Theoretical Computer Science 492 (2013) 88–116

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Efficient asymmetric inclusion of regular expressions with interleaving and counting for XML type-checking[☆]

D. Colazzo[a], G. Ghelli[b], L. Pardini[b], C. Sartiani[c,*]

[a] Université Paris Sud - INRIA, UMR CNRS 8623, Orsay, F-91405, France

[b] Dipartimento di Informatica - Università di Pisa, Largo B. Pontecorvo 2 - Pisa, Italy

[c] Dipartimento di Matematica, Informatica ed Economia, Università della Basilicata, Via dell'Ateneo Lucano, 10, Potenza, Italy

ARTICLE INFO

Article history:
Received 2 February 2011
Received in revised form 30 January 2013
Accepted 19 April 2013
Communicated by W. Fan

Keywords:
XML
Regular expressions
Subtyping
XML schema

ABSTRACT

The inclusion of Regular Expressions (REs) is the kernel of any type-checking algorithm for XML manipulation languages. XML applications would benefit from the extension of REs with interleaving and counting, but this is not feasible in general, since inclusion is EXPSPACE-complete for such extended REs. In Colazzo et al. (2009) [1] we introduced a notion of "conflict-free REs", which are extended REs with excellent complexity behaviour, including a polynomial inclusion algorithm [1] and linear membership (Ghelli et al., 2008 [2]). Conflict-free REs have interleaving and counting, but the complexity is tamed by the "conflict-free" limitations, which have been found to be satisfied by the vast majority of the content models published on the Web.

However, a type-checking algorithm needs to compare machine-generated subtypes against human-defined supertypes. The conflict-free restriction, while quite harmless for the human-defined supertype, is far too restrictive for the subtype. We show here that the PTIME inclusion algorithm can be actually extended to deal with totally unrestricted REs with counting and interleaving in the subtype position, provided that the supertype is conflict-free.

This is exactly the expressive power that we need in order to use subtyping inside type-checking algorithms, and the cost of this generalized algorithm is only quadratic, which is as good as the best algorithm we have for the symmetric case (see [1]). The result is extremely surprising, since we had previously found that symmetric inclusion becomes NP-hard as soon as the candidate subtype is enriched with binary intersection, a generalization that looked much more innocent than what we achieve here.

1. Introduction

Different extensions of Regular Expressions (REs) with interleaving operators and counting are used to describe the content models of XML in the major XML type languages, such as XML Schema and RELAX-NG [4,5]. This fact raised new interest in the study of such extended REs, and, specifically, in the crucial problem of language inclusion. As pointed out by Mayer and Stockmeyer [6] and by Gelade et al. [7], the problem is EXPSPACE-complete. This prevents any practical use of unrestricted versions of regular expressions extended with interleaving and counting. However, in [1] we introduced a class of "conflict-free" REs with interleaving and counting, whose inclusion problem is in PTIME. The class is characterized

☆ An extended abstract of this paper was presented at the 12th International Conference on Database Theory (Colazzo et al., 2009 [3]). This full version contains full proofs, algorithms for the complete language considered – including the $T!$ constructor – and a report on experimental evaluation.

* Corresponding author. Tel.: +39 0971205862.
E-mail addresses: dario.colazzo@lri.fr (D. Colazzo), ghelli@di.unipi.it (G. Ghelli), pardini@di.unipi.it (L. Pardini), sartiani@gmail.com (C. Sartiani).

0304-3975/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.tcs.2013.04.023

by the single occurrence of each symbol and the limitation of Kleene-star and counting to symbols. Hence, an expression $a^*\& b^*$, denoting the interleaving of a sequence of a's and b's, is conflict-free, while $a \cdot b \cdot a$ and $(a \cdot b)^*$ are not. These very strict constraints have been repeatedly reported as being actually satisfied by the overwhelming majority of content models that are published on the Web,[1] which makes that result very promising, and of immediate applicability to the problem of comparing two different human-designed content models.

However, the main use of subtype-checking is in the context of *type-checking*, where *computed types* are checked for inclusion into *expected types*. This happens in the following cases (in each case we use U for the human-defined expected type, and E for the expression whose type is computed by the compiler):

(i) when a function, expecting a type U for its parameter, is applied to an expression E;
(ii) when the result of an expression E is used to update a variable, whose type U has been declared;
(iii) when the body E of a function is compared with the human-declared output type U of the function, in order to declare it type-correct.

In all these cases, the expected type is defined by a programmer, hence we can restrict it to a conflict-free type with little harm. However, the computed type reflects the structure of the code. Hence, the same symbol may appear in many different positions, and Kleene star may appear everywhere. In this situation, the ability to compare two conflict-free types is too limited, and we have to generalize it somehow. Consider, for instance, the following XQuery-like function.

```
function alpha($y : int*) as (a*& b*) {
         for $x in $y
         return if ($x ⩽ 0)
                then  a
                else  a, b, a
}
```

A type-checker would infer a type $(a + (a\cdot b\cdot a))^*$ for the body of this function, a type that corresponds to the structure of the code. This inferred type is not conflict-free, and must be compared for inclusion with the conflict-free declared output type $(a^* \& b^*)$.

Handling situations like this seemed very hard for a time. The result in [1] is based on an exact description of conflict-free types through *constraints*. These constraints are properties of the words in a language. For example, the language $a\cdot (b[3..5] + (c \cdot d))$ satisfies the following constraints (among others): "a occurs in each word" (occurrence constraint), "if c occurs, then d occurs" (co-occurrence constraints), "every a comes before any b" (order constraints), "if b occurs, it occurs between 3 and 5 times" (cardinality constraints) and "no other symbol but a, b, c, d may occur" (upper bound constraints). In [1] we proved that conflict-free types are exactly characterized by the set of such constraints that they satisfy, and we used this property to reduce type inclusion to *constraint implication*.

Unfortunately, even small generalizations of the conflict-free single-occurrence and Kleene-star limitations make types impossible to be exactly described by our constraints, as detailed later in Proposition 2.15. This problem does not arise if types are extended with intersection, since our constraints are closed by intersection. However, we showed in [1] that just one outermost use of binary intersection in the subtype makes inclusion NP-hard. As a consequence, our result seemed quite hard to generalize to the types that are met during type checking.

Our contribution. In this paper we show that we can generalize the result of [1] without leaving PTIME if we embrace asymmetry, and consider the *asymmetric inclusion problem*, i.e., the problem of verifying whether T is included in U, where T and U belong to two different families of extended REs. In this case, we find a surprisingly good result: inclusion is still in PTIME, provided that the supertype is conflict-free, even if no limit is imposed on the subtype, where interleaving, counting, and Kleene-star can be freely used. This means that a programmer must only declare conflict-free types, but the compiler can use the whole power of extended REs to approximate the result of any expression. The key for this result is understanding that, while the supertype has to be exactly described by the constraints, this is not necessary for the subtype.

To summarize, the main contributions of the present work are the following:

- we show that type inclusion can be reduced to constraint satisfaction if the constraint extraction function fully characterizes the supertype, for any subtype, even if the subtype can *not* be described by our constraint language;
- for each of the different kinds of constraints that our constraint language can express, we provide a polynomial algorithm to verify whether a generic type T, not necessarily conflict-free, satisfies that constraint;
- by combining the previous two contributions, we provide a quadratic algorithm to test whether T is included in U, where T and U are extended REs with interleaving and counting, provided that U is conflict-free, with no limitations on T;
- we provide an experimental evaluation, where we compare the performance of our quadratic algorithm with that of an asymmetric algorithm based on Brzozowski derivatives [11], which is our most direct competitor; this comparison shows that the quadratic algorithm scales very well, and is orders of magnitude faster than the competitor, in our experimental range.

[1] Bex et al. [8] found that the 99% of the 819 DTDs and XSDs that they examined belong to the class of chain regular expressions (CHAREs), which are less expressive than our conflict-free types; similar results, in the high range of 90%, have been reported by Barbosa et al. in [9] and by Choi et al. in [10].

Apart from the practical interest of a PTIME inclusion algorithm with no limitation on the subtype, this work also shows that the constraint approach is able to deliver interesting results in situations where traditional automata-based techniques are not easy to apply.

Paper outline. The paper is structured as follows. In Section 2 we introduce the type and constraint languages used here, and discuss the properties of constraint extraction functions. In Section 3 we introduce our inclusion algorithm, prove its correctness and completeness, and discuss its complexity. In Section 4 we experimentally analyze the performance of our algorithm. In Sections 5 and 6, finally, we review some related work and draw our conclusions.

2. Types and constraints

Following the terminology of [1], according to the intended application of these results to the type-checking of XML-based languages, we use the term "types" as a synonym for "extended regular expressions". Hence a "type" denotes a set of words. A *constraint* is a simple word property expressed in the constraint language we introduce below, and denotes the set of words that satisfy it. We say that a type T satisfies a constraint F when every word in T satisfies F, that is, when the denotation of T is included in that of F. Hence, every type is over-approximated by the set of all constraints that it satisfies. In [1] we introduced conflict-free types. For these types, this "approximation" is exact, meaning that a word belongs to a conflict-free type if and only if it satisfies all of its associated constraints.

Our algorithm is based on translating the supertype into a corresponding set of constraints and verifying, in polynomial time, that the subtype satisfies all of these constraints. In an asymmetric comparison, constraints provide an exact characterization for the conflict-free supertype, but just an over-approximation for the subtype; we will prove below that this does not affect the correctness or completeness of the algorithm.

In this paper we focus on the asymmetric inclusion of extended regular expressions. The results we propose can be lifted to *DTDs* and *Single-type Extended DTDs* (EDTDst) by following the approach described by Martens et al. in [12] and by Gelade et al. in [7]. In particular, the fact that the asymmetric inclusion is polynomial and that PTIME is a complexity class *closed under positive reductions* implies that the asymmetric inclusion of two schemas e_1 and e_2, where e_1 uses unrestricted regular expressions and e_2 uses conflict-free expressions, can be evaluated in polynomial time.

2.1. The type language

Our types denote sets of words over a finite alphabet Σ of symbols. We first define our notation for symbol extraction, symbol counting, and a notion of subword.

Definition 2.1 (*Word, $w(i)$, ϵ, length(w), $|w|_a$, Subword, Σ^*, Language*)**.** Given a set Σ, a *word* w over Σ is a function from $[1\ldots n]$ to Σ, where $[1\ldots n] \overset{def}{=} \{i \mid 1 \le i \le n\}$. When n is zero, then $[1\ldots n]$ is the empty set, and the only word over this empty domain is denoted by ϵ.

When $[1\ldots n]$ is the domain of w, we say that n is the length of w, and denote it with *length*(w).

For any $i \in [1\ldots \mathit{length}(w)]$, we say that $w(i)$ is the i-th symbol of w, or that $w(i)$ occurs in w at the i-th position. We use $|w|_a$ to indicate the number of positions of w where a occurs, that is, how many times a occurs in w.

When w' can be obtained by deleting some symbol occurrences from w, we say that w' is a *subword* of w. Formally, for any w of length m, and for any monotone injective $f : [1\ldots n] \to [1\ldots m]$, the word $w \circ f$, of length n, is a subword of w.

We use Σ^* to denote the set of all words over Σ. Any subset of Σ^* is a "language over Σ", or simply a "language".

We adopt the usual definitions for words 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.[2]

Definition 2.2 ($v\&w,\ L_1\&L_2$)**.** 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.

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

When $v \in w_1\&w_2$, we say that v is a shuffle of w_1 and w_2; for example, $w_1 \cdot w_2$ and $w_2 \cdot w_1$ are shuffles of w_1 and w_2, hence $[\![T_1\&T_2]\!] \supseteq [\![T_1 \cdot T_2]\!] \cup [\![T_2 \cdot T_1]\!]$.

We consider the following type language for words over an alphabet Σ:

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

[2] In [7] an alternative but equivalent definition of interleaving has been given.

where: $a \in \Sigma$, $m \in (\mathbb{N} \setminus \{0\})$, $n \in (\mathbb{N}_* \setminus \{0\})$, and $n \geq m$ (see Definition 2.4). The set $\mathbb{N}_*$ is $\mathbb{N} \cup \{*\}$, where $*$ behaves as $+\infty$, i.e., for any $n \in \mathbb{N}_*$, $* \geq n$.

The type ϵ is a singleton type that only contains the empty word ϵ. The type $T!$ denotes the set of T words minus ϵ. The type $T\,[m..n]$ denotes words that are formed by concatenating i words from T, with $m \leq i \leq n$ (Definition 2.5).

A type T is *well-formed* if it satisfies the numerical restrictions on $T\,[m..n]$ and if, for each subterm $T'!$, T' contains at least an a subterm for some $a \in \Sigma$ (Definition 2.4). Hereafter we always implicitly assume that every type we deal with is well-formed.

Definition 2.3 ($sym(w)$, $sym(T)$)**.** For any word w, $sym(w)$ is the range of w, that is, the set of all symbols in Σ appearing in w. For any type T, $sym(T)$ is the set of all symbols in Σ appearing in T.

Definition 2.4 (*Well-formed Type*)**.** A type T is well-formed if it satisfies the following conditions:

1. for any subterm $T'\,[m..n]$ of T, the following holds: $m \in (\mathbb{N} \setminus \{0\})$, $n \in (\mathbb{N}_* \setminus \{0\})$, $m \leq n$;
2. for any subterm $T'!$ of T, $sym(T') \neq \emptyset$.

Note that expressions like $T\,[0..n]$ are not allowed, but the type $T\,[0..n]$ can be equivalently represented by $T\,[1..n] + \epsilon$. The mandatory presence of an a subterm in $T!$ guarantees that T contains at least one word that is different from ϵ, hence $T!$ is never empty (Lemma 2.8), which, in turn, implies that we have no empty types.

The semantics of types is inductively defined by the following equations.

Definition 2.5 ($[\![T]\!]$)**.**

$$\begin{aligned}
[\![\epsilon]\!] &\overset{def}{=} \{\epsilon\}\\
[\![a]\!] &\overset{def}{=} \{a\}\\
[\![T_1 + T_2]\!] &\overset{def}{=} [\![T_1]\!] \cup [\![T_2]\!]\\
[\![T_1 \cdot T_2]\!] &\overset{def}{=} [\![T_1]\!] \cdot [\![T_2]\!]\\
[\![T_1 \& T_2]\!] &\overset{def}{=} [\![T_1]\!] \& [\![T_2]\!]\\
[\![T!]\!] &\overset{def}{=} [\![T]\!] \setminus \{\epsilon\}\\
[\![T\,[m..n]]\!] &\overset{def}{=} \{w \mid w = w_1 \cdot \ldots \cdot w_j,\ \forall i \in 1..j.\ w_i \in [\![T]\!], m \leq j \leq n\}
\end{aligned}$$

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

Types that contain the empty word ϵ are called *nullable* and are characterized as follows. Observe that $\mathrm{N}(T\,[m..n]) = \mathrm{N}(T)$ because m cannot be 0.

Definition 2.6. $\mathrm{N}(T)$ is a predicate on types, defined as follows:

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

Property 2.7 ($\mathrm{N}(T)$)**.**

$$\epsilon \in [\![T]\!] \Leftrightarrow \mathrm{N}(T)$$

In this system, no type is empty, and any symbol in $sym(T)$ appears in some word of $[\![T]\!]$.

Lemma 2.8. *For any type T:*

$$\begin{aligned}
&[\![T]\!] \neq \emptyset && (1)\\
&a \in sym(T) \Leftrightarrow \exists w \in [\![T]\!].a \in sym(w) && (2)
\end{aligned}$$

Proof. Trivial. $\square$

$$\begin{array}{rcl}
w \models F & \overset{def}{\Leftrightarrow} & w \in [\![F]\!] \\
[\![A^+]\!] & \overset{def}{=} & \{w \in \Sigma^* \mid sym(w) \cap A \neq \emptyset\} \\
[\![A^+ \Rrightarrow B^+]\!] & \overset{def}{=} & \{w \in \Sigma^* \mid w \notin [\![A^+]\!] \vee w \in [\![B^+]\!]\} \\
[\![a?[m..n]]\!] & \overset{def}{=} & \{w \in \Sigma^* \mid \ m \leq |w|_a \leq n \ \vee \ |w|_a = 0\} \\
[\![\mathrm{upper}(A)]\!] & \overset{def}{=} & \{w \in \Sigma^* \mid sym(w) \subseteq A\} \\
[\![a \prec b]\!] & \overset{def}{=} & \{w \in \Sigma^* \mid \forall i, j \in [1..length(w)].\ (w(i) = a \wedge w(j) = b) \Rightarrow i < j\}
\end{array}$$

Fig. 1. Constraint semantics.

In the following we will use RE$^+$(#, &) to denote this class of regular expressions.

2.2. *Constraints*

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

$$F ::= A^+ \mid A^+ \Rrightarrow B^+ \mid a?[m..n] \mid \mathrm{upper}(A) \mid a \prec b$$

We do not explicitly consider conjunctive constraints $F \wedge F'$ since we will always associate types with *sets* of constraints, whose conjunction the type has to satisfy. The semantics of constraints is defined in Fig. 1.

The following special cases are worth noticing.

$$\epsilon \models a?[m..n] \qquad \epsilon \models a \prec b \qquad b \models a \prec b \qquad aba \not\models a \prec b$$

Observe that A^+ is monotone with respect to the subword order, i.e., $w \models A^+$ and w is a subword of w' imply that $w' \models A^+$. The constraint $a \prec b$, instead, is anti-monotone, in the sense that, if $w \models a \prec b$ and w' is a subword of w, then $w' \models a \prec b$. The constraint upper(A) is anti-monotone as well.

As pointed out by the first equation of Fig. 1, a constraint F denotes the set of words that satisfy it; the following definition states that a set of constraints $\mathcal{F}$ denotes the words that satisfy each $F \in \mathcal{F}$.

Definition 2.9 ($[\![\mathcal{F}]\!]$)**.** For set of constraints $\mathcal{F}$:

$$[\![\mathcal{F}]\!] \overset{def}{=} \bigcap_{F \in \mathcal{F}} [\![F]\!]$$

A type satisfies a constraint if all of its words do. The previous definition allows us to express this as set inclusion.

Definition 2.10 ($\mathcal{W} \models F, T \models F, T \models \mathcal{F}$)**.** For any set of words $\mathcal{W}$, type T, constraint F, and set of constraints $\mathcal{F}$:

$$\mathcal{W} \models F \overset{def}{\Leftrightarrow} \mathcal{W} \subseteq [\![F]\!] \qquad T \models F \overset{def}{\Leftrightarrow} [\![T]\!] \subseteq [\![F]\!] \qquad T \models \mathcal{F} \overset{def}{\Leftrightarrow} [\![T]\!] \subseteq [\![\mathcal{F}]\!]$$

2.3. *Conflict-free types, constraints and subtyping*

In [1] we introduced the following class of conflict-free types (hereafter we will use the meta-variable U for conflict-free types).

Definition 2.11 (*Conflict-free Type*)**.** A type U is conflict-free iff:

- no symbol appears twice in U, that is, for any subterm $U_1 \circledast U_2$ of U, $sym(U_1) \cap sym(U_2) = \emptyset$.
- counting is only applied to symbols, that is, for any subterm $U'[m..n]$ of U, U' is a symbol $a \in \Sigma$.

The symbol-counting restriction means that, for example, types like $(a \cdot b)^*$ cannot be expressed. However, it has been found that DTDs and XSD schemas[3] use repetition almost exclusively as a^{op} or as $(a + \cdots + z)^{\mathrm{op}}$ (where op $\in \{+, *\}$, see [8]), which can be immediately translated to types that only count symbols, thanks to the $U_1 \& U_2$ and $U!$ operators. For instance, $(a + \cdots + z)^*$ can be expressed as $(a^* \& \ldots \& z^*)$, where a^* is a shortcut for $a\,[1..*] + \epsilon$, while $(a + \cdots + z)^+$ can be expressed as $(a^* \& \ldots \& z^*)!$.

In [1] we also defined an algorithm to check inclusion of conflict-free types, based on the existence of an exact constraint extraction function for conflict-free types, where exactness is defined in Definition 2.12, together with two weaker properties, soundness and completeness, that we will need soon. The notion of *completeness* is relative to a constraint language; for our aims, we can define a constraint language to be any set of sets of constraints. For example, the set of all sets of constraints that have shape $\{a_1 \prec b_1, \ldots, a_j \prec b_j\}$ is a constraint language, "the language of order constraints".

[3] An XSD schema is an XML schema written in the W3C XML Schema Definition Language [4]; this language is often called "XML Schema" or XSD.

Definition 2.12 (*Soundness, Completeness, Exactness of $\mathcal{F}_T$*)**.** Consider a constraint language $\mathscr{F}$, a type T, and a set of constraints $\mathcal{F}_T \in \mathscr{F}$. We define three properties that $\mathcal{F}_T$ may satisfy for T and $\mathscr{F}$:

- soundness: $\mathcal{F}_T$ is sound for T if $T \models \mathcal{F}_T$, that is, $[\![T]\!] \subseteq [\![\mathcal{F}_T]\!]$;
- $\mathscr{F}$-completeness: a sound $\mathcal{F}_T$ is complete for $\mathscr{F}$ and T if $[\![\mathcal{F}_T]\!] = [\![\{F \in \mathscr{F} \mid T \models F\}]\!]$, that is, $\mathcal{F}_T$ is the most precise description of T that can be expressed through the constraint language $\mathscr{F}$;
- exactness: $\mathcal{F}_T$ is exact for T if $[\![T]\!] = [\![\mathcal{F}_T]\!]$.

A function $\mathcal{C}$ mapping types to sets of constraints is called sound/$\mathscr{F}$-complete/exact, if $\mathcal{C}(T)$ is, respectively, sound, $\mathscr{F}$-complete, or exact, for any T.

In short, $\mathcal{F}_T$ is sound if it over-approximates T, is $\mathscr{F}$-complete if is sound and cannot be made more precise by adding any more constraints from $\mathscr{F}$, and is exact if its semantics coincides with $[\![T]\!]$, which implies that it is also complete and sound. When a type admits an exact constraint set, we say that the type is constraint-expressible.

Definition 2.13 (*Constraint-expressible Type*)**.** A type T is constraint-expressible, with respect to a constraint language $\mathscr{F}$, if there exists a set of constraints $\mathcal{F}_T \in \mathscr{F}$ such that $[\![T]\!] = [\![\mathcal{F}_T]\!]$, that is, such that $\mathcal{F}_T$ is exact for T. If no such set exists, we say that T is constraint-inexpressible in $\mathscr{F}$.

When we say that a type T is constraint-expressible, or inexpressible, with no explicit reference to a specific $\mathscr{F}$, we are implicitly referring to the constraint language that we define in this paper.

The algorithm defined in [1] is based on the following result, proved in the same paper.

Theorem 2.14. *There exists an exact constraint extraction function for conflict-free types. That is, every conflict-free type is constraint-expressible.*

The proof of [1] is constructive, since we actually define a constraint extraction function $\mathcal{C}(U)$ satisfying $\forall U : [\![U]\!] = [\![\mathcal{C}(U)]\!]$.

The subtyping algorithm of [1] is based on our ability to exactly characterize conflict-free types through sets of constraints. Unfortunately, conflict-free types, while adequate to express human-defined types, are too weak to capture compile-inferred types. One would hence like to loosen the conflict-free restrictions, in order to enlarge the set of types that we can manipulate. Unfortunately, any small loosening that we considered produces a set of types that includes some constraint-inexpressible type.

For example, and without any pretence of completeness, one may first consider loosening the single-occurrence restriction for $U_1 + U_2$, $U_1 \cdot U_2$ or $U_1 \& U_2$, so that, in some of these three cases, one may have that $sym(U_1) \cap sym(U_2)$ is not empty. One may also consider loosening the counting restriction, so that counting could be applied to a term U_1 that is not just one symbol, but is built using one of the three binary operators, or the counting operator.

We now show that each of the seven possibilities above would allow the enlarged type-language to express some constraint-inexpressible types, already in situations when U_1 and U_2 are extremely simple.

Proposition 2.15. *Each of the following seven types, corresponding to different ways of loosening the restrictions that define conflict-free types, is constraint-inexpressible.*

Loosening single-occurrence restriction	
Loosened restriction	Constraint-inexpressible type
Allowing $a \in sym(U_1) \cap sym(U_2)$ in a subterm $U_1 + U_2$	$(a \cdot b) + (b \cdot a \cdot c)$
Allowing $a \in sym(U_1) \cap sym(U_2)$ in a subterm $U_1 \cdot U_2$	$a \cdot (b \cdot a)$
Allowing $a \in sym(U_1) \cap sym(U_2)$ in a subterm $U_1 \& U_2$	$a \& (b \cdot a)$

Loosening counting restriction	
Loosened restriction	Constraint-inexpressible type
Allowing $U\,[m..n]$ under counting	$a\,[1..2]\,[1..2]$
Allowing $U_1 + U_2$ under counting	$(a + b)\,[2..2]$
Allowing $U_1 \cdot U_2$ under counting	$(a \cdot b)\,[2..2]$
Allowing $U_1 \& U_2$ under counting	$(a \& b)\,[1..2]$

Proof. In Appendix. □

The intuition that allowed us to solve this problem is illustrated by the following proposition. The proposition shows that we do not need an exact characterization for both compared types; we only need exactness for the right-hand side.

Proposition 2.16 (*Asymmetric Subtyping*)**.** *If $\mathcal{C}$ is exact for U,*[4] *then:*

$$[\![T]\!] \subseteq [\![U]\!] \Leftrightarrow T \models \mathcal{C}(U)$$

[4] We use the letter U since we apply this proposition to conflict-free types only, but it actually holds for any type U that is exactly described by $\mathcal{C}(U)$.

Proof. $T \models \mathcal{C}(U) \Leftrightarrow$ (by definition) $[\![T]\!] \subseteq [\![\mathcal{C}(U)]\!] \Leftrightarrow$ (by exactness) $[\![T]\!] \subseteq [\![U]\!]$. $\square$

This observation provides a way to generalize our previous results that is very interesting: rather than looking for generalizations of the conflict-free family in the narrow precinct of those types that can be exactly described, we can aim for the whole set of extended REs in the left-hand side of $[\![T']\!] \subseteq [\![T'']\!]$, if we stay modest with the right-hand side.

To exploit this observation, we need now to extend the exact constraint-extraction $\mathcal{C}(U)$ of [1] with a procedure to test for $T \models \mathcal{C}(U)$.

In [1] we provided a quadratic algorithm for the case when T is conflict-free, while we proved that the problem is NP-hard when T ranges over conflict-free types with intersection. We are going to give here a quadratic procedure when T ranges over general types (with no intersection, of course).

3. Inclusion algorithm

In [1], we defined a constraint-extraction function that is exact for conflict-free types. For each type, this function extracts five classes of constraints: *co-occurrence* constraints $\mathcal{CC}(U)$, *order* constraints $\mathcal{OC}(U)$, *cardinality* constraints $ZeroMinMax(U)$, *lower-bound* constraints $SIf(U)$, and *upper-bound* constraints $upperS(U)$, that is, the exact function that we are going to use is defined as

$$\mathcal{C}(U) = \mathcal{CC}(U) \cup \mathcal{OC}(U) \cup ZeroMinMax(U) \cup SIf(U) \cup upperS(U)$$

To apply Proposition 2.16, we now have to exhibit, for each component $\mathcal{C}_i(U)$ (where $\mathcal{C}_i(U)$ is one of $\mathcal{CC}(U)$, $\mathcal{OC}(U)$, etc.), an algorithm to verify whether, for each $F \in \mathcal{C}_i(U)$, $T \models F$, where T is a general type. This will be done in the following sections. In each section we will recall the definition of the corresponding component of $\mathcal{C}(U)$. The last two components $SIf(U)$ and $upperS(U)$ will be dealt with together.

3.1. Co-occurrence constraints $\mathcal{CC}(U)$

Overview

In this section we present an algorithm to verify, for each T and U, that $T \models A^+ \Rrightarrow B^+$ for each $A^+ \Rrightarrow B^+ \in \mathcal{CC}(U)$, in time $O(n^2)$, where $n = |T| + |U|$. We first recall the definition of $\mathcal{CC}(U)$ from [1]. For each type U, the set $\mathcal{CC}(U)$ contains $O(n)$ constraints with shape $A^+ \Rrightarrow B^+$, and we will present an algorithm to verify $T \models A^+ \Rrightarrow B^+$, using time $O(n)$ for each constraint $A^+ \Rrightarrow B^+$.

To this aim we define a new auxiliary constraint $B^{+?}$, denoting the set of all words in $[\![B^+]\!] \cup \{\epsilon\}$, and we prove that $T \models A^+ \Rrightarrow B^+$ holds iff every occurrence of every symbol of A in T is included in a subterm T' of T such that $T' \models B^{+?}$ (Theorem 3.6). This is the key technical result of this section.

We then define an algorithm that, given B and T, finds all subterms T' of T such that $T' \models B^{+?}$, and marks all occurrences of symbols of T that are included in at least one of these T' subterms, in linear time. Now, in time $O(|T|)$, we can verify whether every occurrence of any symbol of A in T has been marked, which, by the previous result, is equivalent to verifying that $T \models A^+ \Rrightarrow B^+$.

Constraints extraction $\mathcal{CC}(U)$

The first component $\mathcal{CC}(U)$ of $\mathcal{C}(U)$ extracts a set of co-occurrence constraints with shape $A^+ \Rrightarrow B^+$, and is defined, in [1], as follows, where $\{F \mid \neg \mathrm{N}(U)\}$ denotes the singleton $\{F\}$ when $\mathrm{N}(U)$ is false, and denotes the empty set otherwise. Observe that $\mathcal{CC}(U)$ contains at most two constraints for each product node of U.

$$\begin{aligned}
\mathcal{CC}(U_1 \otimes U_2) &\overset{def}{=} \{sym(U_1)^+ \Rrightarrow sym(U_2)^+ \mid \neg \mathrm{N}(U_2)\} \\
&\quad \cup \{sym(U_2)^+ \Rrightarrow sym(U_1)^+ \mid \neg \mathrm{N}(U_1)\} \\
&\quad \cup \mathcal{CC}(U_1) \cup \mathcal{CC}(U_2) \\
\mathcal{CC}(U_1 + U_2) &\overset{def}{=} \mathcal{CC}(U_1) \cup \mathcal{CC}(U_2) \\
\mathcal{CC}(U!) &\overset{def}{=} \mathcal{CC}(U) \\
\mathcal{CC}(\epsilon) &\overset{def}{=} \emptyset \\
\mathcal{CC}(a[m..n]) &\overset{def}{=} \emptyset \\
\mathcal{CC}(a) &\overset{def}{=} \emptyset
\end{aligned}$$

Soundness of $\mathcal{CC}(U)$ is a consequence of the single occurrence of symbols in U. Consider for example the rule for union and a type $U + a \cdot b$. The set $\mathcal{CC}(U + (a \cdot b))$ includes $\mathcal{CC}(a \cdot b)$, that is $a^+ \Rrightarrow b^+$ and $b^+ \Rrightarrow a^+$, because every occurrence of an a in a word $w \in [\![U + (a \cdot b)]\!]$ implies that w belongs to $[\![a \cdot b]\!]$, since a cannot appear in U, because of single occurrence. A type $a \cdot b + (a \cdot c)$, which violates single occurrence, would *not* satisfy the conjunction of $a^+ \Rrightarrow b^+$ and $a^+ \Rrightarrow c^+$, but only their disjunction.

Example 3.1.

- $\mathcal{CC}(a[1..2]\cdot(b[2..*]+c[1..*]+\epsilon)) = \{bc^+ \mapsto\!\!\!\Rightarrow a^+\}$
  $a^+ \mapsto\!\!\!\Rightarrow bc^+$ is not in $\mathcal{CC}(a[1..2]\cdot(b[2..*]+c[1..*]+\epsilon))$ because $(b[2..*]+c[1..*]+\epsilon)$ is nullable;
- $\mathcal{CC}(a[1..1]\cdot(b[2..2]\cdot c[3..3])) = \{a^+ \mapsto\!\!\!\Rightarrow bc^+,\ bc^+ \mapsto\!\!\!\Rightarrow a^+,\ b^+ \mapsto\!\!\!\Rightarrow c^+,\ c^+ \mapsto\!\!\!\Rightarrow b^+\}$.

Formal treatment

We first observe that all symbols at the left-hand side of a co-occurrence constraint can be dealt with one at a time; this is an immediate consequence of the definition of $A^+ \mapsto\!\!\!\Rightarrow B^+$.

Property 3.2 (*Union*). *For any word w and constraint $A^+ \mapsto\!\!\!\Rightarrow B^+$:*

$$w \models A^+ \mapsto\!\!\!\Rightarrow B^+ \Leftrightarrow \forall a \in A.\ w \models a^+ \mapsto\!\!\!\Rightarrow B^+$$

We now introduce an auxiliary constraint $A^{+?}$, whose semantics is defined as follows.

Definition 3.3 ($A^{+?}$).

$$[\![A^{+?}]\!] \stackrel{def}{=} [\![A^+]\!] \cup \{\epsilon\}$$

The constraint $A^{+?}$ satisfies the following properties.

Property 3.4 ($T \models A^{+?}$ *and* $T \models A^+$). *For any type T, and set $A \subseteq \Sigma$, the following properties hold.*

$$T \models A^{+?} \Leftrightarrow \forall w \in [\![T]\!].\ (w = \epsilon \ \vee\ (sym(w) \cap A) \neq \emptyset) \quad (1)$$
$$T \models A^{+?} \Leftrightarrow T! \models A^+ \quad (2)$$
$$T \models A^{+} \Leftrightarrow T \models A^{+?} \wedge \neg \mathrm{N}(T) \quad (3)$$

We are now ready for the main theorem, relating $A^+ \mapsto\!\!\!\Rightarrow B^+$ to $B^{+?}$. The theorem specifies that, for any constraint $a^+ \mapsto\!\!\!\Rightarrow B^+$ with $a \notin B$, the constraint holds in T iff, for every occurrence of a in T, there exists a product subterm T' of T that contains that instance and such that $T' \models B^{+?}$. This subterm may be proper or may be T itself. For example, consider $T' = a[1..1]\cdot(b[2..2]\&a[3..3])$ and $T = c + T'$. The product type T' satisfies $b^{+?}$ (any word of $[\![T']\!]$ contains b), and it contains both occurrences of a inside T, hence $T \models a^+ \mapsto\!\!\!\Rightarrow b^+$, and the same would hold for a type $T'' = c\&T'$. If we consider $T''' = a+T'$, however, the first occurrence of a is not included in any product type, hence this type does not satisfy $a^+ \mapsto\!\!\!\Rightarrow b^+$; indeed, the word $a \in [\![T''']\!]$ violates this constraint.

We first introduce a crucial lemma, then we prove the theorem.

Lemma 3.5. *For any type $T_1 \otimes T_2$:*

$$T_1 \otimes T_2 \models a^+ \mapsto\!\!\!\Rightarrow A^+ \Rightarrow (T_1 \models a^+ \mapsto\!\!\!\Rightarrow A^+ \wedge T_2 \models a^+ \mapsto\!\!\!\Rightarrow A^+) \vee T_1 \otimes T_2 \models A^{+?}$$

Proof. We prove the following, equivalent, proposition:

$$(T_1 \otimes T_2 \models a^+ \mapsto\!\!\!\Rightarrow A^+ \wedge T_1 \otimes T_2 \not\models A^{+?}) \Rightarrow (T_1 \models a^+ \mapsto\!\!\!\Rightarrow A^+ \wedge T_2 \models a^+ \mapsto\!\!\!\Rightarrow A^+)$$

By $T_1 \otimes T_2 \not\models A^{+?}$, there is at least a non-empty word $w \in [\![T_1 \otimes T_2]\!]$ containing no A symbol. This implies that there exist two words $w_1 \in [\![T_1]\!]$ and $w_2 \in [\![T_2]\!]$ containing no A symbol, and such that $w = w_1 \cdot w_2$ if $\otimes = \cdot$, and $w \in w_1 \& w_2$ if $\otimes = \&$. We have now to prove that, for any w'_1:

$$w'_1 \in [\![T_1]\!] \wedge w'_1 \models a^+ \Rightarrow w'_1 \models A^+$$

Consider any $w'_1 \in [\![T_1]\!]$ with $w'_1 \models a^+$. We have that $w'_1 \cdot w_2 \in [\![T_1 \otimes T_2]\!]$, which by hypothesis implies $w'_1 \cdot w_2 \models a^+ \mapsto\!\!\!\Rightarrow A^+$, hence $w'_1 \cdot w_2 \models A^+$, hence $w'_1 \models A^+$, since $w_2 \not\models A^+$. We prove $T_2 \models a^+ \mapsto\!\!\!\Rightarrow A^+$ in the same way. □

Theorem 3.6 ($T \models a^+ \mapsto\!\!\!\Rightarrow B^+$ *from* $T' \models B^{+?}$). *For any type T, any $B \subseteq \Sigma$, any $a \in (\Sigma \setminus B)$, the following sentences are equivalent.*

1. *$T \models a^+ \mapsto\!\!\!\Rightarrow B^+$;*
2. *for each occurrence of a inside T, there exists a subterm T' of T such that all the following properties hold:*
   - *T' includes the occurrence of a;*
   - *$T' \models B^{+?}$;*
   - *T' is a product type, that is, there exist T_1 and T_2 such that $T' = T_1 \otimes T_2$.*

Proof. (1) ⇒ (2). Assume $T \models a^+ \mapsto\!\!\!\Rightarrow B^+$. We prove the thesis by induction and by cases on the shape of T. We omit obvious cases.

$T = b \neq a$ and $T = \epsilon$. a does not occur inside T, hence the thesis holds trivially.
$T = a$. This case is impossible, since $a \models a^+ \mapsto\!\!\!\Rightarrow B^+$ would imply $a \models B^+$, that contradicts the hypothesis $a \in (\Sigma \setminus B)$.
$T = T_1 + T_2$. Hence, $T_1 \models a^+ \mapsto\!\!\!\Rightarrow B^+$ and $T_2 \models a^+ \mapsto\!\!\!\Rightarrow B^+$. By induction, each occurrence of a subterm a in T_1 and in T_2 is part of a product T' with $T' \models B^{+?}$, as required.

$T = T_1 \otimes T_2$. By Lemma 3.5, either $T_1 \models a^+ \Rrightarrow B^+$ and $T_2 \models a^+ \Rrightarrow B^+$, or $T_1 \otimes T_2 \models B^{+?}$. In the first case, by induction, each occurrence of a subterm a in T_1 and in T_2 is part of a product T' with $T' \models B^{+?}$, as required. In the second case, T itself is the product subterm with $T' \models B^{+?}$.

(2) $\Rightarrow$ (1). Assume that, for each occurrence of a inside T, the occurrence is part of a product subterm T' of T such that $T' \models B^{+?}$. We want to prove that $T \models a^+ \Rrightarrow B^+$.

We reason by induction and by cases. We omit obvious cases.

$T = b \neq a$ and $T = \epsilon$. In both cases, $T \models a^+ \Rrightarrow B^+$ holds trivially.

$T = a$. This case is excluded by the hypothesis, because we have an occurrence of a but no product inside T.

$T_1 + T_2$. Each of T_1 and T_2 satisfies the theorem hypothesis, hence, by induction, each of them satisfies $T_i \models a^+ \Rrightarrow B^+$, hence $T \models a^+ \Rrightarrow B^+$.

$T = T_1 \otimes T_2$. We consider two cases, either $T_1 \otimes T_2 \models B^{+?}$ or $T_1 \otimes T_2 \not\models B^{+?}$. If $T_1 \otimes T_2 \models B^{+?}$, by definition $T_1 \otimes T_2 \models a^+ \Rrightarrow B^+$, hence we are done. If $T_1 \otimes T_2 \not\models B^{+?}$, we show that both T_1 and T_2 satisfy the theorem hypothesis, and proceed by induction. To this aim, consider an occurrence of a that is in T_1. By hypothesis, that occurrence is included in a product subterm T' of T that satisfies $B^{+?}$. Since $T_1 \otimes T_2 \not\models B^{+?}$, then T' is inside T_1, hence, T_1 satisfies the theorem hypothesis. Hence, by induction, $T_1 \models a^+ \Rrightarrow B^+$. In the same way we prove $T_2 \models a^+ \Rrightarrow B^+$. Now consider any word $w \in [\![T_1 \otimes T_2]\!]$, so that $w \in w_1 \& w_2$ with $w_1 \in [\![T_1]\!]$ and $w_2 \in [\![T_2]\!]$. If $w \models a^+$, then either $w_1 \models a^+$ or $w_2 \models a^+$, hence either $w_1 \models B^+$ or $w_2 \models B^+$, hence $w \models B^+$, hence $T_1 \otimes T_2 \models a^+ \Rrightarrow B^+$. □

We need now to find an algorithm to prove $T \models A^{+?}$. The structure of the algorithm is described by the following lemma.

Lemma 3.7 ($T \models A^{+?}$). *For any type T, T_1, and T_2:*

(1) $\epsilon \models A^{+?}$
(2) $a \models A^{+?} \quad \Leftrightarrow \quad a \in A$
(3) $T\,[m..n] \models A^{+?} \quad \Leftrightarrow \quad T \models A^{+?}$
(4) $T_1 \otimes T_2 \models A^{+?} \quad \Leftrightarrow \quad (T_1 \models A^{+?} \wedge \neg \mathrm{N}(T_1)) \vee (T_2 \models A^{+?} \wedge \neg \mathrm{N}(T_2)) \vee (T_1 \models A^{+?} \wedge T_2 \models A^{+?})$
(5) $T_1 + T_2 \models A^{+?} \quad \Leftrightarrow \quad T_1 \models A^{+?} \wedge T_2 \models A^{+?}$
(6) $T! \models A^{+?} \quad \Leftrightarrow \quad T \models A^{+?}$

Proof. We prove each case directly.

(1 & 2) Trivial.

(3) ($\Rightarrow$): Assume that $T\,[m..n] \models A^{+?}$. Consider $w \in [\![T]\!]$; by this assumption, w^m (w repeated m times) satisfies $A^{+?}$, hence either w^m is empty or $sym(w^m) \cap A \neq \emptyset$. As a consequence, either w is empty or $sym(w) \cap A \neq \emptyset$, hence $w \models A^{+?}$.
($\Leftarrow$): Assume that any non-empty $w \in [\![T]\!]$ satisfies $sym(w) \cap A \neq \emptyset$. Then, any non-empty word obtained by concatenating words from $[\![T]\!]$ satisfies the same property.

(4) ($\Rightarrow$): Assume (a) $T_1 \otimes T_2 \models A^{+?}$, ($b$) $\neg(T_1 \models A^{+?} \wedge \neg \mathrm{N}(T_1))$ and (c) $\neg(T_2 \models A^{+?} \wedge \neg \mathrm{N}(T_2))$; we want to prove that $T_1 \models A^{+?}$ and $T_2 \models A^{+?}$. By Property 3.4(3), we rewrite (b) and (c) as (b') $T_1 \not\models A^+$ and (c') $T_2 \not\models A^+$. Let $w_1 \in [\![T_1]\!]$: by (c'), $\exists w_2 \in [\![T_2]\!]$ such that $w_2 \not\models A^+$. By (a), $w_1 \cdot w_2 \models A^{+?}$. If $w_1 \cdot w_2$ is not empty, then $w_2 \not\models A^+$ implies that $w_1 \models A^+$, hence $w_1 \models A^{+?}$. If $w_1 \cdot w_2$ is empty, then w_1 is empty, hence $w_1 \models A^{+?}$. Hence, we have proved that $T_1 \models A^{+?}$. We can prove $T_2 \models A^{+?}$ in the same way.
($\Leftarrow$): Assume $(T_1 \models A^{+?} \wedge \neg \mathrm{N}(T_1)) \vee (T_2 \models A^{+?} \wedge \neg \mathrm{N}(T_2)) \vee (T_1 \models A^{+?} \wedge T_2 \models A^{+?})$. Consider any non-empty $w \in [\![T_1 \otimes T_2]\!]$. If the first disjunct holds, then $\neg \mathrm{N}(T_1)$ implies that w contains a non-empty subword from T_1, and $T_1 \models A^{+?}$ implies $w \models A^{+?}$. If the second disjunct holds, then w contains a non-empty subword from T_2, and $T_2 \models A^{+?}$ implies $w \models A^{+?}$. When the third disjunct holds w, being non-empty, contains a non-empty subword either from T_1 or from T_2, and that subword satisfies $A^{+?}$. Hence, in any of the three cases, w contains a non-empty subword that satisfies $A^{+?}$, hence we conclude that $w \models A^{+?}$.

(5) $T_1 + T_2 \models A^{+?}$ iff $[\![T_1]\!] \cup [\![T_2]\!] \subseteq [\![A^{+?}]\!]$, iff $[\![T_1]\!] \subseteq [\![A^{+?}]\!]$ and $[\![T_2]\!] \subseteq [\![A^{+?}]\!]$, iff $T_1 \models A^{+?}$ and $T_2 \models A^{+?}$.

(6) $T! \models A^{+?} \Leftrightarrow T \models A^{+?}$ holds because $T' \models A^{+?}$ only depends on the non-empty words of T', and $T!$ has the same non-empty words as T. □

Case (4) of the lemma is the most interesting: while in case (5) we need both $T_1 \models A^{+?}$ and $T_2 \models A^{+?}$ in order to prove $T_1 + T_2 \models A^{+?}$, case (4) states that, for product types, non-nullability of T_1 would compensate the fact that T_2 does not satisfy $A^{+?}$, and vice versa. Case (6) is also quite important. Most of the results that we illustrate in this section would hold unchanged if we used A^+ rather than $A^{+?}$, apart from the inductive computation of $T! \models A^+$, where we would have had to go through the computation of $T \models A^{+?}$. Case (6) shows that the inductive computation of $T! \models A^{+?}$ is, instead, trivial. Hence, case (6) is the main reason why we base our algorithm on the auxiliary constraint $A^{+?}$ rather than on A^+.

```
$CoImplies$(Markable Type $T$, Type $U$)        -- Markable Type $T$: $T$ is an object with two
                                                -- writable bits $T.Non\text{-}nullable$ and $T.Marked$
  Array $NodesOfSymbol$;          -- Array mapping symbols to lists of nodes of $T$
  $Prepare(T, NodesOfSymbol)$;    -- For each $a \in sym(T)$, initializes $NodesOfSymbol[a]$ with
                                  -- pointers to all subterms $T_i$ of $T$ such that $T_i = a$
                                  -- and, for each subterm $T_i$ of $T$, initializes $T_i.Non\text{-}nullable$
  $CCU = \mathcal{CC}(U)$;        -- Quadratic time computation of $\mathcal{CC}(U)$
  return (every $A^+ \Rightarrow B^+$ in $CCU$ satisfy $CoCheck(T, NodesOfSymbol, A, B)$)

$CoCheck$(Markable Type $T$, Array $NodesOfSymbol$, Set $A$, Set $B$)
  -- Next line unmarks each node of $T$
  for $T'$ in nodes-of($T$) do $T'.Marked$ = false;
  -- Postcondition for $MarkBP(B, T)$: for any $a_i$ in $T$, $a_i.Marked$ = true iff $a_i$ satisfies
  -- the second condition of Theorem 3.6, i.e., $a_i$ is inside a product that satisfies $B^{+?}$
  $MarkBP(T, B)$;
  -- Check whether every occurrence in T of each a in A is marked
  return (every $a$ in $A$, $T_a$ in $NodesOfSymbol[a]$ satisfy $T_a.Marked$)

$MarkBP$(Type $T$, Set $B$)
  -- Returns true iff $T \models B^{+?}$
  -- and marks all nodes of $T$ included in a product subterm $T'$ such that $T' \models B^{+?}$
  Boolean $result$;
  case $T$ when $T_1\,[m..n]$ or $T_1!$ :  $result = MarkBP(T_1, B)$;
           when $T_1 \otimes T_2$:          $BPP1 = MarkBP(T_1, B)$;
                                            $BPP2 = MarkBP(T_2, B)$;
                                            $result = (BPP1 \wedge T_1.Non\text{-}nullable)$
                                            $\vee\, (BPP2 \wedge T_2.Non\text{-}nullable)$
                                            $\vee\, (BPP1 \wedge BPP2)$;
                                            if $result$ then { $MarkAll(T_1)$; $MarkAll(T_2)$; }
           when $T_1 + T_2$:                $result = MarkBP(T_1, B) \wedge MarkBP(T_2, B)$;
           when $\epsilon$:                 $result$ = true;
           when $a$:                        $result = a \in B$;
  $T.Marked = result$;
  return $result$;

$MarkAll$(Type $T$)
  $T.Marked$ = true;
  case $T$ when $T_1!$ or $T = T_1\,[m..n]$: if not $T_1.Marked$ then $MarkAll(T_1)$;
           when $T_1 + T_2$ or $T_1 \otimes T_2$:   if not $T_1.Marked$ then $MarkAll(T_1)$;
                                         if not $T_2.Marked$ then $MarkAll(T_2)$;
           when $\epsilon$ or $a$:                  return;
```

Fig. 2. Algorithm for implication of co-occurrence constraints.

The algorithm

We can now use the previous results to define an algorithm *CoImplies* to verify that $T \models A^+ \Rightarrow B^+$ holds for each $A^+ \Rightarrow B^+ \in \mathcal{CC}(U)$ in time $O(n^2)$, where $n = |T| + |U|$.

In our complexity analysis we assume that every symbol appearing in T and U can be manipulated in constant time, and that symbols can be used as indexes for arrays, that is, for structures that can be accessed in constant time. This reflects the fact that subtyping is typically computed by a compiler, which represents symbols as fixed-size pointers to entries of a symbol table. This approach provides both constant time manipulation of symbols and constant time indexing by symbols. In a more abstract setting, assuming the RAM machine model, this corresponds to assuming that symbols of T and U come from a set of size $k \times n$, that is $k \cdot (|T| + |U|)$, for some fixed k, so that any symbol only occupies $O(\log(n))$ bits, which, in the RAM model, yields the desired constant-time operations. As previously stated, this assumption mirrors the typical implementation. In any case, it could be easily removed by adding a preprocessing phase where symbols are normalized to integers in the range $1 \ldots n$, normalization that can be easily performed in time $O(n \cdot \log(n))$, dominated by the $O(n^2)$ complexity of the whole algorithm.

Our algorithm, listed in Fig. 2, for each constraint $A^+ \Rightarrow B^+ \in \mathcal{CC}(U)$, first marks every leaf of T that is included in a product subterm T' of T such that $T' \models B^{+?}$ — this operation is performed by *MarkBP*, using the procedure specified by

Lemma 3.7. By Property 3.2 and Theorem 3.6, $T \models A^+ \Rrightarrow B^+$ holds if each occurrence of each a in A has been marked — this condition is verified by *CoCheck*.

In greater detail, *CoImplies* first calls *Prepare*(T), which performs some linear-time preprocessing. Specifically, *Prepare*(T) prepares a array *NodesOfSymbol* that associates every symbol a of $sym(T)$ with the list of all leaves in T that contain an occurrence of the symbol a, and decorates each node T' of T with a nullability bit that specifies whether $\mathrm{N}(T')$ holds, so that the nullability tests of *MarkBP* will only need constant time. This procedure runs in time $O(|T|)$, hence $O(n)$, and is not presented in Fig. 2. Then *CoImplies* initializes the list *CCU* with $\mathcal{CC}(U)$. The list of constraints *CCU* represents each constraint $A^+ \Rrightarrow B^+$ of $\mathcal{CC}(U)$ as a list A and a bit array for B, so that A can be scanned in linear time and tests $b \in B$ in *MarkBP* can be run in constant time. Function $\mathcal{CC}(U)$ directly executes the definition of $\mathcal{CC}(U)$, hence it only requires one scan of U and, for each product node of U, time $O(|A| + |B|)$ to build the data structure representing $A^+ \Rrightarrow B^+$, hence $\mathcal{CC}(U)$ needs time $O(|U| \times (|A| + |B|))$. Since $|A| + |B| \leq |sym(U)|$, then $\mathcal{CC}(U)$ runs in time $O(|U| \times |U|)$, that is, $O(n^2)$.

Finally, the algorithm invokes *CoCheck*(T, A, B) once for each constraint $A^+ \Rrightarrow B^+$ in $\mathcal{CC}(U)$ in order to prove that $T \models A^+ \Rrightarrow B^+$ holds for all of them. *CoCheck*(T, A, B) is invoked $O(n)$ times, and we are going to show that it runs in time $O(n)$, hence concluding the proof that *CoImplies* runs in time $O(n^2)$.

CoCheck(T, A, B) performs the following three operations:

- it unmarks all the nodes of T, in time $O(T)$;
- it calls *MarkBP*(T, B) in order to mark all the nodes contained in each subtree T' such that $T' \models B^{+?}$. This phase is performed by a bottom-up visit of T, as specified by Lemma 3.7, and we discuss its run-time cost later on;
- for each $a \in A$, each node corresponding to an occurrence a_i of a in the syntax tree of T is checked to verify whether it has been marked by the previous step; this check only needs time $O(|T|)$. Thanks to Property 3.2 and Theorem 3.6, the algorithm concludes that $T \models A^+ \Rrightarrow B^+$ if, and only if, this step succeeds.

We show now that the auxiliary function *MarkBP(T, B)* can be executed in $O(|T|)$ time. *MarkBP* is called once for each node of T and, for each node, performs some constant-time operations plus two calls, at most, to function *MarkAll*, hence the total cost of *MarkBP* is in $O(|T|)$ plus the total cost of *MarkAll*. *MarkAll* is either invoked by *MarkBP* or, recursively, by *MarkAll* itself, it only performs constant time operations apart from its recursive calls, and is never invoked twice on the same node, hence its total cost is in $O(|T|)$. Therefore *MarkBP* can be computed in $O(|T|)$ time, hence the whole algorithm runs in time $O(n^2)$.

Remark 3.8. Although $\mathcal{C}(U)$ is $\mathscr{F}$-complete for a conflict-free type U, $\mathcal{CC}(U)$ is not complete for U with respect to constraints with shape $A^+ \Rrightarrow B^+$. For example, $\mathcal{CC}(a)$ is the empty set, which denotes the whole Σ^*, and it could be made more precise by adding any non-trivial constraint $A^+ \Rrightarrow B^+$ sound for a, such as, for example, $b^+ \Rrightarrow c^+$, which is sound since b is disjoint from a, and excludes words such as b.

$\mathcal{C}(U)$ is $\mathscr{F}$-complete because it completes $\mathcal{CC}(U)$ with the constraint upper$(sym(U))$. For example, in this case, upper(a) makes $b^+ \Rrightarrow c^+$ redundant.

A similar remark holds for the order constraints that we define in the next section: $\mathcal{OC}(U)$ is complete for order constraints that only use symbols in $sym(U)$, but is not complete for every possible order constraint. However, Proposition 2.16 (*asymmetric subtyping*) does not require that every component of $\mathcal{C}(U)$ is complete on its class of constraints, but only that the whole of $\mathcal{C}(U)$ is $\mathscr{F}$-complete.

3.2. *Order constraints $\mathcal{OC}(U)$*

Overview

In this section we present a polynomial algorithm to verify that $T \models a \prec b$ for each $a \prec b \in \mathcal{OC}(U)$, where U is a conflict-free type while T is an arbitrary extended RE. Our approach is a direct generalization of the algorithm presented in [1]. In that paper we proved that, for any conflict-free type U and any $\{a, b\} \subseteq sym(U)$, one can decide whether $U \models a \prec b$ by inspecting the Least Common Ancestor of the only occurrence of a and the only occurrence of b in U. Here, we show that, for any arbitrary type T, one can decide whether $T \models a \prec b$ by inspecting the Least Common Ancestor of each pair (a_i, b_i), where a_i is an occurrence of a in $l(T)$ – a labelled version of T – and b_i is an occurrence of b in $l(T)$. We also show that this test can be completed, for all constraints in $\mathcal{OC}(U)$, in time $O(|T|^2 + |U|)$, hence in time $O(n^2)$, where $n = (|T| + |U|)$.

To this end, we will first define $\mathrm{p}(T) \subseteq (sym(T) \times sym(T))$ as the set of all ordered pairs of symbols (a, b) such that an a comes before a b in one word of T, so that $T \models a \prec b$ iff $(b, a) \notin \mathrm{p}(T)$. We then define labelled types $l(T)$. These are types where each occurrence of a symbol a is labelled with a unique index, and each occurrence of a binary operator is labelled with a bit that specifies whether that occurrence is in the scope of a counting operator $\_[m..n]$. We finally show how to deduce whether $(a, b) \in \mathrm{p}(T)$ from the Least Common Ancestor of the occurrences of a and b in $l(T)$.

Constraints extraction $\mathcal{OC}(U)$

The second component $\mathcal{OC}(U)$ of $\mathcal{C}(U)$ extracts a set of order constraints with shape $a \prec b$, and is defined, in [1], as follows. The notation $sym(U) \prec sym(U')$ stands for $\{a \prec b \mid a \in sym(U),\ b \in sym(U')\}$.

$$\begin{aligned}
\mathcal{OC}(U_1 \cdot U_2) &\overset{def}{=} sym(U_1) \prec sym(U_2) \ \cup\ \mathcal{OC}(U_1) \cup\ \mathcal{OC}(U_2)\\
\mathcal{OC}(U_1 \& U_2) &\overset{def}{=} \mathcal{OC}(U_1) \cup\ \mathcal{OC}(U_2)\\
\mathcal{OC}(U_1 + U_2) &\overset{def}{=} sym(U_1) \prec sym(U_2) \ \cup\ sym(U_2) \prec sym(U_1) \ \cup\ \mathcal{OC}(U_1) \cup\ \mathcal{OC}(U_2)\\
\mathcal{OC}(U!) &\overset{def}{=} \mathcal{OC}(U)\\
\mathcal{OC}(\epsilon) &\overset{def}{=} \emptyset\\
\mathcal{OC}(a) &\overset{def}{=} \emptyset\\
\mathcal{OC}(a\,[m..n]) &\overset{def}{=} \emptyset
\end{aligned}$$

Observe that, for union types $U_1 + U_2$, the conjunction of the two order constraints $sym(U_1) \prec sym(U_2)$ and $sym(U_2) \prec sym(U_1)$ cannot be satisfied by any word that includes a symbol from $sym(U_1)$ and a symbol from $sym(U_2)$. The type $U_1 + U_2$ satisfies this conjunction since it is conflict-free, hence $sym(U_1)$ and $sym(U_2)$ are disjoint, and since no counting operator may be applied to this sum. For example, a type $(a \cdot b) + (b \cdot a)$ or a type $(a + b)\,[1..2]$ would not satisfy these constraints. Similar considerations apply to the constraints $sym(U_1) \prec sym(U_2)$ associated with $U_1 \cdot U_2$: they also depend on disjointness and lack of external counting.

Since $\mathcal{OC}(U)$ is a subset of $\{a \prec b \ \mid\ a \in sym(U),\ b \in sym(U)\}$, its size is less than $|U|^2$. An inductive computation based on the definition would also take $O(n^2)$ time, since no pair is ever generated twice, hence each union operator in the definition can be implemented as list concatenation.

The following example illustrates the definition.

Example 3.9.

- $\mathcal{OC}(a\,[1..2] \cdot (b\,[2..*] + c\,[1..*])) = \{a \prec b,\ a \prec c,\ b \prec c,\ c \prec b\}$.

Formal treatment

Let us define $\mathsf{p}(T)$ as the set of all pairs of different symbols (a, b) such that there exists a word in $[\![T]\!]$ where an a comes before a b.

Definition 3.10 ($\mathsf{p}(T)$)**.**

$$\mathsf{p}(T) \overset{def}{=} \{(a, b) \mid a \neq b, \exists w_1, w_2, w_3.\ w_1 \cdot a \cdot w_2 \cdot b \cdot w_3 \in [\![T]\!]\}$$

Order constraints specify which pairs cannot appear in a word, hence $\mathsf{p}(T)$ is related to order constraints as follows.

Property 3.11. *For all $a \in \Sigma$, $b \in \Sigma$, such that $a \neq b$:*

$$T \models a \prec b \Leftrightarrow (b, a) \notin \mathsf{p}(T)$$

Our algorithm verifies whether $T \models a \prec b$ by checking whether $(b, a) \in \mathsf{p}(T)$. We verify whether (b, a) is in $\mathsf{p}(T)$ by testing, for each occurrence of a and b in T, their Least Common Ancestor (LCA) in the syntax tree of T. For example, in $(b \cdot c) + a$, the LCA of the only occurrences of a and b is an occurrence of '+' (hereafter, we will just say: the LCA is '+'), and from this fact we will be able to deduce that $(b, a) \notin \mathsf{p}(T)$. In $(b\&a) + a$, however, we have to analyze two different occurrences of a. To this aim, we will consider a decorated version of T — $l(T)$ — such that each leaf of $l(T)$ is decorated with a distinct index i (Definition 3.14).

However, $(b, a) \in \mathsf{p}(T)$ does not only depend on the LCA of a and b in T, but also on the presence, or absence, of a counting operator (different from $T\,[1..1]$) in any higher position of the syntax tree. For example, $(b, a) \notin \mathsf{p}(a + b)$ but $(b, a) \in \mathsf{p}((a + b)\,[1..2])$, since $ba \in [\![(a + b)\,[1..2]]\!]$.

For this reason, in $l(T)$, we also mark each occurrence of a binary operator $\circledast$ as $\circledast_r$ if it is into the scope of any counting operator (where r stands for *repeated*), and as $\circledast_s$ (where s stands for *single*) otherwise; this is formalized in Definition 3.14.

Definition 3.12 (*Labelled Types*)**.** A labelled type L over an alphabet Σ is a term generated by the following grammar, where $a_i \in (\Sigma \times \mathbb{N}), \alpha \in \{s, r\}$,

$$L ::= \epsilon \ \mid\ a_i \mid L\,[m..n] \mid L +_\alpha L \mid L \cdot_\alpha L \mid L \,\&_\alpha\, L \mid L!$$

A labelled type L is *well-formed* if:

- L satisfies the well-formedness conditions of Definition 2.4;
- L satisfies *single-occurrence*, that is, no pair a_i appears twice in L.

Observe that the single-occurrence restriction regards the symbol-integer pair a_i: the same symbol a may occur many times in a well-formed labelled type, provided that the index is different in any occurrence.

The semantics of a labelled type L is a set of words formed by labelled symbols, that is: $[\![L]\!] \subseteq (\Sigma \times \mathbb{N})^*$, and is defined in the obvious way.

Definition 3.13 ($[\![L]\!]$).

$$
\begin{aligned}
[\![\epsilon]\!] &\overset{def}{=} \{\epsilon\}\\
[\![a_i]\!] &\overset{def}{=} \{a_i\}\\
[\![L_1 +_\alpha L_2]\!] &\overset{def}{=} [\![L_1]\!] \cup [\![L_2]\!]\\
[\![L_1 \cdot_\alpha L_2]\!] &\overset{def}{=} [\![L_1]\!] \cdot [\![L_2]\!]\\
[\![L_1 \&_\alpha L_2]\!] &\overset{def}{=} [\![L_1]\!] \& [\![L_2]\!]\\
[\![L!]\!] &\overset{def}{=} [\![L]\!] \setminus \{\epsilon\}\\
[\![L\,[m..n]]\!] &\overset{def}{=} \{w \mid w = w_1 \cdot \ldots \cdot w_j, \quad \forall i \in 1..j.\ w_i \in [\![L]\!],\ m \le j \le n\}
\end{aligned}
$$

We can now define the function $l(\_)$, that maps every type T into a well-formed labelled type $l(T)$. To this aim we define an auxiliary recursive function $l_i^\alpha(T)$ with two parameters α and i. The integer i is the next index that is available to mark a leaf, while $\alpha \in \{s, r\}$ is r if the current subtree is inside a non-trivial counting operator, and is s otherwise. The function $l(\_)$ satisfies a strong form of single-occurrence, since it uses a different index for each different non-ϵ leaf; this strong form of single-occurrence is just an irrelevant consequence of the simple indexing technique that we adopt here.

Definition 3.14 ($l(T)$). $l(T)$ abbreviates $l_1^s(T)$, where $l_i^\alpha(T)$ is defined as follows, for any $\alpha \in \{s, r\}$ and $i \in \mathbb{N}$:

$$
\begin{aligned}
l_i^\alpha(\epsilon) &\overset{def}{=} \epsilon\\
l_i^\alpha(a) &\overset{def}{=} a_i\\
l_i^\alpha(T_1 \circledast T_2) &\overset{def}{=} l_i^\alpha(T_1) \circledast_\alpha l_{m+1}^\alpha(T_2) \text{ where } m = \max\{ j \mid a_j \in sym(l_i^\alpha(T_1)) \}\\
l_i^\alpha(T!) &\overset{def}{=} (l_i^\alpha(T))!\\
l_i^\alpha(T\,[1..1]) &\overset{def}{=} (l_i^\alpha(T))\,[1..1]\\
l_i^\alpha(T\,[m..n]) &\overset{def}{=} (l_i^r(T))\,[m..n] \quad n > 1
\end{aligned}
$$

In a type $a \cdot b$, order is relevant: $(a, b) \in \mathsf{p}(T)$ but $(b, a) \notin \mathsf{p}(T)$. We express this fact by extending the usual definition of $LCA_{l(T)}[a_i, b_j]$, so that it returns a pair $\circledast^d$, where the direction d is $\rightarrow$ if the leaf a_i comes before b_j in T, and is $\leftarrow$ otherwise; we ignore the direction when $\circledast \neq \cdot_s$ (see Definition 3.15 and Example 3.19). For our aims, we only need to define $LCA_{l(T)}[\_, \_]$ for pairs of leaves which are distinct and which are both indexed symbols.

Definition 3.15 ($LCA_{l(T)}[a_i, b_j]$). Let a_i and b_j be two different non-ϵ leaves of $l(T)$. Then, the lowest common ancestor of a_i and b_j in $l(T)$ is inductively defined as follows.

$$
\begin{aligned}
LCA_{L!}[a_i, b_j] &\overset{def}{=} LCA_L[a_i, b_j]\\
LCA_{L[m..n]}[a_i, b_j] &\overset{def}{=} LCA_L[a_i, b_j]\\
LCA_{L_1 \circledast_\alpha L_2}[a_i, b_j] &\overset{def}{=} \begin{cases} LCA_{L_1}[a_i, b_j] & \text{if } a_i \text{ and } b_j \text{ are leaves of } L_1\\ LCA_{L_2}[a_i, b_j] & \text{if } a_i \text{ and } b_j \text{ are leaves of } L_2\\ \circledast^{\rightarrow}{}_\alpha & \text{if } a_i \text{ is in } L_1, b_j \text{ is in } L_2, \text{ and } \circledast_\alpha = \cdot_s\\ \circledast^{\leftarrow}{}_\alpha & \text{if } a_i \text{ is in } L_2, b_j \text{ is in } L_1, \text{ and } \circledast_\alpha = \cdot_s\\ \circledast_\alpha & \text{otherwise} \end{cases}
\end{aligned}
$$

Note that, in each use of $LCA_{L'}[a_i, b_j]$ at the right-hand side of the definition, a_i and b_j are still two different leaves of L'.

We now need to introduce the notion of *non-repetitive context*, with some of its properties. We will use C to denote a context, that is a labelled type where exactly one leaf is the special symbol _, and $C[L]$ to denote the labelled type obtained by substituting _ with L in C. Hence, contexts are generated by the following grammar:

$$C ::= \_ \mid L \circledast_\alpha C \mid C \circledast_\alpha L \mid C! \mid C\,[m..n]$$

We say that a context is *non-repetitive* when is generated by the following grammar, that differs from the full grammar in the last case. When a context can only be generated by the full grammar, we say that it is *repetitive*.

$$C_s ::= \_ \mid L \circledast_\alpha C_s \mid C_s \circledast_\alpha L \mid C_s! \mid C_s\,[1..1]$$

Hence, a context is repetitive if, and only if, the hole _ is in the scope of a non-trivial counting operator.
Repetitive and non-repetitive contexts enjoy the following properties (whose proofs are reported in the Appendix).

Lemma 3.16. *Let C_s be a non-repetitive labelled context and L a labelled type such that $C_s[L]$ is well-formed, then:*

$$w \in [\![C_s[L]]\!] \wedge (sym(w) \cap sym(L)) \neq \emptyset \Rightarrow \exists w_1 \in [\![L]\!], w_2 \in sym(C_s)^* :\ w \in w_1 \& w_2$$

Property 3.17. *Let C_s be a non-repetitive labelled context and L a labelled type such that $C_s[L]$ is well-formed, then:*

$$\{a_i, b_j\} \subseteq sym(L) \wedge (a_i, b_j) \notin \mathsf{p}(L) \Rightarrow (a_i, b_j) \notin \mathsf{p}(C_s[L]) \qquad (1)$$

If C is any labelled context and L a labelled type such that $C[L]$ is well-formed, then:

$$\mathsf{p}(L) \subseteq \mathsf{p}(C[L]) \tag{2}$$

If C_r is a repetitive labelled context and L a labelled type such that $C_r[L]$ is well-formed, then:

$$\{a_i, b_j\} \subseteq sym(L) \Rightarrow (a_i, b_j) \in \mathsf{p}(C_r[L]) \tag{3}$$

We can finally prove the fundamental result of this section, that is the basis of our algorithm.

Property 3.18 ($\mathsf{p}(T)$ *and* $T \models a \prec b$)**.** *For any type T and $a \neq b$:*

$$\begin{array}{lll}
(1)\ (a_i, b_j) \in \mathsf{p}(l(T)) & \Leftrightarrow & LCA_{l(T)}[a_i, b_j] \in \{\circledast_r, \&_s, \cdot_s^{\rightarrow}\} \\
(2)\ (a, b) \in \mathsf{p}(T) & \Leftrightarrow & \exists a_i, b_j \in sym(l(T)).\ LCA_{l(T)}[a_i, b_j] \in \{\circledast_r, \&_s, \cdot_s^{\rightarrow}\} \\
(3)\ (b, a) \notin \mathsf{p}(T) & \Leftrightarrow & \forall a_i, b_j \in sym(l(T)).\ LCA_{l(T)}[a_i, b_j] \in \{+_s, \cdot_s^{\rightarrow}\} \\
(4)\ T \models a \prec b & \Leftrightarrow & \forall a_i, b_j \in sym(l(T)).\ LCA_{l(T)}[a_i, b_j] \in \{+_s, \cdot_s^{\rightarrow}\}
\end{array}$$

Proof. (1) $\Rightarrow$: we observe that the complement of $\{\circledast_r, \&_s, \cdot_s^{\rightarrow}\}$ is $\{+_s, \cdot_s^{\leftarrow}\}$, and show that $LCA_{l(T)}[a_i, b_j] \in \{+_s, \cdot_s^{\leftarrow}\} \Rightarrow (a_i, b_j) \notin \mathsf{p}(l(T))$.

If $LCA_{l(T)}[a_i, b_j] = +_s$, then $l(T) = C_s[L_1 + L_2]$, where C_s is a non-repetitive context, and where a_i only appears in L_1 and b_j only appears in L_2, or vice versa (exchanging 1 with 2). In both cases, no single word of $L_1 + L_2$ may contain both a_i and b_j, hence $(a_i, b_j) \notin \mathsf{p}(L_1 + L_2)$, hence $(a_i, b_j) \notin \mathsf{p}(C_s[L_1 + L_2])$, by Property 3.17(1). The case for $LCA_{l(T)}[a_i, b_j] = \cdot_s^{\leftarrow}$ is analogous, with the only difference that $l(T) = C_s[L_2 \cdot L_1]$ where a_i only appears in L_1 and b_j only appears in L_2, which implies that $(a_i, b_j) \notin \mathsf{p}(C_s[L_2 \cdot L_1])$.

(1) $\Leftarrow$: assume $LCA_{l(T)}[a_i, b_j] \in \{\circledast_r, \&_s, \cdot_s^{\rightarrow}\}$. In the first case, the LCA of a_i and b_j is in a repetitive context, hence $(a_i, b_j) \in \mathsf{p}(l(T))$ follows from Property 3.17(3). If $LCA_{l(T)}[a_i, b_j] = \&_s$, then $l(T) = C[L_1\&L_2]$ with $a_i \in sym(L_1)$ and $b_j \in sym(L_2)$ or vice versa, hence, by Lemma 2.8(2) and the definition of $\mathsf{p}(\_)$, $(a, b) \in \mathsf{p}(L_1\&L_2)$, and the result follows because $\mathsf{p}(L_1\&L_2) \subseteq \mathsf{p}(C[L_1\&L_2])$ (Property 3.17(2)). In case $LCA_{l(T)}[a_i, b_j] = \cdot_s^{\rightarrow}$ we reason in the same way.

(2) $(a, b) \in \mathsf{p}(T) \Leftrightarrow \exists i, j.\ a_i, b_j \in \mathsf{p}(l(T))$, hence (2) follows from (1).

(3) follows from (2) by negating the two sides and exchanging a with b.

(4) follows from (3) since $T \models a \prec b$ iff $(b, a) \notin \mathsf{p}(T)$. $\square$

Observe that, if any of a and b is not in $sym(T)$, then $T \models a \prec b$ holds trivially. This is expressed by the universal quantification found in Property 3.18(4): if, for example, no leaf of T is b, then no b_j belongs to $sym(l(T))$, hence the condition $\forall a_i, b_j \in sym(l(T)) :\ LCA_{l(T)}[a_i, b_j] \in \{+_s, \cdot_s^{\rightarrow}\}$ is trivially satisfied.

Example 3.19 ($LCA_{l(T)}[a_i, b_j]$)**.** If $T = a\cdot ((b + a)\,[1..3])$, then $l(T) = a_1 \cdot_s ((b_2 +_r a_3)\,[1..3])$, and $LCA_{l(T)}[a_i, b_j]$ is defined as in the following table.

	a_1	b_2	a_3
a_1	–	$\cdot_s^{\rightarrow}$	$\cdot_s^{\rightarrow}$
b_2	$\cdot_s^{\leftarrow}$	–	$+_r$
a_3	$\cdot_s^{\leftarrow}$	$+_r$	–

Hence, we have $(b, a) \in \mathsf{p}(T)$ because $LCA_{l(T)}[b_2, a_3] = +_r$, and $(a, b) \in \mathsf{p}(T)$ because $LCA_{l(T)}[a_1, b_2] = \cdot_s^{\rightarrow}$, but also because $LCA_{l(T)}[a_3, b_2] = +_r$. Hence, $T \not\models a \prec b$ and $T \not\models b \prec a$.

The algorithm

Our algorithm to verify whether $\forall F \in \mathcal{OC}(U) :\ T \models F$ is based on Property 3.18 and is shown in Fig. 3. It first decorates T and builds, for both $l(T)$ and U, a data structure to compute the LCA of any two leaves in constant time. This preprocessing phase is done in linear time using the algorithm described by Bender and Farach-Colton in [13].[5] The lists $Leaves_{l(T)}$ and $Symbols_T$ contain the leaves of $l(T)$ and the symbols of T, the array $LeafOfSymbol_U$ maps each symbol in $sym(U)$ to the only corresponding leaf in U, and $SymbolOfNode_{l(T)}$ maps each leaf node a_i in $l(T)$ to the corresponding symbol a.

The algorithm first prepares a data structure OC_T such that $OC_T[a][b] =$ **true** iff $T \models a \prec b$. $OC_T[a][b]$ is first initialized as **true** everywhere, then every pair of leaves a_i, b_j in $Leaves_{l(T)}$ with $a \neq b$ is analyzed and the corresponding value $OC_T[a][b]$ is set to **false** if the LCA of a_i, b_j is not in $\{+_s, \cdot_s^{\rightarrow}\}$, according to Property 3.18. Then, in order to check whether $\forall (a \prec b) \in \mathcal{OC}(U) :\ T \models a \prec b$, we scan each pair of symbols (a, b) in T such that $a \prec b \in \mathcal{OC}(U)$, and use OC_T to verify whether $T \models a \prec b$. In this way, we check the implication $(a \prec b) \in \mathcal{OC}(U) \Rightarrow T \models a \prec b$ for all the constraints $(a \prec b) \in \mathcal{OC}(U)$ where both symbols are in $sym(T)$. We can safely ignore the constraints $(a \prec b) \in \mathcal{OC}(U)$ where a, b, or both, are in $sym(U) \setminus sym(T)$, since these constraints trivially hold in T.

The preprocessing phase of *OrderImplies* (lines 1–2) has complexity $O(|T| + |U|)$. The definition and initialization of OC_T (lines 3–7) requires two different loops and takes time $O(|T|^2)$, since we only require a constant time for any pair of leaves of T.

The main loop of the algorithm (lines 8–12) only needs $O(|sym(T)|^2)$ time, since we only have constant time operations for each pair of symbols in T, hence the total complexity of the three phases is $O(|T|^2 + |U|)$. Hence, also in this case, the extension from conflict-free inclusion to asymmetric inclusion adds no time complexity to the algorithm.

[5] In this case, LCA_i is not really a bidimensional array, it is a linear-space object with a constant time access.

```
OrderImplies(Type $T$, Type $U$): -- we assume that $sym(T) \subseteq sym(U)$
 -- Lines 1 and 2: Linear time preprocessing to build all data structures
 -- needed to compute $LCA_{I(T)}[n_1, n_2]$ and $LCA_U[n_1, n_2]$ in constant time
 -- and to scan the symbols and leaves of $T$ and $U$
 1 $LCA_{I(T)}, Leaves_{I(T)}, Symbols_T, SymbolOfNode_{I(T)} =$ PreprocessGeneralType($T$);
 2 $LCA_U, LeafOfSymbol_U =$ PreprocessCFType($U$);
 -- $OC_T[a][b]$ will contain false iff $(a, b) \in (Symbols_T \times Symbols_T)$ and $T \not\models a \prec b$
 3 for each $(a, b)$ in $Symbols_T \times Symbols_T$ where $a \neq b$: $OC_T[a][b] =$ true
 4 for each $n_1$ in $Leaves_{I(T)}$, $n_2$ in $Leaves_{I(T)}$:
 5     $a = SymbolOfNode_{I(T)}[n_1]$
 6     $b = SymbolOfNode_{I(T)}[n_2]$
 7     if $a \neq b$ then $OC_T[a][b] = OC_T[a][b] \wedge (LCA_{I(T)}[n_1, n_2] \in \{+_s, \cdot_s^{\rightarrow}\})$
 -- the main loop looks for an $F \in \mathcal{OC}(U)$ such that $T \not\models F$
 8 for each $(a, b)$ in $Symbols_T \times Symbols_T$ where $a \neq b$:
 9     if   $(LCA_U[LeafOfSymbol_U(a), LeafOfSymbol_U(b)]$ in $\{+_s, \cdot_s^{\rightarrow}\})$
 10         and not $OC_T[a][b]$
 11     then return false
 12 return true
```

Fig. 3. Algorithm for implication of order constraints.

3.3. Cardinality constraints *ZeroMinMax*(U)

Overview

In this section we present a polynomial algorithm to check $T \models a?[m..n]$ for each $a?[m..n] \in ZeroMinMax(U)$. In the constraint $a?[m..n]$, values m and n denote, respectively, the minimum and the maximum values of $|w|_a$ for those words w of U where a actually appears. Our basic idea is to compute the same quantities for any type T. We denote these as $\mathrm{Min}^{app}(T, a)$ and $\mathrm{Max}(T, a)$ respectively, and we verify $T \models a?[m..n]$ by checking that $m \leqslant \mathrm{Min}^{app}(T, a)$ and $\mathrm{Max}(T, a) \leqslant n$.

Evaluating the minimal value of $|w|_a$ over the words of T where a appears is not immediate, because of symbol repetition and generalized counting. Consider, for example, the type $a[2..*] \cdot a[3..*]$: it clearly satisfies $a?[5..*]$; here, $\mathrm{Min}^{app}(T_1 \cdot T_2, a) = \mathrm{Min}^{app}(T_1, a) + \mathrm{Min}^{app}(T_2, a)$. However, the type $(a[2..*] + \epsilon) \cdot (a[3..*] + \epsilon)$ only satisfies $a?[2..*]$: since a is optional on both sides, we consider here $\min(\mathrm{Min}^{app}(T_1, a), \mathrm{Min}^{app}(T_2, a))$ rather than their summation. Finally, $(a[m..*] + \epsilon) \cdot (a[n..*])$ satisfies $a?[n..*]$: since a is optional in the first subterm only, the result corresponds to n, that is, to $\mathrm{Min}^{app}(T_2, a)$. In the same way, while $a[3..*][4..*]$ satisfies $a?[12..*]$, a non-empty word of $(a[3..*] + \epsilon)[4..*]$ may have as few as 3 a's, hence $(a[3..*] + \epsilon)[4..*]$ only satisfies $a?[3..*]$: also in this case, $\mathrm{Min}^{app}(T[m..n], a)$ is not a function of $\mathrm{Min}^{app}(T, a)$ only.

We solve this problem by the mutually inductive computation of three different functions $\mathrm{Min}(T, a) \leq \mathrm{Min}^!(T, a) \leq \mathrm{Min}^{app}(T, a)$, where $\mathrm{Min}(T, a)$ computes the minimal value of $|w|_a$ over the whole T, $\mathrm{Min}^!(T, a)$ computes the minimal value of $|w|_a$ over $T!$, and $\mathrm{Min}^{app}(T, a)$ computes the minimal value of $|w|_a$ over the words of T where a appears.

The inductive computation of $\mathrm{Min}(T, a)$ is quite easy, apart from the case for $\mathrm{Min}(T!, a)$, where $\mathrm{Min}^!(T, a)$ comes handy (Lemma 3.22). The inductive computation of $\mathrm{Min}^{app}(T, a)$ is now also easy, even in the tricky cases of $T_1 \otimes T_2$ and $T[m..n]$, through a combined use of $\mathrm{Min}(T, a)$ and $\mathrm{Min}^{app}(T, a)$, as follows.

In the case of $T_1 \otimes T_2$, observe that any word of $T_1 \otimes T_2$ that contains a is built by combining either a word of T_1 that contains a with any word of T_2, or by combining a word of T_2 that contains a with any word of T_1. This gives us the following equation, that shows the correct way of combining summation and minimization when computing $\mathrm{Min}^{app}(T_1 \otimes T_2, a)$.

$$\mathrm{Min}^{app}(T_1 \otimes T_2, a) = \min(\mathrm{Min}^{app}(T_1, a) + \mathrm{Min}(T_2, a), \mathrm{Min}(T_1, a) + \mathrm{Min}^{app}(T_2, a))$$

In the case of $T[m..n]$, any word of $T[m..n]$ that contains a is built by combining one word of T that contains a with at least $m - 1$ words of T. This gives us the following equation.

$$\mathrm{Min}^{app}(T[m..n], a) = \mathrm{Min}^{app}(T, a) + (m - 1) \cdot \mathrm{Min}(T, a)$$

The function $\mathrm{Min}^!(\_, a)$ can be inductively computed from $\mathrm{Min}(\_, a)$ and $\mathrm{Min}^!(\_, a)$ in a similar way.

In this section, we define these three functions and show that they can be computed together with a single linear scan of the parse tree of T. The same property holds for $\mathrm{Max}(T, a)$, with no need of auxiliary functions.

Constraints extraction ZeroMinMax(U)

The third component $ZeroMinMax(U)$ of $\mathcal{C}(U)$ extracts a set of cardinality constraints with shape $a?[m..n]$. For each $a[m..n]$ that appears in U, it extracts the corresponding constraint $a?[m..n]$; a symbol a that is not subject to a counting operator is treated as $a[1..1]$. $ZeroMinMax(U)$ is defined as follows [1].

$$\begin{aligned} ZeroMinMax(U) &= \{a?[m..n] \mid a[m..n] \text{ is subterm of } U\} \\ &\quad \cup \{a?[1..1] \mid a \in sym(U) \text{ and } a[\_..\_] \text{ is not a subterm of } U\} \end{aligned}$$

Soundness of *ZeroMinMax*(U) depends on both the single-occurrence and the symbol-counting restrictions of conflict-free types. Consider a conflict-free U that contains a subterm $a\,[m..n]$: every a appearing in a word w of $[\![U]\!]$ is generated by that subterm, because of single-occurrence, and that subterm cannot be subject to any further counting operator, hence a will occur in w between m and n times. In the same way, if a subterm a appears in U and that a is not immediately included into a subterm $a\,[m..n]$, then no other counting operator may enclose that a, hence no word in $[\![U]\!]$ may contain more than one occurrence of a (for a formal proof see [1]).

Formal treatment

We begin with a formal definition of $\mathrm{Min}(T,a)$, $\mathrm{Min}^!(T,a)$ and $\mathrm{Min}^{app}(T,a)$. We first give a semantic definition of these functions (Definition 3.20), and will then show how to compute them (Lemma 3.22).

The semantics of the three functions is based on a common function $\mathrm{MinOrStar}(W,a)$, that corresponds to $\min_{w\in W}|w|_a$ when W is not empty, but yields $+\infty$, which we denote here as $*$, when W is empty. This usage of $+\infty$ to deal with empty sets is quite standard, since it ensures natural properties such as:

$$\mathrm{MinOrStar}(W \cup W', a) = \min(\mathrm{MinOrStar}(W,a), \mathrm{MinOrStar}(W',a))$$

Definition 3.20 ($\mathrm{Min}(T,a)$, $\mathrm{Min}^!(T,a)$, $\mathrm{Min}^{app}(T,a)$)**.** Let a be a symbol, W a set of words, and T a type. The functions $\mathrm{MinOrStar}(W,a)$, $\mathrm{Min}(T,a)$, $\mathrm{Min}^!(T,a)$, $\mathrm{Min}^{app}(T,a)$ are defined as follows.

$$\begin{array}{lll}
\mathrm{MinOrStar}(W,a) \overset{def}{=} \min_{w\in W}|w|_a & & \text{if } W \neq \emptyset \\
\mathrm{MinOrStar}(W,a) \overset{def}{=} * & & \text{if } W = \emptyset \\
\mathrm{Min}(T,a) & \overset{def}{=} \mathrm{MinOrStar}([\![T]\!], a) & \\
\mathrm{Min}^!(T,a) & \overset{def}{=} \mathrm{MinOrStar}(([\![T]\!] \setminus \{\epsilon\}), a) & \\
\mathrm{Min}^{app}(T,a) & \overset{def}{=} \mathrm{MinOrStar}(\{w \mid w \in [\![T]\!] \wedge w \models a^+\}, a) &
\end{array}$$

Observe that $\mathrm{Min}(T,a)$ can never be $*$, since $[\![T]\!]$ is never empty. On the other side, $\mathrm{Min}^{app}(T,a)$ can never be 0, since it only considers the words of T where a appears.

Proposition 3.21. *For any T, a, the following holds:*

$$\begin{array}{llll}
\mathrm{Min}(T,a) \neq * & & & (1) \\
\mathrm{Min}^!(T,a) = * & \Leftrightarrow & sym(T) = \emptyset & (2) \\
\mathrm{Min}^{app}(T,a) = * & \Leftrightarrow & a \notin sym(T) & (3) \\
\mathrm{Min}^{app}(T,a) \geq 1 & & & (4)
\end{array}$$

We can now show how these three functions can be computed by mutual induction. Note that, by our convention about $*$, all of $n + *$, $* + n$, $n \times *$, $* \times n$ denote here $*$.

Lemma 3.22.

$$\begin{array}{ll}
\mathrm{Min}(\epsilon,a) & = 0 \\
\mathrm{Min}(a,a) & = 1 \\
\mathrm{Min}(b,a) & = 0 \textit{ if } b \neq a \\
\mathrm{Min}(T_1+T_2,a) & = \min(\mathrm{Min}(T_1,a), \mathrm{Min}(T_2,a)) \\
\mathrm{Min}(T_1 \otimes T_2,a) & = \mathrm{Min}(T_1,a) + \mathrm{Min}(T_2,a) \\
\mathrm{Min}(T\,[m..n]\,,a) & = m \cdot \mathrm{Min}(T,a) \\
\mathrm{Min}(T!,a) & = \mathrm{Min}^!(T,a) \\
\mathrm{Min}^!(\epsilon,a) & = * \\
\mathrm{Min}^!(a,a) & = 1 \\
\mathrm{Min}^!(b,a) & = 0 \textit{ if } b \neq a \\
\mathrm{Min}^!(T_1+T_2,a) & = \min(\mathrm{Min}^!(T_1,a), \mathrm{Min}^!(T_2,a)) \\
\mathrm{Min}^!(T_1 \otimes T_2,a) & = \min(\mathrm{Min}^!(T_1,a) + \mathrm{Min}(T_2,a),\ \mathrm{Min}(T_1,a) + \mathrm{Min}^!(T_2,a)) \\
\mathrm{Min}^!(T\,[m..n]\,,a) & = \mathrm{Min}^!(T,a) + (m-1) \cdot \mathrm{Min}(T,a) \\
\mathrm{Min}^!(T!,a) & = \mathrm{Min}^!(T,a) \\
\mathrm{Min}^{app}(\epsilon,a) & = * \\
\mathrm{Min}^{app}(a,a) & = 1 \\
\mathrm{Min}^{app}(b,a) & = * \textit{ if } b \neq a \\
\mathrm{Min}^{app}(T_1+T_2,a) & = \min(\mathrm{Min}^{app}(T_1,a), \mathrm{Min}^{app}(T_2,a)) \\
\mathrm{Min}^{app}(T_1 \otimes T_2,a) & = \min(\mathrm{Min}^{app}(T_1,a) + \mathrm{Min}(T_2,a),\ \mathrm{Min}(T_1,a) + \mathrm{Min}^{app}(T_2,a)) \\
\mathrm{Min}^{app}(T\,[m..n]\,,a) & = \mathrm{Min}^{app}(T,a) + (m-1) \cdot \mathrm{Min}(T,a) \\
\mathrm{Min}^{app}(T!,a) & = \mathrm{Min}^{app}(T,a)
\end{array}$$

```
CardImplies(Type $T$, Type $U$):
  every $a\,[m..n]$ in $U$ satisfy let $(\_, \_, MinA, Max) = MinBangAppMax(T, a)$;
                                  in $MinA \geq m \wedge Max \leq n$
  and every $a$ in $U$ where $a\,[m..n]$ not in $U$
                          satisfy let $(\_, \_, \_, Max) = MinBangAppMax(T, a)$;
                                  in $Max \leq 1$

MinBangAppMax(Type $T$, Symbol $a$):
  case $T$
  when $T_1 \otimes T_2$:   let $(M_1, B_1, A_1, Max_1) = MinBangAppMax(T_1, a)$;
                 let $(M_2, B_2, A_2, Max_2) = MinBangAppMax(T_2, a)$;
                 return($M_1 + M_2$,
                        $\min(B_1 + M_2, M_1 + B_2)$,
                        $\min(A_1 + M_2, M_1 + A_2)$,
                        $\max(Max_1, Max_2)$);
  when $T_1 + T_2$:   let $(M_1, B_1, A_1, Max_1) = MinBangAppMax(T_1, a)$;
                 let $(M_2, B_2, A_2, Max_2) = MinBangAppMax(T_2, a)$;
                 return($\min(M_1, M_2)$, $\min(B_1, B_2)$, $\min(A_1, A_2)$,
                        $\max(Max_1, Max_2)$);
  when $T_1\,[m..n]$:  let $(M_1, B_1, A_1, Max_1) = MinBangAppMax(T_1, a)$;
                 return($m \cdot M_1, B_1 + (m-1) \cdot M_1, A_1 + (m-1) \cdot M_1, n \cdot Max_1$);
  when $T_1!$:        let $(\_, B_1, A_1, Max_1) = MinBangAppMax(T_1, a)$;
                 return($B_1, B_1, A_1, Max_1$);
  when $\epsilon$:         return(0, *, *, 0)
  when $a$:         return(1, 1, 1, 1)
  when $b \neq a$:     return(0, 0, *, 0)
```

Fig. 4. Algorithm for implication of cardinality constraints.

Proof. See Appendix. □

The upper bound is much easier, and is defined and computed as follows.

Definition 3.23 (Max(T, a))**.**

$$\begin{aligned} \mathrm{Max}(T, a) &\overset{def}{=} \max_{w \in [\![T]\!]} |w|_a && \text{if } (\max_{w \in [\![T]\!]} |w|_a) \in \mathbb{N} \\ \mathrm{Max}(T, a) &\overset{def}{=} * && \text{if } \forall n \in \mathbb{N}.\ \exists w \in [\![T]\!].\ |w|_a > n \end{aligned}$$

Lemma 3.24.

$$\begin{aligned} \mathrm{Max}(\epsilon, a) &= 0 \\ \mathrm{Max}(a, a) &= 1 \\ \mathrm{Max}(b, a) &= 0 \textit{ if } b \neq a \\ \mathrm{Max}(T_1 + T_2, a) &= \max(\mathrm{Max}(T_1, a), \mathrm{Max}(T_2, a)) \\ \mathrm{Max}(T_1 \otimes T_2, a) &= \mathrm{Max}(T_1, a) + \mathrm{Max}(T_2, a) \\ \mathrm{Max}(T\,[m..n]\,, a) &= n \cdot \mathrm{Max}(T, a) \\ \mathrm{Max}(T!, a) &= \mathrm{Max}(T, a) \end{aligned}$$

By the definition of $\mathrm{Min}^{app}(T, a)$ and $\mathrm{Max}(T, a)$, cardinality constraint satisfaction can be decided as follows.

Corollary 3.25.

$$T \models a?[m..n] \Leftrightarrow m \leq \mathrm{Min}^{app}(T, a) \ \wedge \ \mathrm{Max}(T, a) \leq n$$

The algorithm

We can now introduce the algorithm that we use to verify that a general type T satisfies every F in $ZeroMinMax(U)$. It is listed in Fig. 4; the function *MinBangAppMax* computes, in one pass, and in linear time, the values of $\mathrm{Min}(T, a)$, $\mathrm{Min}^!(T, a)$, $\mathrm{Min}^{app}(T, a)$ and $\mathrm{Max}(T, a)$. The values of $\mathrm{Min}^{app}(T, a)$ and $\mathrm{Max}(T, a)$ are then used to verify the constraint satisfaction.

MinBangAppMax(T, a) can be computed in time $O(|T|)$, since it performs constant-time operations for each node of T. *CardImplies* invokes it on T once for each symbol of U, hence *CardImplies* can be computed in time $O(|U| \times |T|)$.

3.4. Lower bounds and upper bounds — *upperS*(U) and *SIf*(U)

The two last components of $\mathcal{C}(U)$ are the lower bound and upper bound components, called $SIf(U)$ and $upperS(U)$, which are defined, in [1], as follows.

$$\text{Lower-bound:} \quad SIf(U) \overset{def}{=} \text{if } \neg \mathrm{N}(U) \text{ then } \{sym(U)^+\} \text{ else } \emptyset$$

$$\text{Upper-bound:} \quad upperS(U) \overset{def}{=} \{\mathrm{upper}(sym(U))\}$$

Notice that the problem of constraint verification is simplified by verifying the implication of lower and upper bounds at the same time: by restricting ourselves to the case when $T \models upperS(U)$, we do not need to check whether $T \models SIf(U)$, but we only have to check that $\mathrm{N}(T) \Rightarrow \mathrm{N}(U)$, as proved below.

Theorem 3.26 (*Implication of* $SIf(T_2)$ *and* $upperS(T_2)$)**.** *For any two types* T_1 *and* T_2*:*

$$T_1 \models SIf(T_2) \cup upperS(T_2) \;\Leftrightarrow\; (\mathrm{N}(T_1) \Rightarrow \mathrm{N}(T_2)) \wedge sym(T_1) \subseteq sym(T_2)$$

Proof. $(\Rightarrow)$ $T_1 \models upperS(T_2)$ means that

$$\neg \exists a, w.\ (a \in sym(w) \wedge w \in [\![T_1]\!] \wedge a \notin sym(T_2))$$

hence

$$\forall a, w.\ (a \in sym(w) \wedge w \in [\![T_1]\!]) \Rightarrow a \in sym(T_2)$$

hence

$$\forall a.\ (\exists w.\ a \in sym(w) \wedge w \in [\![T_1]\!]) \Rightarrow a \in sym(T_2)$$

hence, by Lemma 2.8(2),

$$\forall a.\ a \in sym(T_1) \Rightarrow a \in sym(T_2)$$

We prove now that $\neg\mathrm{N}(T_2) \Rightarrow \neg\mathrm{N}(T_1)$. Assume $\neg\mathrm{N}(T_2)$; then $T_1 \models SIf(T_2)$ means $T_1 \models sym(T_2)^+$, hence $\epsilon \notin [\![T_1]\!]$, hence $\neg\mathrm{N}(T_1)$.

$(\Leftarrow)$ By Lemma 2.8(2), direction $\Leftarrow$, and by definition of $upperS(\_)$, we have that $T_1 \models upperS(T_1)$, from which the implication $sym(T_1) \subseteq sym(T_2) \Rightarrow T_1 \models upperS(T_2)$ follows. If $\mathrm{N}(T_2)$ is true, then $T_1 \models SIf(T_2)$ holds trivially. If $\neg\mathrm{N}(T_2)$, then, by $\mathrm{N}(T_1) \Rightarrow \mathrm{N}(T_2)$, $\mathrm{N}(T_1)$ is false as well, hence every word of T_1 contains a symbol from $sym(T_1)$, hence a symbol from $sym(T_2)$. □

Hence, the function *UpperLowerImplies*(T, U) that verifies whether $T \models SIf(U) \cup upperS(U)$ will just check whether $(\mathrm{N}(T) \Rightarrow \mathrm{N}(U)) \wedge sym(T) \subseteq sym(U)$, which can be trivially done in time $O(|T| + |U|)$. The algorithm is quite obvious, hence we provide no pseudocode.

3.5. Summing up

We have recalled each of the five components of the constraint-extraction function $\mathcal{C}(U)$ defined in [1] and, for each component $\mathcal{C}_i$, we defined a function that verifies, for any general T, whether $T \models \mathcal{C}_i(U)$. Since the union of these five components is exact for conflict-free types [1], the following theorem holds.

Theorem 3.27. *For any type T, for any conflict-free type U, $[\![T]\!] \subseteq [\![U]\!]$ iff all of CoImplies(T, U), OrderImplies(T, U), CardImplies(T, U), and UpperLowerImplies(T, U) return* ***true****.*

CoImplies, *OrderImplies*, and *CardImplies* have quadratic time-complexity, while *UpperLowerImplies*, which checks both lower and upper constraints, is linear, hence the algorithm is quadratic.

The algorithm that we described in [1], for the much simpler case when the subtype is conflict-free, is also quadratic. Quite surprisingly, despite the much higher expressive power of general extended REs with respect to conflict-free types, the only case whose complexity is affected by the presence of general types in the subtype position is that of cardinality constraints. Satisfaction of cardinality constraints can be checked in linear time when two conflict-free types are compared, while here the presence of multiple occurrences of a symbol and the nesting of $T[m..n]$ operators both concur in making the problem slightly harder to solve, forcing us to adopt a quadratic algorithm for the *CardImplies* case.

4. Experimental evaluation

To validate our claim of efficiency we present here an experimental evaluation of our algorithm. In particular, we first study its scalability properties, and then compare its performance with that of a subtyping algorithm based on Brzozowski's derivatives [11]. Our experiments have been performed on a random sample of subtype–supertype pairs, hence we also discuss here the problem of generating significant test samples.

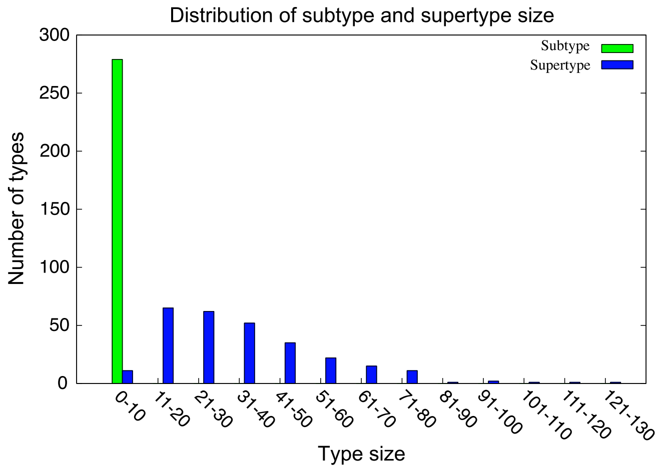

Fig. 5. Distribution of subtype and supertype sizes.

4.1. *Experimental setup*

We implemented the algorithms being tested in Java 1.6, and evaluated their performance on a 2.53 GHz Intel Core 2 Duo machine with 4 GB of main memory and running Mac OSX 10.6.8. 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.

4.2. *Sample generation*

We perform our experiments on randomly generated type pairs. The problem of generating random samples for subtyping is not trivial: when a random pair of types is generated, even if they share the same alphabet, the probability that one of the two types is a subtype of the other is extremely low. Hence, a set of random pairs would mostly test the algorithm behaviour in the negative case. However, when a compiler checks a piece of code, the vast majority of the input pairs satisfies the subtype relation, hence the positive case is the one we would really like to measure. (The problem of generating a pair that is subtype-related is similar but complementary to those studied by Antonopoulos et al. in [14], where the focus is on generating, counting, and sampling regular tree languages.)

In our analysis we explored and used two different schemes in order to generate random samples of subtype-related pairs. The first approach we consider here is the simplest one:

(i) generating a random conflict-free type U;
(ii) generating a random type T in RE(#, &) (the class of unrestricted REs with interleaving and counting), forcing its alphabet to be included in that of U;
(iii) discarding the pair if the two types are not related by subtyping.

To implement this approach, we developed two random generators, which receive in input the expected depth of the type being generated and the probability distribution of operators (we used in both cases the uniform distribution). At each step, the generators choose a type operator and, when the actual depth of the generated type reaches the expected depth, they generate leaf nodes only.

This approach proved to be totally unsatisfactory regarding both the quality and the quantity of subtype-related pairs. We report here the results of a generation experiment, where we generate 1,000,000 pairs. In this collection, only 279 pairs are in the subtype–supertype relation (0.0279%). Furthermore, as shown in Fig. 5, the generated subtypes are very small with respect to the supertype: indeed, the size of all of the 279 subtypes is less than 10 nodes, while supertype sizes have a much better distribution. Given these results, we used this approach for generating samples for negative tests only.

The second scheme we analyze is more complex and is specifically designed for generating pairs satisfying the inclusion relation. In order to better exploit the algebraic properties of the binary operators, our generators represent types in n-ary form rather than in binary form; hence, $T_1 \circledast \cdots \circledast T_n$ is used to represent $(((T_1 \circledast T_2) \circledast \cdots \circledast T_n)$. Furthermore, $T\,[0..*]$ is also used by the generator, and it denotes $T\,[1..*] + \epsilon$.

We first observe that the pairs of types that are examined by a compiler are not really random pairs, since the subtype corresponds to an expression that is written by a programmer who is aware of the expected supertype, and aims to write an expression that matches that type. As a consequence, it will often be the case that the subtype bears some syntactic relationship, loose or strict, with the supertype, rather than being a random type expression that luckily happens to denote a subset of the supertype.

This observation inspires the following algorithm: we first randomly generate a conflict-free supertype U, and then apply a set of probabilistic rewriting rules that output a type T such that $T \leq U$. Our rewrite rules, unfortunately, produce a type T where counting is only applied to its leaves, which is hardly acceptable. For this reason, we add a third step, where we apply a further transformation to T that lifts counting operators from the leaves to the intermediate nodes, hence returning a subtype $T' \leq T \leq U$. The pair (T', U) is the result of our subtype generation algorithm.

In detail, the type rewriter applies the following rules:

1. if $U = U_1 + \cdots + U_n$, each U_i is recursively transformed into a union $T_{i1} + \cdots + T_{im_i}$, where each T_{ij} is generated from the corresponding U_i (each set $\{T_{ij}\}^{j \in 1..m_i}$ may be empty); a random permutation is applied to the result; since any U_i generates a set of T_{ij}, any symbol from U_i may be repeated many times in the generated subtype, or may not appear in the subtype at all; observe that, when the set $\cup_{i \in 1..n}\{T_{ij}\}^{j \in 1..m_i}$ is a singleton, then the subtype of the union type U may not be a union type; in general, our rewrite rules may always generate a subtype whose outermost operator is not the same as that of the supertype;
2. if $U = U_1 \cdot \ldots \cdot U_n$ and U is not nullable, we generate a product type $T = T_1 \cdot \ldots \cdot T_n$, where $T_i \leq U_i$; if U_i is nullable, T_i may be ϵ;
3. if $U = U_1 \cdot \ldots \cdot U_n$ and U is nullable, we either generate a product type $T = T_1 \cdot \ldots \cdot T_n$, as for the previous rule, or transform U into $U_1 + \cdots + U_n$ and recursively apply the first rule;
4. as in the previous two cases, if $U = U_1 \& \ldots \& U_n$ is not nullable, we generate a product subtype, while, when U is nullable, we generate either a product subtype or a sum subtype. In both cases, when the generated subtype is a product, we randomly choose between a subtype $T = T_1 \& \ldots \& T_n$ and a subtype $T = T_1 \cdot \ldots \cdot T_n$; in all cases, a random permutation is applied to the result;
5. if $U = a\,[m..n]$, we return $T = a\,[p..q]$, where p and q are randomly generated, so that $m \leq p \leq q \leq n$; in the special case when both $p = 0$ and $q = 0$, the resulting type $U = a\,[0..0]$ is just ϵ. In greater detail, we proceed as follows:
   - if $U = a\,[m..n]$ with $n \neq *$, we randomly generate two numbers r and s uniformly distributed in $[m..n]$, and return $a\,[\min(r, s)..\max(r, s)]$;
   - if $U = a\,[m..*]$, we first randomly decide whether we will generate an open interval $T = a\,[r..*]$ or a closed interval $T = a\,[r..s]$. In the first case, we generate an offset i using a Poisson distribution with average m, and return $a\,[m + i..*]$. In the second case, we generate i with the same Poisson distribution, we randomly generate two numbers r and s uniformly distributed in $[m..m + i]$, and return $a\,[\min(r, s)..\max(r, s)]$.

After the type rewriting phase, we get a type $T \leq U$, where labels may be repeated, thanks to rewritings of cases (1), (3) and (4), but counting is still confined in the leaf nodes of T, as it was in U. To overcome this issue, we add a third step, that recursively applies the rewriting rule $T_1\,[m_1..n_1]\,\&T_2\,[m_2..n_2] \rightarrow (T_1\&T_2)\left[m'..n'\right]$, where $[m'..n'] = [m_1..n_1] \cap [m_2..n_2]$; if $[m_1..n_1]$ and $[m_2..n_2]$ have empty intersection, the rule is not applied.

This approach has the advantage of producing couples of types that are in the subtype relation and whose sizes are comparable. This approach generates T starting from U, hence T is not only a subtype of U, but it also bears a structural relation with U, although this relation is not very strong, given the extensive set of rewriting rules that we apply. The existence of this structural relation does not affect the behaviour of our algorithm, since it works on the constraints that are extracted from the two types, without taking any shortcut in cases of high similarity.

4.3. Derivative-based algorithm

We would have liked to compare our algorithm with some established competitor, but no other algorithm for inclusion of regular expressions with interleaving and counting has been experimentally evaluated, to our knowledge. Hence we decided to choose the most promising alternative algorithm among those presented in the literature (see Section 5), and to implement it ourselves. We did not consider the classical algorithm based on automata complement and intersection, since the automata that have been studied for interleaving and counting do not behave well under complement and intersection [7,15], and no such automaton has been defined to take advantage of the specific limitations of conflict-free types. We use instead the algorithmic scheme described by Chen and Chen in [16]. The scheme is based on Brzozowski derivatives, which are very well behaved for deterministic types, so that the technique is well suited to work with conflict-free supertypes. While the worst-time complexity of this algorithm is still very high, as happens with all algorithms that have been defined for subtype inclusion in presence of interleaving and counting, the average behaviour of this scheme seems interesting.

The notion of Brzozowski's derivative is standard, and is defined as follows (see [17]).

Definition 4.1 (*Derivative*). T_1 is a derivative of T according to a iff $\{a\} \cdot [\![T_1]\!] = [\![T]\!]$.

The algorithmic scheme is reported in Fig. 6. In this figure, $d_a(T)$, defined in Definition 4.5, is a Brzozowski's derivative of a type T by a symbol $a \in \Sigma$, and $first(T)$ is the set of the first symbols of all words accepted by T.

The algorithm checks that all derivatives of T are included in those of U and stores in M the pairs that have already been met. It first verifies that $\epsilon \in T$ implies $\epsilon \in U$ and that $first(T) \subseteq first(U)$. If this is the case, the current pair is added to M, the global data structure that contains all the already-met pairs, which is initially empty and grows at each call of *DerivInclude*. At this point, *DerivInclude* recursively verifies that, for every symbol a that is accepted by T, its derivative is included in the

```
global Set $M := \emptyset$;
DerivInclude(Type $T$, Type $U$):
  if (N($T$) and not N($U$)) then return false
  if ($first(T) \not\subseteq first(U)$) then return false
  $M := M \cup \{(T, U)\}$
  for each $a \in first(T)$ do
     $T_1 = d_a(T)$
     $U_1 = d_a(U)$
     if$(T_1, U_1) \notin M$
     then if DerivInclude($T_1, U_1$) = false
          then return false
  od
  return true
```

Fig. 6. Skeleton of the derivative-based inclusion algorithm.

corresponding derivative of U. When the same pair is met for the second time, it is ignored. The algorithm is quite natural; for a proof of correctness, see [16].

This algorithm was originally designed for the symmetric inclusion of 1-unambiguous regular expressions without interleaving and counting. To adapt this algorithm to our context, we extended Brzozowski's derivatives to conflict-free types and unrestricted regular expressions with interleaving and counting (see also [18]). To this aim, we first extend our type language with the *empty* expression $\emptyset$. Moreover, we relax the well-formation constraints of Definition 2.4: we allow 0 to appear in both the m and n positions of $T\,[m..n]$, with the obvious semantics (specifically, $[\![T\,[0..0]]\!] = [\![\epsilon]\!]$), and we allow expressions $T!$ where $sym(T) = \emptyset$: the semantics of these expressions, according to Definition 2.5, is just an empty set. Hereafter we will use RE(#, &, 0) to denote the class RE(#, &) extended with empty types, 0 bounds and unrestricted use of _!, and cf-RE(#, &, 0) for the class of conflict-free expressions, extended in the same way. We use this wider class since it greatly simplifies the definition of derivatives, and a subtyping algorithm for this wider class can of course be used for RE(#, &) and cf-RE(#, &).

Definition 4.2 (*Empty Expression*)**.** $\emptyset$ denotes the *empty* regular expression, that is, $[\![\emptyset]\!] \stackrel{def}{=} \emptyset$.

Proposition 4.3 (*Empty Expression Properties*)**.** *$\emptyset$ satisfies the following properties:*

$$\begin{array}{lll} T + \emptyset = \emptyset + T & = T \\ T \cdot \emptyset = \emptyset \cdot T & = \emptyset \\ T\&\emptyset = \emptyset\&T & = \emptyset \end{array}$$

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

Notation 4.4 (m^-, $* - 1$)**.** In the following definitions, we use m^- to denote $max(m-1, 0)$, and assume that $* - 1 = *$.

Definition 4.5 ($d_a(U)$)**.** The function $d_a(U)$, where U is in cf-RE(#, &, 0) and a is a symbol, is defined as follows:

$$\begin{array}{ll}
d_a(\epsilon) & \stackrel{def}{=} \emptyset \\
d_a(\emptyset) & \stackrel{def}{=} \emptyset \\
d_a(T!) & \stackrel{def}{=} d_a(T) \\
d_a(b\,[m..n]) & \stackrel{def}{=} \begin{cases} \emptyset & \text{if } a \neq b \text{ or } n \leq 1 \\ b\left[m^-..n-1\right] & \text{otherwise} \end{cases} \\
d_a(U_1 + U_2) & \stackrel{def}{=} \begin{cases} d_a(U_1) & \text{if } a \in first(U_1) \\ d_a(U_2) & \text{if } a \in first(U_2) \\ \emptyset & \text{otherwise} \end{cases} \\
d_a(U_1 \cdot U_2) & \stackrel{def}{=} \begin{cases} d_a(U_1) \cdot U_2 & \text{if } a \in first(U_1) \\ d_a(U_2) & \text{if } a \in first(U_2) \text{ and } \mathrm{N}(U_1) \\ \emptyset & \text{otherwise} \end{cases} \\
d_a(U_1 \& U_2) & \stackrel{def}{=} \begin{cases} d_a(U_1)\&U_2 & \text{if } a \in sym(U_1) \\ d_a(U_2)\&U_1 & \text{if } a \in sym(U_2) \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

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 derivatives of U: $D(U) \overset{def}{=} \{d_w(U) \mid w \in \Sigma^*\}$. As it can be easily observed, $D(U\,[m..n])$ has a size that grows linearly with m, hence is exponential in $|U\,[m..n]\,|$, although each element of $D(U)$ is smaller than U. The derivatives of general expressions are even less well-behaved, as Brzozowski's derivation for non strongly-deterministic types may generate exponentially larger derivatives, as shown by the following definition.

Definition 4.6 (*Derivation for RE(#, &, 0)*)**.** $d_a(T)$, where T is an unrestricted regular expression in RE(#, &, 0) and a is a symbol, is defined as follows:

$$\begin{array}{lll}
d_a(\epsilon) & \overset{def}{=} & \emptyset \\
d_a(\emptyset) & \overset{def}{=} & \emptyset \\
d_a(b) & \overset{def}{=} & \begin{cases} \epsilon & \text{if } a = b \\ \emptyset & \text{otherwise} \end{cases} \\
d_a(T!) & \overset{def}{=} & d_a(T) \\
d_a(T\,[m..n]) & \overset{def}{=} & \begin{cases} \emptyset & \text{if } n = 0 \\ d_a(T)\cdot T\left[m^-..n-1\right] & \text{if (not N}(T) \text{ and } n > 0) \\ & \text{ or } (m = 0 \text{ and } n = *) \\ d_a(T)\cdot T\left[m^-..n-1\right] & \\ \quad + d_a\left(T\left[m^-..n-1\right]\right) & \text{otherwise} \end{cases} \\
d_a(T_1 + T_2) & \overset{def}{=} & d_a(T_1) + d_a(T_2) \\
d_a(T_1 \cdot T_2) & \overset{def}{=} & \begin{cases} d_a(T_1)\cdot T_2 + d_a(T_2) & \text{if N}(T_1) \\ d_a(T_1)\cdot T_2 & \text{otherwise} \end{cases} \\
d_a(T_1 \& T_2) & \overset{def}{=} & d_a(T_1)\& T_2 + T_1 \& d_a(T_2)
\end{array}$$

The algorithm tests $T \leq U$ by generating up to $|D(T)| \cdot |D(U)|$ pairs, which gives this algorithm a worst-case exponential complexity.

To ensure the termination of the derivation process for non-deterministic regular expressions, our derivation algorithm works modulo associativity and commutativity of union. In detail, derivation is implemented as follows:

- by memoizing the derivation process, as some types can be derived many times by the algorithm;
- by flattening memoized derivatives, so to make the order and the associativity of addenda irrelevant;
- and by systematically simplifying derivatives through the following rules: $T + T \to T, T + \emptyset \to T, T \cdot \epsilon \to T, T \cdot \emptyset \to \emptyset$.

This strategy not only ensures the termination of the derivation process, but also brings great benefits to the overall behaviour of our implementation of this algorithm.

4.4. Experimental results

In our first experiment we evaluate the scalability of our quadratic algorithm on a sample of 50,000 positive randomly generated subtype–supertype pairs. We use as input size the sum of the number of nodes of both the supertype and the subtype, and measure the time required for completing the inclusion checking. The results we obtained, for input sizes up to 1000 nodes and up to 2500 nodes, are shown in Figs. 7 and 8.

As it can be observed, the points in the graphs lay between two quadratic curves, corresponding to the best and the worst cases. Even in the worst cases, the algorithm is quite fast and efficiently processes large input types.

In our second experiment battery we compare the performance of our algorithm with that of the derivative-based algorithm on both a positive and a negative sample.

The results of the experiment on the positive sample are shown in Figs. 9 and 10. We use here a smaller sample consisting of 20,000 pairs with maximum size about 250, as the derivative-based algorithm proved to be very slow on types exceeding this size threshold. Fig. 9 shows that the quadratic algorithm always outperforms the derivative-based one. This is even better illustrated in Fig. 10, where we used a logarithmic scale for the y-axis; this graph clearly shows that, in our experimental range, our algorithm is several orders of magnitude faster than the derivative-based one.

It can also be noted that the derivative-based algorithm is very sensitive to the structure of the types being compared, which, instead, has little influence on the constraint-based one. Hence, the performance of the derivative-based algorithm is quite erratic and unstable, while the performance of our algorithm is very stable and predictable.

The results of our experiment on a negative sample are shown in Fig. 11. As in the previous experiment, we considered here a sample of 20,000 pairs with maximum size around 250 nodes. In this case, the performances of the algorithms are much closer, as the derivative-based algorithm tries to detect failure conditions as soon as possible. The derivative-based

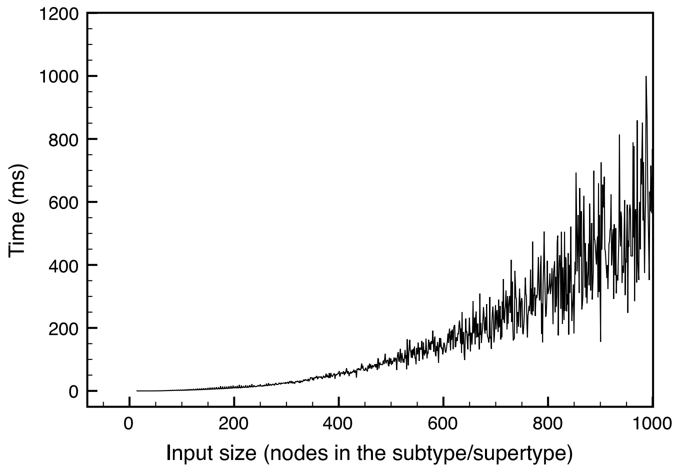

Fig. 7. Scalability of the quadratic inclusion algorithm: input size ⩽1000.

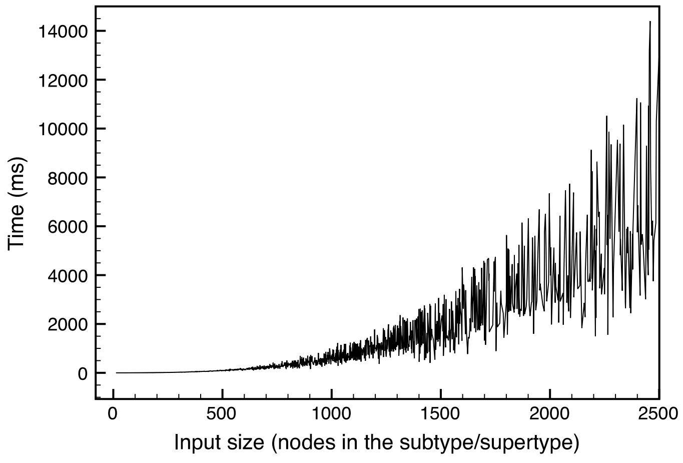

Fig. 8. Scalability of the quadratic inclusion algorithm: input size ⩽ 2500.

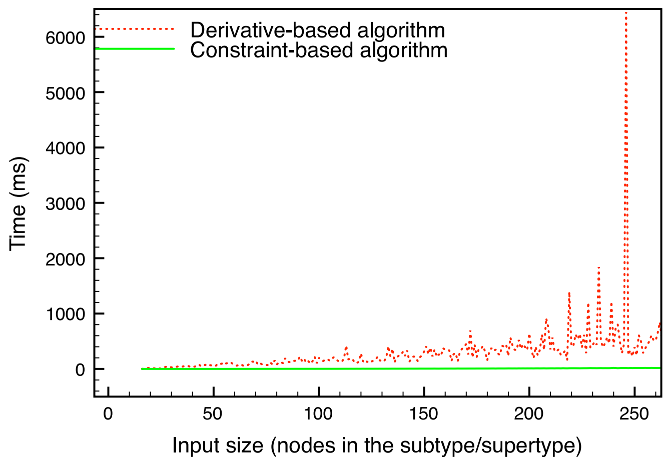

Fig. 9. Positive sample experiment.

algorithm has a worst performance on very small types, which is counterintuitive. This happens because the derivative-based algorithm performs, in a sense, a breadth-first exploration of the two compared types, looking for a reason to fail that is easy to spot, and smaller types tend to have a smaller amount of such "fast exits".

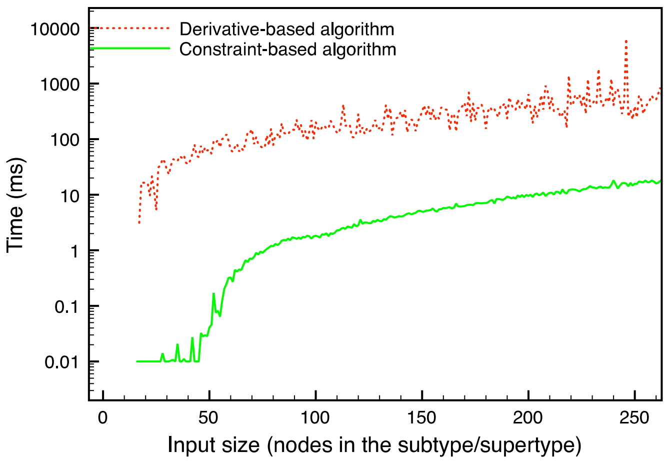

Fig. 10. Positive sample experiment: logarithmic scale.

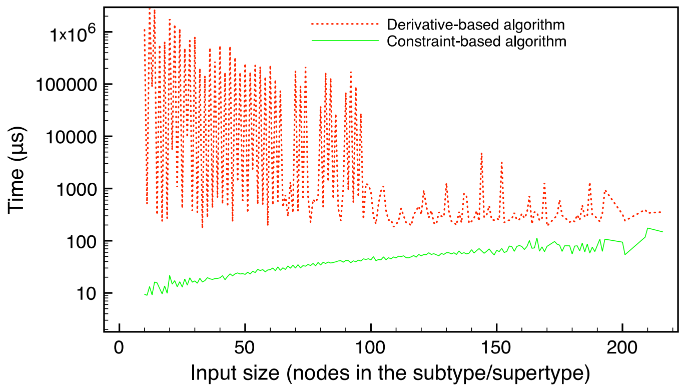

Fig. 11. Negative sample experiment: logarithmic scale.

We performed this negative-case test just to verify whether these algorithms have an acceptable performance also in this situation, which is actually true for both of them. Apart from this, the negative-case comparison has little practical interest, since positive checks are largely dominant in a typical typechecking workload.

To summarize, these experiments show that our quadratic algorithm behaves much better than its most direct competitor. They also show that it is reliably fast on sizeable types, hence it represents a viable option for the construction of a practical use typechecker in a language with interleaving and counting.

5. Related work

5.1. Some flavours of determinism

Membership testing for full REs with interleaving and counting is NP-hard [6], hence extended languages meant for practical use are usually endowed with some restrictions, aimed to reduce membership complexity. These restrictions are typically designed to allow for the efficient construction of a compact deterministic automaton, and we introduce them here, since we need these notions in order to discuss the literature about RE inclusion.

A typical restriction is 1-*unambiguity*, that means (informally) that, when a string is analyzed, any analyzed character can be matched against one specific character in the regular expression, that is determined by the part of string that has been read so far. For example, $(a \cdot b)^{+}a$ is 1-unambiguous, but $(a?b)^{*}a$ is not: while reading $ba\ldots$, we do not know whether a should be matched against the first a or the second one.

The *single-occurrence* restriction, meaning that no character occurs twice in an expression, trivially implies 1-unambiguity.

Strong determinism is another constraint stronger than 1-unambiguity, having to do with Kleene-star and with counting. Consider the expression $(a\,[1..2])\,[2..3]$. While reading $aa\ldots$, we do not know whether the second a matches the second repetition of a in $a\,[1..2]$, or whether we should match the whole $a\,[1..2]$ with the first a, and the second a with the first character of the second repetition of $a\,[1..2]$. Strong determinism means, very informally, that the part of string that has already been read and the current character determine both the next leaf to match and which counting operator (or Kleene star) is affected (see [19] for a formal definition).

Single-occurrence and strong determinism both imply 1-unambiguity, but none is stronger than the other one. *Conflict-freedom*, as defined in this paper, implies both. It implies single-occurrence by definition. It also implies strong determinism: since in a conflict-free type the content of a counting operator is just one character, there is no ambiguity about the effect of each character on the only counting operator that may contain it.

Conflict-freedom is very restrictive, but is trivial to define and check. The precise definition and automated checking of 1-unambiguity and strong determinism are a bit less trivial. In [19], cubic time algorithms to test for 1-unambiguity and strong determinism are presented. In [20], Kilpeläinen presents a $O(n^2/log(n))$ algorithm to test whether a RE with counting is 1-unambiguous, and describes how some well-known studies and implementations of the same notion are actually incorrect.

5.2. Inclusion of regular expressions with interleaving and counting

The problem of inclusion of regular expressions with interleaving has been studied in many papers, but none of them provides PTIME inclusion algorithms for languages with interleaving, counting, and an expressive power that is acceptable for our intended application.

In [6], Mayer and Stockmeyer studied the complexity of membership, inclusion, and inequality for several classes of regular expressions with interleaving and intersection. In particular, interleaving is proved to make inclusion EXPSPACE-complete.

Starting from the results of [6], Gelade et al. [7] studied the complexity of decision problems for DTDs, single-type EDTDs, and EDTDs with interleaving and counting. By considering several classes of regular expressions with interleaving and counting, they showed that their inclusion is almost invariably EXPSPACE-complete, even when counting is restricted to terminal symbols only; they also showed how these results extend to various kinds of schemas for XML documents. We did not discuss here how to extend our results from REs to XML schema languages because the problem is indeed solved in [7], where it is shown how an inclusion algorithm for REs can be lifted to schema languages that use that class of REs without changing the complexity class. In [21,22] Kilpeläinen and Tuhkanen proved that inclusion is coNP-hard for regular expressions with counting even if attention is restricted to 1-unambiguous REs and without interleaving.

The properties of a commutative type language for XML data have been discussed by Foster et al. in [23]. Here, the authors essentially described the techniques they used while implementing a type-checker for commutative XML types. Their type language resembles our language of conflict-free types, as repetition types can be applied to element types only, and interleaving is supported. The paper is focused on heuristics that improve scalability, but do not affect computational complexity.

In [1], previously published as [24], we defined a polynomial time algorithm for inclusion of conflict-free types, but we were not able to extend the result to reach any more general class. In that paper, we specified the constraint extraction procedure that we use here, and we proved that it is exact for conflict-free types. The specific contribution of this paper is the extension of those techniques from the case when the subtype is conflict-free to the general, asymmetric, case when the subtype is any RE with interleaving and counting.

None of these papers presents a viable algorithm to test inclusion of regular expressions with interleaving and counting. However, in [16], Chen and Chen describe an algorithm for the symmetric inclusion of 1-unambiguous regular expressions without interleaving and counting, based on Brzozowski derivatives, which can be easily extended to our problem, and, despite being exponential in the worst case, gives reasons to hope in an acceptable behaviour in practice. For these reasons, we choose this algorithm for our comparison in Section 4.

5.3. Inclusion of XML types

XML Schema [4] and RELAX-NG [5] are two well-known type languages that allow some form of interleaving and counting.

XML Schema is based on REs with counting, plus an extremely limited form of interleaving: the *all* group, that only allows symbols to be interleaved. XML Schema adopts a constraint known as *Unique Particle Attribution* (UPA) ([25], Section 3.6.6). There is some debate about the actual meaning of that constraint, but it is usually interpreted as a way to require 1-unambiguity [22,19].[6] The coNP-hard problem presented for 1-unambiguous REs with counting in [21,22] can be easily expressed by a 1-unambiguous XML Schema, hence XML Schema inclusion is coNP-hard.

RELAX-NG [5] is based on REs extended with interleaving. RELAX-NG does not impose any form of unambiguity in general, with the only exception of interleaving: in any occurrence of $E_1 \& \ldots \& E_n$, the first characters recognized by the E_i expressions must be all mutually disjoint. RELAX-NG restricts the use of interleaving ([5], Section 7.4) and has no counting. However, it does not restrict the expressions that use no interleaving, hence inclusion for RELAX-NG is PSPACE-hard [26].

[6] This constraint serves the purpose of implementing linear time membership algorithms that do not require lookahead or backtracking.

All the works we discussed up to now deal with symmetric inclusion. Asymmetric inclusion of REs or of XML types has also been studied elsewhere in the recent past. We discuss some of these papers here, but they are not very relevant to our problem since they deal with languages without interleaving and without counting. In [27] Colazzo and Sartiani showed that complexity of RE inclusion can be lowered from EXPSPACE to EXPTIME when a weaker form of conflict-freedom is satisfied by the supertype. In [28], by using automata-based encodings of types, Champavère et al. provide polynomial algorithms to check inclusion among EDTDs, with the restriction that the supertype is 1-unambiguous. In [29] Hovland provides an efficient algorithm to check inclusion of standard REs. The algorithm runs in polynomial time. It is sound and complete when the supertype is 1-unambiguous, otherwise the algorithm may either terminate with a definite answer or may signal its inability to answer because the supertype is not 1-unambiguous. The algorithm is defined via an inference system driven by the REs syntax, hence avoiding possibly expensive automata construction.

6. Conclusions

In [24] we introduced the idea of representing REs with interleaving and counting as sets of constraints, and the use of this representation for inclusion checking. Inclusion of such extended REs has EXPSPACE complexity in general, hence very far from what is usually regarded as 'feasible', while our approach produced a cubic algorithm, later reduced to $O(n^2)$ (in [1]), for the important subclass of conflict-free types. Unfortunately, while conflict-free types fit well the common practice of XML schema definitions, they are far too restrictive to capture the types that are typically inferred by a compiler. Subtype-checking during type checking is arguably the most important application of type inclusion, and is the one where efficiency is most important, hence this was a serious limitation for our approach. However, any minimal attempt to relax the constraints of conflict-free types seems to immediately bring us into the NP class, or out of the expressive power of the constraint language.

In this paper we have described a way out of this impasse. Through the lateral step of asymmetric inclusion, we have been able to widen our approach up to the point where all limitations are removed from the subtype, which makes it perfect for the typical use of inclusion checking by the type-checking algorithm of a compiler. The resulting algorithm retains the quadratic complexity of the pure case, and our experiments show that it runs very fast in practice, even on types that are quite large, which makes it viable for practical use.

Appendix. Proofs

In this appendix we report the proofs we omitted in the body for the sake of readability. For each of the proofs we recall the property statement.

Proposition 3.17. *Each of the following seven types, corresponding to different ways of loosening the restrictions that define conflict-free types, is constraint-inexpressible.*

Loosening counting restriction	
Allowing $U\,[m..n]$ under counting	$a\,[1..2]\,[1..2]$
Allowing $U_1 + U_2$ under counting	$(a+b)\,[2..2]$
Allowing $U_1 \cdot U_2$ under counting	$(a\cdot b)\,[2..2]$
Allowing $U_1 \& U_2$ under counting	$(a\& b)\,[1..2]$

Loosening single-occurrence restriction	
Allowing $a \in sym(U_1) \cap sym(U_2)$ in a subterm $U_1 + U_2$	$(a\cdot b) + (b\cdot a\cdot c)$
Allowing $a \in sym(U_1) \cap sym(U_2)$ in a subterm $U_1 \cdot U_2$	$a\cdot (b\cdot a)$
Allowing $a \in sym(U_1) \cap sym(U_2)$ in a subterm $U_1 \& U_2$	$a\&(b\cdot a)$

Proof. For each type T above we exhibit a word $w_T \notin [\![T]\!]$ such that, for any single constraint F, we have that:

$$T \models F \Rightarrow w_T \models F \tag{$*$}$$

Existence of such a $w_T \notin [\![T]\!]$ implies that T is constraint-inexpressible, as follows: consider any constraint set $\mathcal{F}$ such that $T \models \mathcal{F}$, hence $\forall F \in \mathcal{F} .\, T \models F$, hence, by ($*$):

$$\forall F \in \mathcal{F} . w_T \models F$$

Hence, $T \models \mathcal{F} \Rightarrow w_T \models \mathcal{F}$, hence no constraint set $\mathcal{F}$ can be exact for T.

For each T and F, the proof that $w_T \models F$ will rely on the following facts, that are direct consequence of the definition of constraints (Fig. 1):

(i) Assume F is not a counting constraint. If w and w' only differ in the length of some substring of consecutive equal symbols, then $w \models F \Leftrightarrow w' \models F$. Formally, for any sequence $m_1, \ldots, m_k, n_1, \ldots, n_k$ of integers strictly greater than 0, we have ${a_1}^{n_1} \ldots {a_k}^{n_k} \models F \Leftrightarrow {a_1}^{m_1} \ldots {a_k}^{m_k} \models F$, where a^n is a sequence of n consecutive a's.

(ii) Assume F is not an order constraint, and w' a permutation of w. Then $w \models F \Leftrightarrow w' \models F$.
(iii) Assume F is a counting constraint, and w and w' such that $sym(w) \cap sym(w') = \emptyset$. If $w \models F$ and $w' \models F$ then $w \cdot w' \models F$ (recall that a counting constraint counts only one symbol).

We now present the proofs for each of the seven types in the statement.

- $T = a\,[1..2]\,[1..2]$, $w_T = aaa$. From $T \models F$ we have $aa \models F$ and $aaaa \models F$. If F is not a counting constraint, then by (i) and $aa \models F$ we have $aaa \models F$. If it is a counting constraint, then $aa \models F$ and $aaaa \models F$ imply that either $F = b?[n..m]$ with $a \neq b$, or $F = a?[n'..m']$ with $n' \leq 2$ and $m' \geq 4$. In both cases we have $aaa \models F$.
- $T = (a + b)\,[2..2]$, $w_T = aabb$. From $T \models F$ we have $aa \models F$, $bb \models F$, and $ab \models F$. If F is a counting constraint, then by (iii) $aa \models F$ and $bb \models F$ imply $aabb \models F$. If F is not a counting constraint, then by (i) and $ab \models F$ we have $aabb \models F$.
- $T = (a \cdot b)\,[2..2]$, $w_T = aabb$. From $T \models F$ we have $abab \models F$. If F is not an order constraint, then by (ii) and $abab \models F$ we have $aabb \models F$. If F is in an order constraint, then $abab \models F$ implies $aabb \models F$, because $abab$ only satisfies trivial order constraint, i.e. constraints $c \prec d$ where either $c \notin \{ab\}$ or $d \notin \{ab\}$.
- $T = (a\&b)\,[1..2]$, $w_T = aab$. From $T \models F$ we have $ab \models F$ and $abab \models F$. If F is not a counting constraint, then (i) and $ab \models F$ imply $aab \models F$. If F is a counting constraint, then $ab \models F$ and $abab \models F$ imply that it is satisfied by any word with either 1 or 2 a's and either 1 or 2 b's, hence $aab \models F$.
- $T = (a \cdot b) + (b \cdot a \cdot c)$, $w_T = abc$. From $T \models F$ we have $bac \models F$. If F is not an order constraint, then (ii) and $bac \models F$ imply $abc \models F$. If F is in an order constraint that includes symbols out of $\{a, b, c\}$, then $abc \models F$ holds trivially. The only order constraints that can be expressed with $\{a, b, c\}$ and that hold for both ab and bac are $a \prec c$ and $b \prec c$, and abc satisfies both.
- $T = a\&(b \cdot a)$, $w_T = aab$. From $T \models F$ we have $aba \models F$. If F is not an order constraint, then (ii) and $aba \models F$ imply $aab \models F$. If F is an order constraint, then $aba \models F$ implies $aab \models F$, because aba only satisfies trivial order constraints, i.e., constraints $c \prec d$ where either $c \notin \{ab\}$ or $d \notin \{ab\}$.
- $T = a\&(b \cdot a)$, $w_T = aab$. From $T \models F$ we have $aba \models F$. As in the previous case, $aba \models F$ implies $aab \models F$. □

Lemma 3.16. *Let C_s be a non-repetitive labelled context and L a labelled type such that $C_s[L]$ is well-formed, then:*

$$w \in [\![C_s[L]]\!] \wedge (sym(w) \cap sym(L)) \neq \emptyset \Rightarrow \exists w_1 \in [\![L]\!],\, w_2 \in sym(C_s)^* :\ w \in w_1\&w_2$$

Proof. By induction on C_s, and by cases. In all cases, we exploit the fact that a well-formed labelled type satisfies the single-occurrence property, that is, no labelled symbol a_i appears twice in $C_s[L]$.

Case $C_s = \_$: in this case $C_s[L] = L$; the thesis follows by taking $w_1 = w$ and $w_2 = \epsilon$.

Case $C_s = L' + C_s'$: since $C_s[L]$ is single-occurrence, from $w \in [\![(L' + C_s')[L]]\!]$ and $(sym(w) \cap sym(L)) \neq \emptyset$ we deduce that $w \in [\![C_s'[L]]\!]$, and the thesis follows by induction.

Case $C_s = L' \otimes C_s'$: $w \in [\![(L' \otimes C_s')[L]]\!]$ implies that $w \in w'\&w''$ with $w' \in [\![L']\!]$ and $w'' \in [\![C_s'[L]]\!]$. Since $C_s[L]$ is single-occurrence, we have that $(sym(w) \cap sym(L)) = (sym(w'') \cap sym(L))$, hence $(sym(w'') \cap sym(L)) \neq \emptyset$. By induction, $\exists w_1'' \in [\![L]\!],\, w_2'' \in sym(C_s')^* :\ w'' \in w_1''\&w_2''$, and the thesis follows: $w_1 = w_1''$ and w_2 is a shuffle of w' and w_2''. Since $w' \in sym(L')^*$ and $w_2'' \in sym(C_s')^*$, then $w_2 \in (sym(L') \cup sym(C_s'))^* = sym(C_s)^*$.

Cases $C_s = C_s' \circledast L'$: similar to the previous two cases.

Case $C_s = C_s'!$: $w \in [\![(C_s'!)[L]]\!] = [\![C_s'[L]!]\!]$ implies $w \in [\![C_s'[L]]\!]$, hence, by induction, $\exists w_1 \in [\![L]\!], w_2 :\ w \in w_1\&w_2$.

Case $C_s = C_s'\,[1..1]$: $w \in [\![(C_s'\,[1..1])[L]]\!] = [\![(C_s'[L])\,[1..1]]\!]$ implies $w \in [\![C_s'[L]]\!]$, hence, by induction, $\exists w_1 \in [\![L]\!], w_2 :\ w \in w_1\&w_2$. □

Property 3.17. *Let C_s be a non-repetitive labelled context and L a labelled type such that $C_s[L]$ is well-formed, then:*

$$\{a_i, b_j\} \subseteq sym(L) \wedge (a_i, b_j) \notin \mathsf{p}(L) \Rightarrow (a_i, b_j) \notin \mathsf{p}(C_s[L]) \qquad (1)$$

If C is any labelled context and L a labelled type such that $C[L]$ is well-formed, then:

$$\mathsf{p}(L) \subseteq \mathsf{p}(C[L]) \qquad (2)$$

If C_r is a repetitive labelled context and L a labelled type such that $C_r[L]$ is well-formed, then:

$$\{a_i, b_j\} \subseteq sym(L) \Rightarrow (a_i, b_j) \in \mathsf{p}(C_r[L]) \qquad (3)$$

Proof. (1) We prove that $\{a_i, b_j\} \subseteq sym(L) \wedge (a_i, b_j) \in \mathsf{p}(C_s[L]) \Rightarrow (a_i, b_j) \in \mathsf{p}(L)$, which is equivalent to (1).

Assume $\{a_i, b_j\} \subseteq sym(L)$ and $(a_i, b_j) \in \mathsf{p}(C_s[L])$. Then, there exists $w \in [\![C_s[L]]\!]$ such that $w = w' \cdot a_i \cdot w'' \cdot b_j \cdot w'''$. Hence, by Lemma 3.16, $\exists w_1 \in [\![L]\!],\, w_2 \in sym(C_s)^* :\ w \in w_1\&w_2$. Since $C_s[L]$ meets the single-occurrence property, a_i and b_j do not appear in w_2, hence they appear, in that order, inside w_1, hence $(a_i, b_j) \in \mathsf{p}(L)$.

(2) We prove that for any $w \in [\![L]\!]$ exists $w' \in (\Sigma \times \mathbb{N})^*$ and $W \in [\![C[L]]\!]$ such that $W \in w\&w'$, by induction on C and by cases, exploiting the fact that no type is empty (Lemma 2.8(1)). The thesis follows immediately.

(3) Since C_r is repetitive, there exist contexts C and C' such that $C_r = C[(C'\,[m..n])]$, with $n > 1$. We want to prove that $\{a_i, b_j\} \subseteq sym(L)$ implies $(a_i, b_j) \in \mathsf{p}(C_r[L])$. By Lemma 2.8(2), we have two words W_a and W_b with $\{W_a, W_b\} \subseteq [\![C'[L]]\!]$, $a_i \in sym(W_a)$ and $b_j \in sym(W_b)$. Hence, $W_a \cdot (W_b)^{n-1} \in [\![(C'[L])\,[m..n]]\!]$, hence $(a, b) \in \mathsf{p}((C'[L])\,[m..n])$, hence $(a, b) \in \mathsf{p}(C[(C'[L])\,[m..n]]) = \mathsf{p}(C_r[L])$, by Property (2). □

Lemma 3.22

$$\begin{array}{ll}
\mathrm{Min}(\epsilon, a) & = 0 \\
\mathrm{Min}(a, a) & = 1 \\
\mathrm{Min}(b, a) & = 0 \text{ if } b \neq a \\
\mathrm{Min}(T_1 + T_2, a) & = \min(\mathrm{Min}(T_1, a), \mathrm{Min}(T_2, a)) \\
\mathrm{Min}(T_1 \otimes T_2, a) & = \mathrm{Min}(T_1, a) + \mathrm{Min}(T_2, a) \\
\mathrm{Min}(T\,[m..n]\,, a) & = m \cdot \mathrm{Min}(T, a) \\
\mathrm{Min}(T!, a) & = \mathrm{Min}^{!}(T, a)
\end{array}$$

$$\begin{array}{ll}
\mathrm{Min}^{!}(\epsilon, a) & = * \\
\mathrm{Min}^{!}(a, a) & = 1 \\
\mathrm{Min}^{!}(b, a) & = 0 \text{ if } b \neq a \\
\mathrm{Min}^{!}(T_1 + T_2, a) & = \min(\mathrm{Min}^{!}(T_1, a), \mathrm{Min}^{!}(T_2, a)) \\
\mathrm{Min}^{!}(T_1 \otimes T_2, a) & = \min(\mathrm{Min}^{!}(T_1, a) + \mathrm{Min}(T_2, a),\ \mathrm{Min}(T_1, a) + \mathrm{Min}^{!}(T_2, a)) \\
\mathrm{Min}^{!}(T\,[m..n]\,, a) & = \mathrm{Min}^{!}(T, a) + (m-1) \cdot \mathrm{Min}(T, a) \\
\mathrm{Min}^{!}(T!, a) & = \mathrm{Min}^{!}(T, a)
\end{array}$$

$$\begin{array}{ll}
\mathrm{Min}^{app}(\epsilon, a) & = * \\
\mathrm{Min}^{app}(a, a) & = 1 \\
\mathrm{Min}^{app}(b, a) & = * \text{ if } b \neq a \\
\mathrm{Min}^{app}(T_1 + T_2, a) & = \min(\mathrm{Min}^{app}(T_1, a), \mathrm{Min}^{app}(T_2, a)) \\
\mathrm{Min}^{app}(T_1 \otimes T_2, a) & = \min(\mathrm{Min}^{app}(T_1, a) + \mathrm{Min}(T_2, a),\ \mathrm{Min}(T_1, a) + \mathrm{Min}^{app}(T_2, a)) \\
\mathrm{Min}^{app}(T\,[m..n]\,, a) & = \mathrm{Min}^{app}(T, a) + (m-1) \cdot \mathrm{Min}(T, a) \\
\mathrm{Min}^{app}(T!, a) & = \mathrm{Min}^{app}(T, a)
\end{array}$$

Proof. All cases for $\mathrm{Min}(T, a)$ are obvious.

For $\mathrm{Min}^{!}(T, a)$, the only non-obvious cases are $\mathrm{Min}^{!}(T_1 \otimes T_2, a)$ and $\mathrm{Min}^{!}(T\,[m..n]\,, a)$, and we discuss them here.

$\mathrm{Min}^{!}(T_1 \otimes T_2, a) = \min(\mathrm{Min}^{!}(T_1, a) + \mathrm{Min}(T_2, a),\ \mathrm{Min}(T_1, a) + \mathrm{Min}^{!}(T_2, a))$

If $sym(T_1) = \emptyset$, then $[\![T_1]\!] = \{\epsilon\}$ (Lemma 2.8) hence $[\![T_1 \otimes T_2]\!] = [\![T_2]\!]$, hence $\mathrm{Min}^{!}(T_1 \otimes T_2, a) = \mathrm{Min}^{!}(T_2, a)$. Since $sym(T_1) = \emptyset$, then $\mathrm{Min}^{!}(T_1, a) = *$ and $\mathrm{Min}(T_1, a) = 0$, hence $\min(\mathrm{Min}^{!}(T_1, a) + \mathrm{Min}(T_2, a),\ \mathrm{Min}(T_1, a) + \mathrm{Min}^{!}(T_2, a)) = \min(* + \mathrm{Min}(T_2, a),\ 0 + \mathrm{Min}^{!}(T_2, a)) = \mathrm{Min}^{!}(T_2, a)$.

If $sym(T_2) = \emptyset$, we reason in the same way.

We are left with the case when $sym(T_1) \neq \emptyset$ and $sym(T_2) \neq \emptyset$, which implies that $\mathrm{Min}^{!}(T, a) \neq *$ for T_1, T_2 and $T_1 \otimes T_2$.

For this case, we first prove that

$$\mathrm{Min}^{!}(T_1 \otimes T_2, a) \leq \min(\mathrm{Min}^{!}(T_1, a) + \mathrm{Min}(T_2, a),\ \mathrm{Min}(T_1, a) + \mathrm{Min}^{!}(T_2, a))$$

Let $m = \mathrm{Min}^{!}(T_1 \otimes T_2, a)$, and consider two words $w_1 \in [\![T_1]\!]$ and $w_1' \in [\![T_1]\!]$ with $w_1' \neq \epsilon$, such that $|w_1|_a = \mathrm{Min}(T_1, a)$ and $|w_1'|_a = \mathrm{Min}^{!}(T_1, a)$, and two words $w_2 \in [\![T_2]\!]$ and $w_2' \in [\![T_2]\!]$ with $w_2' \neq \epsilon$, such that $|w_2|_a = \mathrm{Min}(T_2, a)$ and $|w_2'|_a = \mathrm{Min}^{!}(T_2, a)$. Both words $w_1 \cdot w_2'$ and $w_1' \cdot w_2$ belong to $[\![T_1 \otimes T_2]\!]$ and differ from ϵ, hence $|w_1' \cdot w_2|_a \geq \mathrm{Min}^{!}(T_1 \otimes T_2, a)$ and $|w_1 \cdot w_2'|_a \geq \mathrm{Min}^{!}(T_1 \otimes T_2, a)$, hence $\mathrm{Min}^{!}(T_1, a) + \mathrm{Min}(T_2, a) \geq \mathrm{Min}^{!}(T_1 a \otimes T_2, a)$ and $\mathrm{Min}(T_1, a) + \mathrm{Min}^{!}(T_2, a) \geq \mathrm{Min}^{!}(T_1 \otimes T_2, a)$, hence $\mathrm{Min}^{!}(T_1 \otimes T_2, a) \leq \min(\mathrm{Min}^{!}(T_1, a) + \mathrm{Min}(T_2, a),\ \mathrm{Min}(T_1, a) + \mathrm{Min}^{!}(T_2, a))$.

For the other inequality, consider any word $w \in [\![T_1 \otimes T_2]\!]$ with $w \neq \epsilon$ such that $|w|_a = m$. By $w \in [\![T_1 \otimes T_2]\!]$, there exist $w_1 \in [\![T_1]\!]$ and $w_2 \in [\![T_2]\!]$ such that $w \in w_1 \& w_2$ hence $|w|_a = |w_1|_a + |w_2|_a$. Since $w \neq \epsilon$, either $w_1 \neq \epsilon$ or $w_2 \neq \epsilon$. In the first case, we have that $|w_1|_a \geq \mathrm{Min}^{!}(T_1, a)$ and $|w_2|_a \geq \mathrm{Min}(T_2, a)$, hence $m = |w|_a \geq \mathrm{Min}^{!}(T_1, a) + \mathrm{Min}(T_2, a)$, in the second case we have $m \geq \mathrm{Min}(T_1, a) + \mathrm{Min}^{!}(T_2, a)$, hence $m \geq \min(\mathrm{Min}^{!}(T_1, a) + \mathrm{Min}(T_2, a),\ \mathrm{Min}(T_1, a) + \mathrm{Min}^{!}(T_2, a))$.

$\mathrm{Min}^{!}(T\,[m..n]\,, a) = \mathrm{Min}^{!}(T, a) + (m-1) \cdot \mathrm{Min}(T, a)$

If $sym(T) = \emptyset$, then $sym(T\,[m..n]) = \emptyset$, hence both sides of the equation are equal to $*$. Otherwise, both $\mathrm{Min}^{!}(T\,[m..n]\,, a)$ and $\mathrm{Min}^{!}(T, a)$ differ from $*$.

In the case when $sym(T) \neq \emptyset$, we first prove that $\mathrm{Min}^{!}(T\,[m..n]\,, a) \geq \mathrm{Min}^{!}(T, a) + (m-1) \cdot \mathrm{Min}(T, a)$. Consider a word $w \in [\![T\,[m..n]]\!]$ such that $w \neq \epsilon$ and $|w|_a = \mathrm{Min}^{!}(T\,[m..n]\,, a)$. Since $w \in [\![T\,[m..n]]\!]$, we have l words $w_1, \ldots, w_l$ in $[\![T]\!]$, with $m \leq l \leq n$, such that $w = w_1 \cdot \ldots \cdot w_l$. For each w_i we have that $|w_i|_a \geq \mathrm{Min}(T, a)$, and at least one of them, say w_j, has $w_j \neq \epsilon$, hence $|w_j|_a \geq \mathrm{Min}^{!}(T, a)$; as a consequence, $|w_1 \cdot \ldots \cdot w_l|_a \geq \mathrm{Min}^{!}(T, a) + (l-1) \cdot \mathrm{Min}(T, a)$. Since $|w_1 \cdot \ldots \cdot w_l|_a = \mathrm{Min}^{!}(T\,[m..n]\,, a)$ by construction, we conclude that $\mathrm{Min}^{!}(T\,[m..n]\,, a) \geq \mathrm{Min}^{!}(T, a) + (l-1) \cdot \mathrm{Min}(T, a) \geq \mathrm{Min}^{!}(T, a) + (m-1) \cdot \mathrm{Min}(T, a)$.

For the other inequality, consider two words $w \in [\![T]\!]$ and $w' \in [\![T]\!]$ with $w' \neq \epsilon$, such that $|w|_a = \mathrm{Min}(T, a)$ and $|w'|_a = \mathrm{Min}^{!}(T, a)$. The word $W = w' \cdot w \cdot \ldots \cdot w$, where w is repeated $m-1$ times, belongs to $[\![T\,[m..n]]\!]$, and, by construction, $|W|_a = \mathrm{Min}^{!}(T, a) + (m-1) \cdot \mathrm{Min}(T, a)$, hence $\mathrm{Min}^{!}(T\,[m..n]\,, a) \leq \mathrm{Min}^{!}(T, a) + (m-1) \cdot \mathrm{Min}(T, a)$.

The same proof holds for the corresponding cases for $\text{Min}^{app}(T, a)$, if we substitute $sym(T) = \emptyset$ with $a \notin sym(T)$, and $w \neq \epsilon$ with "a appears in w". □

References

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

[2] G. Ghelli, D. Colazzo, C. Sartiani, Linear time membership in a class of regular expressions with interleaving and counting, in: J.G. Shanahan, S. Amer-Yahia, I. Manolescu, Y. Zhang, D.A. Evans, A. Kolcz, K.-S. Choi, A. Chowdhury (Eds.), CIKM, ACM, 2008, pp. 389–398.

[3] D. Colazzo, G. Ghelli, C. Sartiani, Efficient asymmetric inclusion between regular expression types, in: R. Fagin (Ed.), ICDT, in: ACM International Conference Proceeding Series, vol. 36, ACM, 2009, pp. 174–182.

[4] D.C. Fallside, P. Walmsley, XML Schema Part 0: Primer – second ed., 2004. W3C Recommendation.

[5] RELAX NG specification, The Organization for the Advancement of Structured Information Standards [OASIS], 2001. Committee Specification 3 December 2001.

[6] A.J. Mayer, L.J. Stockmeyer, Word problems — this time with interleaving, Inform. and Comput. 115 (1994) 293–311.

[7] W. Gelade, W. Martens, F. Neven, Optimizing schema languages for XML: numerical constraints and interleaving, in: T. Schwentick, D. Suciu (Eds.), Proceedings of the 11th International Conference on Database Theory – ICDT 2007, Barcelona, Spain, January 10–12, 2007, in: Lecture Notes in Computer Science, vol. 4353, Springer, 2007, pp. 269–283.

[8] G.J. Bex, F. Neven, T. Schwentick, K. Tuyls, Inference of concise DTDs from XML data, in: U. Dayal, K.-Y. Whang, D.B. Lomet, G. Alonso, G.M. Lohman, M.L. Kersten, S.K. Cha, Y.-K. Kim (Eds.), Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12–15, 2006, ACM, 2006, pp. 115–126.

[9] D. Barbosa, G. Leighton, A. Smith, Efficient incremental validation of XML documents after composite updates, in: XSym, in: LNCS, vol. 4156, Springer, 2006, pp. 107–121.

[10] B. Choi, What are real DTDs like?, in: Proceedings of the Fifth International Workshop on the Web and Databases, WebDB 2002, Madison, Wisconsin, USA, June 6–7, 2002, in conjunction with ACM PODS/SIGMOD 2002, pp. 43–48.

[11] J.A. Brzozowski, Derivatives of regular expressions, J. ACM 11 (1964) 481–494.

[12] W. Martens, F. Neven, T. Schwentick, Complexity of decision problems for XML schemas and chain regular expressions, SIAM J. Comput. 39 (2009) 1486–1530.

[13] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: G.H. Gonnet, D. Panario, A. Viola (Eds.), LATIN, in: Lecture Notes in Computer Science, vol. 1776, Springer, 2000, pp. 88–94.

[14] T. Antonopoulos, F. Geerts, W. Martens, F. Neven, Generating, sampling and counting subclasses of regular tree languages, in: T. Milo (Ed.), ICDT, ACM, 2011, pp. 30–41.

[15] J. Jedrzejowicz, A. Szepietowski, Shuffle languages are in P, Theoret. Comput. Sci. 250 (2001) 31–53.

[16] H. Chen, L. Chen, Inclusion test algorithms for one-unambiguous regular expressions, in: J.S. Fitzgerald, A.E. Haxthausen, H. Yenigün (Eds.), ICTAC, in: Lecture Notes in Computer Science, vol. 5160, Springer, 2008, pp. 96–110.

[17] A. Brüggemann-Klein, D. Wood, One-unambiguous regular languages, Inform. and Comput. 142 (1998) 182–206.

[18] C.M. Sperberg-McQueen, Applications of Brzozowski derivatives to XML Schema processing, in: Proceedings of the Extreme Markup Languages 2005 Conference, 1–5 August 2005, Montréal, Quebec, Canada.

[19] W. Gelade, M. Gyssens, W. Martens, Regular expressions with counting: weak versus strong determinism, in: R. Královic, D. Niwinski (Eds.), MFCS, in: Lecture Notes in Computer Science, vol. 5734, Springer, 2009, pp. 369–381.

[20] P. Kilpeläinen, Checking determinism of XML Schema content models in optimal time, Inf. Syst. 36 (2011) 596–617.

[21] P. Kilpeläinen, R. Tuhkanen, Regular expressions with numerical occurrence indicators — preliminary results, in: P. Kilpeläinen, N. Päivinen (Eds.), SPLST, University of Kuopio, Department of Computer Science, 2003, pp. 163–173.

[22] P. Kilpeläinen, R. Tuhkanen, One-unambiguity of regular expressions with numeric occurrence indicators, Inform. and Comput. 205 (2007) 890–916.

[23] J.N. Foster, B.C. Pierce, A. Schmitt, A logic your typechecker can count on: unordered tree types in practice, in: PLAN-X 2007, Programming Language Technologies for XML, An ACM SIGPLAN Workshop colocated with POPL 2007, Nice, France, January 20, 2007, pp. 80–90.

[24] G. Ghelli, D. Colazzo, C. Sartiani, Efficient inclusion for a class of XML types with interleaving and counting, in: M. Arenas, M.I. Schwartzbach (Eds.), Proceedings of the 11th International Symposium on Database Programming Languages, DBPL 2007, in: Lecture Notes in Computer Science, vol. 4797, Springer, 2007, pp. 231–245.

[25] H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, XML Schema Part 1: structures second ed., Technical Report, World Wide Web Consortium, W3C Recommendation, 2004.

[26] D. Kozen, Lower bounds for natural proof systems, in: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, IEEE Computer Society, 1977, pp. 254–266.

[27] D. Colazzo, C. Sartiani, Efficient Subtyping for Unordered XML Types, Technical Report, Dipartimento di Informatica – Università di Pisa, 2007.

[28] J. Champavère, R. Gilleron, A. Lemay, J. Niehren, Efficient inclusion checking for deterministic tree automata and XML schemas, Inform. and Comput. 207 (2009) 1181–1208.

[29] D. Hovland, The inclusion problem for regular expressions, in: A.H. Dediu, H. Fernau, C. Martín-Vide (Eds.), LATA, in: Lecture Notes in Computer Science, vol. 6031, Springer, 2010, pp. 309–320.