4.1 Consider the type $T = (a\&b\&c) \cdot (d^*) \cdot (e + f + g)$ (Figure 4), where we use a to abbreviate $a\,[1..1]$ and a^* to abbreviate $a\,[1..*] + \epsilon$.

The record *OC*[3] is null since only one child has a non-empty set of symbols. Assume we read a word *bcadddgdga*. When b is read, nodes 2 and 1 are visited, and their constraints are residuated to: *CC*[2]=(A^+,[t, f, t],2), *CC*[1]=(A^+,[f, f, t],1), *OC*[1]=($^- \prec$,1). The constraint *OC*[1] is actually unaffected, because both *allowed* and the child position *pos* are equal to 1. Both 2 and 1 are inserted in *ToCheck*, since they have now an A^+ kind. When c is read, *CC*[2] becomes (A^+,[t, f, f],1) and the constraints in node 1 are unaffected. When a is read, *CC*[2] becomes ($\mathbf{true}$,[f, f, f],0), since its *neededCount* is 0. When d is read, 3 and 1 are visited; *CC*[1] is unaffected, since *CC*[1][2] is already *false*, while *OC*[1] becomes ($^- \prec$,2), which means that symbols from the first subtree are now disallowed. 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), *CC*[1] becomes ($\mathbf{true}$, [f, f, f],0), and *OC*[1] becomes ($^- \prec$,3). If the word ended here, the algorithm would return *true*, since both nodes in *ToCheck* have now kind $\mathtt{true}$. But now a new d is read, which would residuate *OC*[1] to **false**, hence the algorithm stops with *false*. ■

This version of the algorithm is in $O(|T|+|w| * \mathit{flatdepth}(T))$, 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 (see (Bex et al. 2004)), hence this algorithm is "almost linear".

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

Theorem 4.3 (complexity) MemberFlat(w,T) *runs in time* $O(|T| + |w| * \mathit{flatdepth}(T))$.

```
MemberFlat(w,T)
  ...;
  for a in w
  do ...;
     for (n,pos) in Ancestors(NodeOfSymbol[a])
     do  case CC[n].kind
           when (⤇)   then  CC[n].kind := A+;
                            push(n,ToCheck);
                            ResiduatePlus(CC[n],pos);
           when (A+)  then  ResiduatePlus(CC[n],pos);
           else ; esac;
           case OC[n].kind
           when (⁻≺)  then  if (OC[n].allowed <= pos)
                            then OC[n].allowed := pos;
                            else  return(false);
                            fi;
           when (≺≻)  then  OC[n].kind := A−;
                            OC[n].allowed := pos;
           when (A−)  then  if (OC[n].allowed ≠ pos)
                            then return(false);
                            fi;
           else ; esac;
     od;
  od;
  if (exists n in ToCheck with (CC[n].kind ≠ True))
  then return(false); fi;
  ...;
  return(true);

ResiduatePlus(ccn,childPos)
  if (ccn.needed[childPos])
  then ccn.needed[childPos] := false;
       ccn.neededCount := ccn.neededCount-1;
       if (ccn.neededCount =0) then ccn.kind := True fi;
  fi
```

Figure 5. Checking with n-ary operators

5. The Linear Algorithm

We present here an orthogonal optimization, which makes the algorithm truly linear.

The base algorithm visits all the ancestors of *NodeOfSymbol*$[a]$ every time a is found in w, which is redundant. 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). 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, but will cause the algorithm to yield "false".

Formally, we have the following "A_2-stability" property. For any constraint F with shape $A_1{}^{+} \Mapsto A_2{}^{+}$, $A_1{}^{+} \Leftrightarrow A_2{}^{+}$, $A_1 \prec\succ A_2$ or $A_1 \prec A_2$, with A_1 disjoint from A_2, the following holds:

$$a \in A_2 \;\wedge\; F \xrightarrow{aw+} F' \;\;\Rightarrow\;\; \forall a' \in A_2\; F' \xrightarrow{a'} F'$$

The same property holds for $a \in A_1$ and $a' \in A_1$ for all the constraints but $A_1 \prec A_2$. Specifically, A_1-stability is only violated when a letter $b \in A_2$ is in w, as follows:

$$a \in A_1,\; b \in A_2,\; a' \in A_1:\;\; (A_1 \prec A_2) \xrightarrow{ab}{}^{+} (A_1{}^{-}) \xrightarrow{a'} \mathbf{false}$$

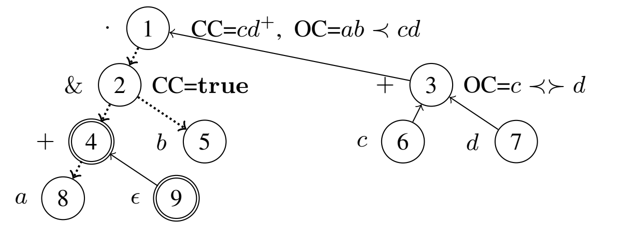


Figure 6. The tree for $T = ((a+\epsilon)\&b\,[1..5])\cdot(c+d^+)$ after word bba has been read; the dotted lines are those that are represented into the *Deleted* array.

The linear algorithm exploits this observation as follows:

1. whenever an upward pointer *Parent*[*child*], yielding *(parent,direction)*, is followed, the same pointer is set to null, so that it will not be traversed again; however, the node *child* is stored in *Deleted*[*parent,direction*];
2. to deal with the A_1-A_2-A_1 case of the $A_1 \prec A_2$ constraint, when a node *parent* marked with $\prec$ is reached from its right subtree, we use *Deleted*[*parent,left*] to recursively rebuild all the upward pointers in its left subtree; this is the only case when upward pointers are rebuilt.

Example 5.1 Consider the type $T = ((a+\epsilon)\&b\,[1..5])\cdot(c+d^+)$ (Figures 1, 6, and 7), and assume we read a word *bbac*. The first time we read b, when we visit the ancestor (*2,right*), we assign *Parent*[5]=*null* and *Deleted*[2,*right*]=5, so that now 5 has no parent, but the fact that the *right* child of 2 was 5 is recorded (the pointer from 5 to 2 is "reversed"). When we then visit (*1,left*), we also assign *Parent*[2]=*null* and *Deleted*[1,*left*]=2. The second time b is read, the parent of 5 is found to be null, hence no action is taken, apart from counting b. When a is read, its *UnvisitedAncerstors* list only contains (*2,left*), since the parent of 2 has been deleted. When node 2 is visited, we set *Parent*[4]=*null* and *Deleted*[2,*left*]=4. The situation is depicted in Figure 6.

Now, the arrival of any letter l from $\{a, b\}$ would only increment *Count*[l], but no ancestor would be visited. When c is read, both *Parent*[6] and *Parent*[3] are deleted, but, since *OC*[1]=$\prec$ and the direction is *right*, *Reactivate(1,left)* is invoked, which restores all pointers that are reachable from the *left* pointer of 1; the resulting tree is depicted in Figure 7.

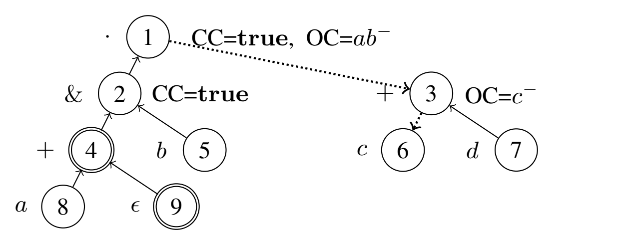


Figure 7. The tree for $T = ((a+\epsilon)\&b\,[1..5])\cdot(c+d^+)$ after word *bbac* has been read

The algorithm would now terminate with success. If a new a or b were read at this point, it would not be missed, but would cause the algorithm to stop with *false*. ■

No upward pointer can be deleted and rebuilt twice. Assume that an edge e has been deleted and rebuilt; this only happens if e is in the left subtree of a node n with constraint $A_1 \prec A_2$, and two symbols from A_1 and A_2 have been met, and hence $OC[n]$ is now

L^-. If the same e is deleted again, then the algorithm will return **false** as soon as the n ancestor of e is reached. Hence, any edge is traversed at most three times, to be deleted, rebuilt, and deleted for good. Similarly, the linear algorithm visits any internal node of the type at most three times, arriving twice from the left subtree and once from the right subtree. Hence, the algorithm has an $O(|T|)$ set-up cost, an $O(|w|)$ cost to access *NodeOfSymbol*$[a]$ for $|w|$ times, and a total residuation cost which is bound by $O(|T|)$, which gives a total of $O(|T| + |w|)$.

The initialization of the algorithm is identical to the non-optimized version, apart from the construction of the empty *Deleted*[] array. The body of the algorithm only changes when pointers are cut, and in the management of the *(≺, right)* case (Figure 8).

```
MemberLin(w,T)
  ...;
  for  a in w
  do   ...;
       for (nchild,(n,direction))
       in UnvisitedAncestors(NodeOfSymbol[a])
       do Parent[nchild]:=null;
          Deleted[n,direction] := nchild;
          case (CC[n], direction)
          ...;
          else ; esac;
          case (OC[n], direction)
          when (≺≻, right) then OC[n] := L⁻;
          when (≺, right)   then OC[n] := L⁻;
                                 Reactivate(n,left);
          when (≺≻, left)   then OC[n] := R⁻;
          when (L⁻, left) or (R⁻, right)
                            then return(false);
          else ; esac;
       od;
  od;
  ...;

Reactivate(parent,direction)
  if (Deleted[parent,direction] is not null)
  then child := Deleted[parent,direction];
       Deleted[parent,direction] := null;
       Parent[child] := (parent,direction);
       Reactivate(child,left); Reactivate(child,right);
  fi

UnvisitedAncestors(n)
  if (Parent[n] is null) then return(emptylist);
  else return((n,Parent[n])
              ++ UnvisitedAncestors([Parent[n]])); fi;
```

Figure 8. Version in $O(|w| + |T|)$

Theorem 5.2 (soundness) MemberLin(w,T) *yields* true *iff* $w \in [\![T]\!]$.

Theorem 5.3 (complexity) MemberLin(w,T) *runs in time* $O(|w| + |T|)$.

This optimization can be easily combined with flattening, obtaining a *MemberFlatLin* version, which would outperform *MemberFlat* in situations where we have long words with repeated characters, and would outperform *MemberLin* when types are "large", especially if the set-up phase can be shared by different runs of the algorithms, as discussed in the next section (see Appendix **??** for the code).

6. Multi-Words Checking

Before moving from words to trees, we must study the multi-words membership problem, the case where one type T is used to check m words $w_1, \ldots, w_m$. The repeated application of *MemberFlatLin* gives us an upper bound of $m*(|T|+|w|)$, where $|w|$ is the average length of the words. This bound is not linear, in general, in the input size $|T| + (m * |w|)$. In the special case when $|T| \leq |w|$, then $m * (|T| + |w|)$ is smaller than $2m * |w|$, hence the algorithm is indeed linear. In the general case, where $|T|$ may be much bigger than $|w|$, we get a better result if we avoid re-building the T structure from scratch after each word is checked. To this aim, we build the *Parent*[], *CC*[], *OC*[], etc., structures once, and we also build two copies *CCSave*[], *OCSave*[] of *CC*[] and *OC*[]. We then run a version *MultiFlatLin* of the *MemberFlatLin* algorithm with an undo-enabling line

```
Updated[n,direction] := nchild;
```

added immediately after the line

```
Deleted[n,direction] := nchild;
```

(See the Appendix for the whole code.) After each word is checked, we apply the following code to the root of the type, to restore *Parent*[], *CC*[], *OC*[], *Updated*[], *Deleted*[], and *Count*[] to their original state; we have first built a *Symbol*[] array that associates each $a?[m..n]$ node with its symbol a.

```
OC[root]:= OCSave[root]; CC[root]:= CCSave[root];
RestoreChild(root,left); RestoreChild(root,right);
```

where

```
RestoreChild(parent,direction)
  if (Updated[parent,direction] is not null)
  then child := Updated[parent,direction];
       Updated[parent,direction] := null;
       Deleted[parent,direction] := null;
       Parent[child] := (parent,direction);
       OC[child]:= OCSave[child];
       CC[child]:= CCSave[child];
       if (Symbol[child] is not null)
       then Count[Symbol[child]]:=0;
       else  RestoreChild(child,left);
             RestoreChild(child,right);
  fi;
fi
```

This *restoring* phase does not visit the whole T but only the modified part, hence is in $O(\min(|T|, |w| * \mathit{flatdepth}(T)))$, hence, after a set-up phase with cost $O(|T|)$, the cost of checking each word is in $O(\min(|w| + |T|, |w| * \mathit{flatdepth}(T)))$.

In the "easy case", when $|T|$ is smaller than $|w|$, each word is checked in $O(|w|)$, including the $O(|T|)$ time needed to set-up and restore the T structure, which gives the same linear complexity $O(m * |w|)$ as if T were rebuilt from scratch. In the "hard case", when $|T|$ is not bound by $|w|$, we at least know that this algorithms checks each word, and restores the structures, in time $O(|w_i| * \mathit{flatdepth}(T))$, giving a total complexity of $O(|T| + m * |w| * \mathit{flatdepth}(T))$. If we assume a constant upper-bound k for $\mathit{flatdepth}(T)$, the complexity is in $O(|T| + m * |w| * k)$, hence is still linear in the input size. Without the "restoring" optimization, the total cost would be $O(m * (|T| + |w|))$, which is much worse, since, in practice, we cannot reasonably assume an upper bound on either m or $|T|$.

7. Membership for 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 forests generated by the following grammar:

$$x ::= \epsilon \mid \langle a\rangle x\langle/a\rangle x$$

Following a long tradition, (see (Gelade et al. 2007), 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 (Thompson et al. 2004)). 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 *MultiFlatLin* 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 phase with cost $O(\min(|\tau(\alpha_i)|+|w_i|, |w_i|*\mathit{flatdepth}(\tau(\alpha_i))))$ for each $w_i \in [\![\tau(\alpha_i)]\!]$ test.[2] If the size of each w_i dominates the size of $\tau(\alpha_i)$, then the total cost is linear, by

$$\begin{aligned}
&\textstyle\sum_{i\in I} \min(|\tau(\alpha_i)| + |w_i|, |w_i| * \mathit{flatdepth}(\tau(\alpha_i)))\\
&\le \textstyle\sum_{i\in I}(|\tau(\alpha_i)| + |w_i|)\\
&\le 2\textstyle\sum_{i\in I} |w_i| \le 2|x|
\end{aligned}$$

Here we exploit the optimization described in Section 5. Observe that it is not true, in general, that $\sum_{i\in I} |\tau(\alpha_i)| \le |\tau|$ since the set I enumerates the elements inside x, not the components of τ.

This linear approximation does not hold when $\tau(\alpha_i)$ may be bigger than w_i, as happens in cases where complex content models are used to check documents where each element has a small number of children. In this case, which is quite common, we still have a quasi-linear complexity, by:

$$\begin{aligned}
&\textstyle\sum_{i\in I} \min(|\tau(\alpha_i)| + |w_i|, |w_i| * \mathit{flatdepth}(\tau(\alpha_i)))\\
&\le \textstyle\sum_{i\in I}(|w_i| * \mathit{flatdepth}(\tau(\alpha_i)))\\
&\le (\textstyle\sum_{i\in I} |w_i|) * \max(\mathit{flatdepth}(\tau(\alpha_i)))\\
&\le |x| * \max(\mathit{flatdepth}(\tau(\alpha_i)))
\end{aligned}$$

This upper bound is linear if we assume a constant upper bound k for $\mathit{flatdepth}(\tau(\alpha_i))$. Here we exploit the combined optimizations described in Section 4 and Section 6.

Since DTDs can be modeled as a special case of EDTDs, this quasi-linearity result holds for DTDs as well.

[2] 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

8. Intersection Types

8.1 Conflict-free Intersection Types

We left intersection out of our system because intersection is not present in XSD or in the DTD language, and also because it makes subtyping NP-hard, as proved in (Ghelli et al. 2007). However, intersection is extremely natural when types are seen as constraints, since it corresponds to constraint conjunction, and we prove here that it does not increase the complexity of our membership algorithm.

We first extend the syntax and semantics of types, and the definition of $\mathrm{N}(T)$, with intersection and empty types.

$$\begin{array}{lcl}
T ::= \ldots \mid \emptyset \mid T \cap T & & \\
\ldots & & \\
[\![\emptyset]\!] & = & \emptyset\\
[\![T_1 \cap T_2]\!] & = & [\![T_1]\!] \cap [\![T_2]\!]\\
\ldots & & \\
\mathrm{N}(\emptyset) & = & \text{false}\\
\mathrm{N}(T \cap T') & = & \mathrm{N}(T) \text{ and } \mathrm{N}(T')\\
\ldots & &
\end{array}$$

In this system, some types may be empty. Emptiness is CoNP-hard, even if we restrict ourselves to normal-form conflict-free types (defined later) where intersection is used just once (Ghelli et al. 2007). On the other side, nullability is still decidable in linear time, using the $\mathrm{N}(T)$ function.

Lemma 8.1 $\epsilon \in [\![T]\!]$ *iff* $\mathrm{N}(T)$.

Conflict-freedom interacts well with intersection types: any two types T_1 and T_2 can be combined through intersection, with no constraint about the relationship between $S(T_1)$ and $S(T_2)$. We call this notion *weak conflict-freedom*.

Definition 8.2 (Weakly-Conflict-free types) *A type T is* weakly-conflict-free *if for each subexpression $(U + V)$ or $(U \otimes V)$: $S(U) \cap S(V) = \emptyset$.*

Equivalently, a type T is weakly-conflict-free if, for any two distinct subterms $a\,[m..n]$ and $a\,[m'..n']$ that occur in T, their lowest common ancestor is a $\cap$ node.

Every weakly-conflict-free type can be transformed into a normal-form conflict-free type, defined as follows.

Definition 8.3 (Normal-form) *A type T is* normal-form conflict-free *if (a) for each subexpression $(U + V)$ or $(U \otimes V)$: $S(U) \cap S(V) = \emptyset$; (b) for each subexpression $(U \cap V)$: $\mathit{Atoms}(U) = \mathit{Atoms}(V)$; (c) $\emptyset$ is not a proper subtree of T.*

Observe that, in a normal-form conflict-free type T with j intersections, each symbol in $S(T)$ appears up to $j + 1$ times, and always with the same m-n bounds.

Informally, the normalization algorithm considers any $T \cap T'$ node that violates (b) because of a symbol a, and distinguishes two cases.

1. If a appears in T but not in T' (or vice versa), it is substituted with the empty type; the semantics of $T \cap T'$ does not change, since we have just removed all and only the words with a from $[\![T]\!]$, and these words do not belong to $[\![T \cap T']\!]$.
2. If $a\,[m..n]$ appears in T, $a\,[m'..n']$ appears in T', and $m'' = \max(m, m')$ and $n'' = \min(n, n'')$, then, if $m'' \le n''$, both $a\,[m..n]$ and $a\,[m'..n']$ are substituted with $a\,[m''..n'']$; if $m'' > n''$, they are both substituted with the empty type; also in this case, we are only deleting words, from T and T', which do not belong to the intersection.

The empty type is then easily eliminated using equivalences like $\emptyset \otimes T = \emptyset$ and $\emptyset + T = T$.

8.2 Constraint Extraction

Flat constraints only depends on $Atoms(T)$, and their definition is not affected by the addition of $\emptyset$ and $\cap$. In particular, since $\emptyset$ is not nullable, we have that $\mathit{SIf}(\emptyset) = S^{+}(\emptyset) = \emptyset^{+} = \mathbf{false}$, hence $\mathcal{FC}(\emptyset)$ is equivalent to **false**, as expected. Intersection nodes do not add any new nested constraint either.

Definition 8.4 (Nested constraints)

$$\begin{array}{llll} \mathcal{CC}(T_1 \cap T_2) & =_{def} & \mathcal{OC}(T_1 \cap T_2) & =_{def} \quad \mathbf{true} \\ \mathcal{CC}(\emptyset) & =_{def} & \mathcal{OC}(\emptyset) & =_{def} \quad \mathbf{true} \end{array}$$

The following theorem states that such constraints provide a sound and complete characterization of type semantics.

Theorem 8.5 *Given a (normal-form) conflict-free type T, it holds that:*

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

8.3 Membership checking

In a conflict-free type T with n intersections, each symbol in $S(T)$ appears many times, up to $n+1$, hence, instead of an array *NodeOfSymbol*[] of nodes, we have an array *NodesOfSymbol*[] of node lists. Whenever we meet an a in w, we check whether a is "to-do" (Figure 9), in which case we residuate all the ancestors of each node in *NodesOfSymbol*$[a]$. A symbol is "to-do" if it has never been seen before, or if it has been seen once and has been later re-activated, because it is in the A_1 set of a constraint $A_1 \prec A_2$ which has been transformed into L^{-}; this means that each symbol has to be examined at most twice. This is the only change to the *MultiFlatLin* algorithm. Observe that no constraint is associated with the internal nodes that correspond to $\cap$ in the type.

The algorithm needs one *to-do* test for each a in w plus, at most, a constant number of traversals and constant-time manipulations of the links and nodes of a $|T|$-size data structure, hence we have an upper bound $|T|$ for the set-up and an upper bound $O(|w|+|T|)$ for each word, hence the algorithm is linear for single-word membership. For the multi-word problem, the further bound $O(|w| * \mathit{flatdepth}(T))$, which holds for the *MultiFlatLin* algorithm, must be upgraded here to $O(|w| * \mathit{flatdepth}(T) * (\mathit{intnum}(T) + 1))$, where $\mathit{intnum}(T)$ is the number of intersections in T, because each symbol of w may occur at most $\mathit{intnum}(T)$ times in T, and for any such occurrence we have at most $\mathit{flatdepth}(T)$ constraints to residuate. This gives us a total complexity of $O(|T| + m * |w| * \mathit{flatdepth}(T) * (\mathit{intnum}(T) + 1))$. While we do not expect $\mathit{intnum}(T)$ ever to grow in practice, we have no experimental data to check, since the DTD and XSD languages do not support intersection.

```
IntMemberFlatLin(w,T)
  ...;
  for  a in w
  do   if (NodesOfSymbol[a] is null)
       then return(false); fi;
       Count[a] := Count[a]+1;
       if (ToDo[a])
       then ToDo[a]:=false;
            for  leaf in NodesOfSymbol[a]),
                 (nchild,(n,childPos))
                      in UnvisitedAncestors(leaf)
            do ... od;
       fi;
  od;
  ...;
```

Figure 9. Checking with intersection types

9. Conclusions

Membership checking is NP-hard for REs with interleaving. We have presented here a subclass of these REs which admits a simple polynomial membership algorithm. The algorithm is based on the transformation of the RE into a set of constraints, and on the parallel incremental *residuation* of these constraints. We have discussed the practical relevance of this class of extended REs, and have presented some optimizations that make our algorithm linear in the size of $|T| + |w|$. Apart from the practical motivations, we believe that it is important to understand how far the expressive power of REs can be extended with "hard" operators such as interleaving and counting before making membership NP-hard.

Our algorithm is not linear when used to check m words $\{w_i\}^{i \in 1..m}$ against one type T, since T appears once in the input, but it is visited m times by the algorithm. We have presented an optimization that makes the algorithm *almost* linear for repeated checking, that is, makes it linear in $|T| + (\sum_{i \in 1..m} |w_i|) * \mathit{flatdepth}(T)$, and $\mathit{flatdepth}(T)$ is very small in practice. Repeated checking is at the heart of XML membership checking with respect to DTDs and XSD schemas, hence the same quasi-linear complexity is preserved when we use our approach for XML membership checking. Finally, we showed how to extend the approach to REs with intersection.

References

Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. Efficient incremental validation of XML documents. In *ICDE*, pages 671–682. IEEE Computer Society, 2004.

Denilson Barbosa, Gregory Leighton, and Andrew Smith. Efficient incremental validation of XML documents after composite updates. In *XSym*, volume 4156 of *LNCS*, pages 107–121. Springer, 2006.

Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML schema: A practical study. In *WebDB*, pages 79–84, 2004.

Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise DTDs from XML data. In *VLDB*, pages 115–126, 2006.

Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML schema definitions from XML data. In *VLDB*, pages 998–1009, 2007.

J.A. Brzozowski. Derivates of regular expression. *Journal of the ACM*, 11: 481–494, 1964.

Byron Choi. What are real DTDs like? In *WebDB*, pages 43–48, 2002.

Wouter Gelade, Wim Martens, and Frank Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. In *ICDT*, 2007.

G. Ghelli, D. Colazzo, and C. Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. In *DBPL*, 2007. To appear.

Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2): 117–148, May 2003.

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

Alain J. Mayer and Larry J. Stockmeyer. Word problems — this time with interleaving. *Inf. Comput.*, 115(2):293–311, 1994.

Manizheh Montazerian, Peter T. Wood, and Seyed R. Mousavi. XPath query satisfiability is in PTIME for real-world DTDs. In *XSym*, volume 4704 of *LNCS*, pages 17–30. Springer, 2007.

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

Peter T. Wood. Containment for xpath fragments under DTD constraints. In *ICDT*, pages 300–314, 2003.