Efficient Asymmetric Inclusion Between Regular Expression Types

Dario Colazzo
Laboratoire de Recherce en Informatique (LRI)
Université Paris Sud
UMR CNRS 8623, Orsay
F-91405 - France
dario.colazzo@lri.fr

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

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

ABSTRACT

The inclusion of Regular Expressions (REs) is the kernel of any subtype checking algorithm for XML schema 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 [9] we introduced a notion of "conflict-free REs", which are extended REs with excellent complexity behaviour, including a cubic inclusion algorithm [9] and linear membership [10]. 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, the most important use of subtype checking is in the context of type-cheching of XML manipulation languges. A type checker works by testing the inclusion of inferred subtypes in declared supertypes. The conflict-free restriction, while quite harmless for the human-defined supertype, is far too restrictive for the inferred subtype, whose shape is difficult to constrain.

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 [5]). The result is extremely surprising, since we had previously found that asymmetric 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.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Schema and subschema; H.2.3 [**Database Management**]: Database (persistent) programming languages

General Terms

Algorithms, Theory

Keywords

XML, language inclusion, regular expressions

ICDT 2009, March 23–25, 2009, Saint Petersburg, Russia.

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 DTDs, XML Schema, and RELAX-NG. This fact raised new interest in the study of such extended REs, and, specifically, in the crucial problem of language inclusion. The problem is EXPSPACE-complete [12, 8], but, in [9], we introduced a class of "conflict-free" REs with interleaving and counting, whose inclusion problem is in PTIME. The class is characterized by the single occurrence of each symbol and the limitation of Kleene-star to symbols. 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 when a function, expecting a type for its parameter, is applied to an expression, whose type is computed; this happens when the result of an expression is used to update a variable, whose expected type has been declared; this happens when the final result of a piece of code is compared with its expected type, in order to declare the code 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

[1]Quoting [3] "*an examination of the 819 DTDs and XSDs gathered from the Cover Pages (including many high-quality XML standards) as well as from the web at large, reveals that more than 99% of the REs occurring in practical schemas are CHAREs (and therefore also SOREs)*" (see also [11]); our conflict-free types are more expressive than CHAREs; similar results, in the high range of 90%, have been reported in [1] and [4]

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.

This seemed very hard for a time. The result in [9] is based on an exact description of conflict-free types through constraints, which reduces type inclusion to constraint implication. The smallest generalization of the conflict-free single-occurrence and Kleene-star limitations makes types impossible to be exactly described by our constraints. This problem does not arise if types are extended with intersection, since our constraints are closed by intersection. However, we showed in [9] that just one outermost use of binary intersection in the subtype makes inclusion NP-hard.

Luckily enough, we prove here that we can generalize our result without leaving PTIME if we embrace asymmetry, and consider the *mixed 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, while 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.

2. TYPES AND CONSTRAINTS

Following the terminology of [9], 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; a constraint 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 upper-approximated by the set of all constraints that it satisfies. In [9] we introduced conflict-free types, where 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 a mixed comparison, constraints provide an exact characterization for the conflict-free supertype, but just an upper-approximation for the subtype; we will prove below that this does not affect the correctness or completeness of the algorithm.

2.1 The Type Language

We describe here the specific syntax that we use for our extended REs, or "types".

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.

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

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

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.

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

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

where: $a \in \Sigma$, $m \in (\mathbb{N} \setminus \{0\})$, $n \in (\mathbb{N}_* \setminus \{0\})$, $n \geq m$, and, for any $T!$, at least one of the subterms of T has shape a. Here, $\mathbb{N}_*$ is $\mathbb{N} \cup \{*\}$, where $*$ behaves as $+\infty$, i.e., for any $n \in \mathbb{N}_*$, $* \geq n$.

Note that expressions like $T\,[0..n]$ are not allowed, due to the domain $(\mathbb{N} \setminus \{0\})$ of m, but the type $T\,[0..n]$ can be equivalently represented by $T\,[1..n] + \epsilon$. The type $T!$ denotes $[\![T]\!] \setminus \{\epsilon\}$. 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 2.4), which, in turn, implies that we have no empty types.

Definition 2.2 ($S(w), S(T)$) *For any word w, $S(w)$ is the set of all symbols appearing in w. For any type T, $S(T)$ is the set of all symbols appearing in T.*

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

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

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 2.3 *$N(T)$ is a predicate on types, defined as follows:*

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

In this system, no type is empty, hence any symbol in $S(T)$ appears in some word of T.

Lemma 2.4 (Not empty) *For any type T:*

$$[\![T]\!] \neq \emptyset \quad (1)$$
$$a \in S(T) \Rightarrow \exists w \in [\![T]\!].\ a \in S(w) \quad (2)$$

2.2 Constraints

Constraints are simple word properties, 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 $n \geq m$:

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

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. Constraint semantics is defined in Figure 1.

The following special cases are worth noticing.

$\epsilon \not\models A^+$	$\epsilon \models \mathrm{upper}(A)$	$\epsilon \models a?[m..n]$
$\epsilon \models a \prec b$	$b \models a \prec b$	$aba \not\models a \prec b$
$w \not\models \emptyset^+$	$w \models \emptyset^+ \Rightarrow A^+$	$w \models \emptyset^+ \Rightarrow \emptyset^+$

Observe that A^+ is monotone, i.e., $w \models A^+$ and w is subword of w' imply that $w' \models A^+$, while $\mathrm{upper}(A)$ and $a \prec b$ are anti-monotone.

A constraint F denotes the set of words that satisfy it, and a set of constraints S denotes the words that satisfy each $F \in S$, as follows.

Definition 2.5 ($[\![F]\!]$) *For any constraint F, set of constraints S:*

$$[\![F]\!] \stackrel{def}{=} \{w \mid w \models F\}$$
$$[\![S]\!] \stackrel{def}{=} \cap_{F \in S} [\![F]\!]$$

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

Definition 2.6 ($L \models F$, $T \models F$, $T \models S$) *For any set of words L, type T, constraint F, set of constraints S:*

$$L \models F \stackrel{def}{\Leftrightarrow} L \subseteq [\![F]\!]$$
$$T \models F \stackrel{def}{\Leftrightarrow} [\![T]\!] \subseteq [\![F]\!]$$
$$T \models S \stackrel{def}{\Leftrightarrow} [\![T]\!] \subseteq [\![S]\!]$$

2.3 Constraints and Subtyping

If we consider a function $\mathcal{C}$ mapping types to sets of constraints expressed in a language $\mathscr{F}$, we may define three properties that $\mathcal{C}$ may satisfy on a type T:

- soundness: $\mathcal{C}$ is sound for T if $T \models \mathcal{C}(T)$;
- $\mathscr{F}$-completeness: a sound $\mathcal{C}$ is complete for $\mathscr{F}$ and T if $[\![\mathcal{C}(T)]\!] = [\![\{F \in \mathscr{F} \mid T \models F\}]\!]$;
- exactness: $\mathcal{C}$ is exact for T if $[\![T]\!] = [\![\mathcal{C}(T)]\!]$.

Soundness is the basic property. A sound function is *complete* for T and $\mathscr{F}$ if its description of T cannot be made more precise by adding more constraints from $\mathscr{F}$: when $\mathcal{C}$ is $\mathscr{F}$-complete, for any $F \in \mathscr{F}$ *s.t.* $T \models F$, we have that $[\![C(T)]\!] \subseteq [\![F]\!]$, i.e., any valid F is subsumed by the constraints in $\mathcal{C}(T)$.

A function is exact for T if $\mathcal{C}(T)$ is satisfied by no more words than $[\![T]\!]$. A complete function is not necessarily exact; for example, no constraint set in our language is exact for the type $(aa)\,[1..2]$. If a complete function is not exact, no incomplete function may be exact. However, when an $\mathscr{F}$-complete function *is* exact, all and only the $\mathscr{F}$-complete functions are exact.

In [9] we defined a class of "conflict-free types", defined as those types that respect the following restrictions (hereafter we will use the meta-variable U for conflict-free types):

- *symbol counting*: if U has a subterm $U'\,[m..n]$, then U' must be the type a, for some $a \in \Sigma$ (only symbols can be counted or subject to Kleene-star);
- *single occurrence*: if U has a binary subterm $U_1 \circledast U_2$, then $S(U_1) \cap S(U_2) = \emptyset$ (no symbol appears twice).

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 (XML Schema Definition) schemas use repetition almost exclusively as a^{op} or as $(a + \ldots + z)^{\mathrm{op}}$ (where $\mathrm{op} \in \{+, *\}$, see [3]), which can be immediately translated to types that only count symbols, thank to the $U_1 \& U_2$ and $U!$ operators. For instance, $(a + \ldots + z)^*$ can be expressed as $(a^* \& \ldots \& z^*)$, where a^* is a shortcut for $a\,[1..*] + \epsilon$, while $(a + \ldots + z)^+$ can be expressed as $(a^* \& \ldots \& z^*)!$.

The main result of [9] is a complete constraint extraction procedure for the set of conflict-free types, plus the following exactness theorem.

Theorem 2.7 *If a type U is conflict-free, then any constraint extraction function that is complete for our constraint language is exact for U.*

A function satisfying $[\![U]\!] = [\![\mathcal{C}(U)]\!]$ reduces asymmetric inclusion to constraint-checking, as follows. The property is asymmetric because U must admit an exact constraint-extraction function, but T can be any type.[2]

Proposition 2.8 (Mixed subtyping) *If $\mathcal{C}$ is exact for U, then $[\![T]\!] \subseteq [\![U]\!] \Leftrightarrow T \models \mathcal{C}(U)$.*

This observation is obvious once it is framed in the right context, but it provides a way to generalize our previous results that is extremely interesting: rather than hunting for small generalizations of the conflict-free family in the narrow precinct of types which 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 complement the exact constraint-extraction of [9] with a procedure to test for $T \models \mathcal{C}(U)$. In [9] we (indirectly) 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).

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

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

Figure 1: Constraint semantics.

3. INCLUSION ALGORITHM

In [9], 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

$$
\begin{array}{rcl}
\mathcal{C}(U) & = & \mathcal{CC}(U) \cup \mathcal{OC}(U) \cup \\
 & & \textit{ZeroMinMax}(U) \cup \textit{upperS}(U) \cup \textit{SIf}(U)
\end{array}
$$

To apply Proposition 2.8, 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 $\mathcal{C}(U)$.

3.1 Co-Occurrence Constraints

The first component $\mathcal{CC}(U)$ of $\mathcal{C}(U)$ extracts a set of co-occurrence constraints with shape $A^+ \Mapsto B^+$, and is defined 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 [9].

$$
\begin{array}{lcl}
\mathcal{CC}(\epsilon) & \stackrel{def}{=} & \emptyset \\
\mathcal{CC}(a\,[m..n]) & \stackrel{def}{=} & \emptyset \\
\mathcal{CC}(U!) & \stackrel{def}{=} & \mathcal{CC}(U) \\
\mathcal{CC}(U_1 + U_2) & \stackrel{def}{=} & \mathcal{CC}(U_1) \cup \mathcal{CC}(U_2) \\
\mathcal{CC}(U_1 \otimes U_2) & \stackrel{def}{=} & \{S(U_1)^+ \Mapsto S(U_2)^+ \mid \neg\mathrm{N}(U_2)\} \\
 & & \cup\ \{S(U_2)^+ \Mapsto S(U_1)^+ \mid \neg\mathrm{N}(U_1)\} \\
 & & \cup\ \mathcal{CC}(U_1) \cup\ \mathcal{CC}(U_2)
\end{array}
$$

In this section, we define an algorithm to test $T \models A^+ \Mapsto B^+$ for any general type T.

This algorithm is based on the ability to discover which subterms T' of T satisfy the simpler constraints B^+ and $\Sigma^+ \Mapsto B^+$, which we abbreviate as B^{++}.

Definition 3.1 (A^{++}) *We define* $A^{++} \stackrel{def}{=} \Sigma^+ \Mapsto A^+$.

Property 3.2 ($[\![A^{++}]\!]$)

$$
\begin{array}{rcl}
[\![A^{++}]\!] & = & [\![A^+]\!] \cup \{\epsilon\} \\
T \models A^{++} & \Leftrightarrow & [\![T]\!] \setminus \{\epsilon\} \models A^+
\end{array}
$$

We focus on constraints with shape $a^+ \Mapsto A^+$ because of the following property, that is an immediate consequence of the definition of $A^+ \Mapsto B^+$.

Property 3.3 (Union) *For any word* w *and constraint* $A^+ \Mapsto B^+$*:*

$$w \models A^+ \Mapsto B^+ \ \Leftrightarrow\ \forall a \in A.\ w \models a^+ \Mapsto B^+$$

The reduction of $a^+ \Mapsto B^+$ to B^+ and B^{++} exploits the following lemmas (for reasons of space, most proofs are omitted).

Lemma 3.4 *For any type* $T_1 \otimes T_2$*:*

$$
\begin{array}{l}
T_1 \otimes T_2 \models a^+ \Mapsto A^+ \ \Leftrightarrow \\
\quad (T_1 \models a^+ \Mapsto A^+ \ \wedge\ T_2 \models a^+ \Mapsto A^+) \ \vee\ T_1 \otimes T_2 \models A^+
\end{array}
$$

Lemma 3.5 *For any type* T*,* T_1 *and* T_2*:*

$$
\begin{array}{lcl}
\epsilon \not\models A^+ & & \\
a \models A^+ & \Leftrightarrow & a \in A \\
T\,[m..n] \models A^+ & \Leftrightarrow & T \models A^+ \\
T_1 \otimes T_2 \models A^+ & \Leftrightarrow & T_1 \models A^+ \ \vee\ T_2 \models A^+ \\
T_1 + T_2 \models A^+ & \Leftrightarrow & T_1 \models A^+ \ \wedge\ T_2 \models A^+ \\
T! \models A^+ & \Leftrightarrow & T \models A^{++} \\
 & & \\
\epsilon \models A^{++} & & \\
a \models A^{++} & \Leftrightarrow & a \in A \\
T\,[m..n] \models A^{++} & \Leftrightarrow & T \models A^{++} \\
T_1 \otimes T_2 \models A^{++} & \Leftrightarrow & T_1 \models A^+ \ \vee\ T_2 \models A^+ \ \vee \\
 & & (T_1 \models A^{++} \wedge T_2 \models A^{++}) \\
T_1 + T_2 \models A^{++} & \Leftrightarrow & T_1 \models A^{++} \ \wedge\ T_2 \models A^{++} \\
T! \models A^{++} & \Leftrightarrow & T \models A^{++}
\end{array}
$$

We can now present the main result of this section, Theorem 3.6. It specifies that $T \models a^+ \Mapsto B^+$ can be verified by finding, for each occurrence a_i of a inside T, a subterm T' of T that contains a_i and such that $T' \models B^+$; T' may, or may not, coincide with T. Intuitively, each $w \in [\![T]\!]$ has a "parse tree" inside T, specifying one branch for each $+$. When $w \models a^+$, its parse tree must contain one a leaf and all its ancestors up to the root of T. If one of these ancestors enjoys $T' \models B^+$, then the piece of w recognized by that T' must satisfy B^+, hence $w \models B^+$. The tricky part is proving that this condition is necessary.

Theorem 3.6 ($T \models a^+ \Mapsto B^+$ **from** $T' \models B^+$) *For any type* T*,* $T \models a^+ \Mapsto B^+$ *iff, for each occurrence of* a *inside* T*, the occurrence is part of a subterm* T' *of* T *such that* $T' \models B^+$*.*

Moreover, when $T \models a^+ \Mapsto B^+$ *and* $a \notin B$*, then, for each occurrence of* a *inside* T*, the occurrence is part of a subterm* $T_1 \otimes T_2$ *of* T *such that* $T_1 \otimes T_2 \models B^+$*.*

Proof. ($\Rightarrow$). Assume $T \models a^+ \Mapsto B^+$. We prove the thesis by induction and by cases on the shape of T.

$T = T_1 + T_2$. Hence, $T_1 \models a^+ \Mapsto B^+$ and $T_2 \models a^+ \Mapsto B^+$. By induction, each occurrence of a subterm a in T_1 and in T_2 is part of a T' with $T' \models B^+$, as required.

$T = T_1 \otimes T_2$. By Lemma 3.5, either $T_1 \models a^+ \Rightarrow B^+$ and $T_2 \models a^+ \Rightarrow B^+$, or $T_1 \otimes T_2 \models B^+$. In the first case, we reason as in the case for $T = T_1 + T_2$. In the second case, T itself is the T' subterm with $T' \models B^+$.

$T = T'[m..n]$: immediate by induction.

$T = T'!$: $T'! \models a^+ \Rightarrow B^+$ implies that $T' \models a^+ \Rightarrow B^+$, since $\epsilon \models a^+ \Rightarrow B^+$, and the thesis follows by induction.

$T = a$: $T \models a^+ \Rightarrow B^+$ implies that $a \in B$, hence we choose $T' = T = a$.

$T = b \neq a$ and $T = \epsilon$: a does not occur inside T, hence the thesis holds trivially.

($\Leftarrow$). Assume that, for each occurrence of a inside T, the occurrence is part of a subterm T' of T such that $T' \models B^+$. We want to prove that $T \models a^+ \Rightarrow B^+$.

If $T \models B^+$, by definition $T \models a^+ \Rightarrow B^+$, hence we are done; when $T = a$, a is the only subterm of T, hence the hypothesis implies $a \models B^+$, hence we are in the case $T \models B^+$. If $T \not\models B^+$, we have two cases. When $T = b \neq a$ or $T = \epsilon$, then $T \models a^+ \Rightarrow B^+$ holds trivially. Otherwise, T must be a composite type $T_1 \circledast T_2$, $T_1!$, or $T_1[m..n]$, such that each of the components T_1 and T_2 satisfies the theorem hypothesis, hence, by induction, each of them satisfies $T_i \models a^+ \Rightarrow B^+$, hence $T \models a^+ \Rightarrow B^+$.

The second sentence of the statement can be proved in the same way.

We can now present the algorithm that we use to verify that, for each $A^+ \Rightarrow B^+ \in \mathcal{CC}(U)$, $T \models A^+ \Rightarrow B^+$ (Figure 2). For space reasons, we present a version that only works in absence of the $T!$ constructor, so that we can verify $S(T)^+$ with no reference to $S(T)^{++}$. This version is all we need to discuss the time bound; generalization to the full language is easy.

The auxiliary function MarkBPlus(T,B) returns *true* iff $T \models B^+$, and also marks each subterm a of T that appears inside a subterm T' of T such that $T' \models B^+$, exploiting Lemma 3.5. It is used by the function CoCheck, which verifies that $T \models A^+ \Rightarrow B^+$ by first marking all subterms a of T that appear inside a subterm T' of T such that $T' \models B^+$, and then checking that each occurrence of each $a \in A$ has been marked, according to Property 3.3 and to Theorem 3.6. The $NodesOfSymbol[]$ array associates each symbol a with the set of leaves in T that are labeled by a, and can be easily initialized in time $O(|T|)$. After preprocessing A and B so that membership can be checked in constant time, MarkBPlus can be computed in $O(|T| + |B|)$ time, since MarkBPlus and MarkAll never visit the same subtree twice, hence CoCheck can be computed in $O(|T| + |A| + |B|)$ time. CoImplies invokes CoCheck once for each $A^+ \Rightarrow B^+ \in \mathcal{CC}(U)$, i.e., at most twice for each $\otimes$ in U, hence CoImplies has $O((|T| + |U|) \times |U|)$ worst case time complexity, which is even better than the algorithm that we defined in [9] for the pure conflict-free case.

3.2 Order Constraints

Let us define $\mathcal{P}(T)$ 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 3.7 (Pairs)

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

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

Property 3.8 $T \models a \prec b \Leftrightarrow a \neq b$ *and* $(b, a) \notin \mathcal{P}(T)$

We verify whether a pair $(b, a) \in \mathcal{P}(T)$ by testing, for each instance of a and b in T, their Lower Common Ancestor (LCA) in the syntax tree of T; to this aim, we will manipulate a decorated version of T, $L(T)$, where each instance of a leaf is decorated with a distinct index i, and is denoted as a_i, and will consider the words generated by $L(T)$.

For example, if $T = a + b$, then $L(T) = a_1 + b_2$, and the LCA of a_1 and b_2 in $L(T)$ ($LCA_{L(T)}[a_i, b_j]$) is $+$. This LCA means that a_1 and b_2 never appear together, hence $(b_2, a_1) \notin \mathcal{P}(L(T))$, hence, since no other instance of a and b is present in T, $(b, a) \notin \mathcal{P}(T)$. In $a_1 \& b_2$, the LCA is $\&$, meaning that both (a, b) and (b, a) are in $\mathcal{P}(T)$. The use of LCA is justified by Lemma 2.4: with any two types $T_1 \& T_2$, as soon as $a_i \in S(T_1)$ and $b_j \in S(T_2)$, then T_1 has a word with a and T_2 has a word with b, hence (a, b) and (b, a) are in $\mathcal{P}(T)$. In a type $a_1 \cdot b_2$, order is relevant: $(a, b) \in \mathcal{P}(T)$ but $(b, a) \notin \mathcal{P}(T)$. We express this by extending the usual definition of $LCA_{L(T)}[a_i, b_j]$, assuming 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 $\circledast \neq \cdot$ (see Example 3.11).

$LCA_{L(T)}[a_i, b_j] \in \{\&, \cdot^{\rightarrow}\}$ implies that $(a, b) \in \mathcal{P}(T)$, but $(a, b) \in \mathcal{P}(T)$ also holds when $LCA_{L(T)}[a_i, b_j] \in \{+, \cdot^{\leftarrow}\}$, provided that the LCA is in the scope of a $T[m..n]$ operator with $n > 1$, as in $(a+b)[1..2]$ or in $(b \cdot a)[1..2]$; for this reason, in $L(T)$, we mark as $\circledast_r$ (for *repeated*) all binary operators $\circledast$ in the scope of a $T[m..n]$ with $n > 1$, and use $\circledast_1$ for all the other operator instances. Finally, if many occurrences of a and b appear in T, then $(a, b) \in \mathcal{P}(T)$ as soon as one pair (a_i, b_j) satisfies the test we described. This is formalized here.

Definition 3.9 ($L(T)$) *$L(T)$ is obtained from T by:*

- *rewriting each a as a_i, so that no two leaves get the same index;*
- *rewriting every instance of a binary operator $\circledast$ that is in the scope of at least one instance of $T[m..n]$ (with $n > 1$) as $\circledast_r$; every other instance of a binary operator is rewritten as $\circledast_1$.*

Property 3.10 (Pairs) *For any $a \neq b$:*

$$\begin{array}{lll} (a,b) \in \mathcal{P}(T) & \Leftrightarrow & \exists a_i, b_j \in L(T).\ LCA_{L(T)}[a_i, b_j] \in \{+_r, \otimes_r, \&_1, \cdot_1^{\rightarrow}\} \\ (b,a) \notin \mathcal{P}(T) & \Leftrightarrow & \forall a_i, b_j \in L(T).\ LCA_{L(T)}[a_i, b_j] \in \{+_1, \cdot_1^{\rightarrow}\} \\ T \models a \prec b & \Leftrightarrow & \forall a_i, b_j \in L(T).\ LCA_{L(T)}[a_i, b_j] \in \{+_1, \cdot_1^{\rightarrow}\} \end{array}$$

Example 3.11 ($LCA_{L(T)}[a_i, b_j]$) If $T = a \cdot ((b + a)[1..3])$, then $L(T) = a_1 \cdot {}_1((b_2 +_r a_3)[1..3])$, and $LCA_{L(T)}[a_i, b_j]$ is defined as in the following table. We avoid the arrow superscript on $+_r$, since the arrow direction is only significant in the $\cdot_1$ case.

	a_1	b_2	a_3
a_1	a_1	$\cdot_1^{\rightarrow}$	$\cdot_1^{\rightarrow}$
b_2	$\cdot_1^{\leftarrow}$	b_2	$+_r$
a_3	$\cdot_1^{\leftarrow}$	$+_r$	a_3

```
MarkBPlus(Type T, Set B)
  boolean result;
  case T  when T1 [m..n]: result = MarkBPlus(B, T1);
          when T1 ⊗ T2:   result = MarkBPlus(B, T1) ∨ MarkBPlus(B, T2);
                          if result then MarkAll(T1); MarkAll(T2);
          when T1 + T2:   result = MarkBPlus(B, T1) ∧ MarkBPlus(B, T2);
          when ϵ:         result = false;
          when a:         result = a ∈ B;
  marked[T] = result;
  return result;

MarkAll(Type T):
  if marked[T] then return;  else marked[T] = true;
  case T  when T1! or T = T1 [m..n]: MarkAll(T1);
          when T1 + T2 or T1 ⊗ T2:   MarkAll(T1);  MarkAll(T2);
          when ϵ or a:               return;

CoCheck(Type T , Set A, Set B)
  Global Array <Type → boolean> marked = [false];
  Global Array<Symbol → Set of Type> NodesOfSymbol = Prepare(T);
  MarkBPlus(B, T);
  every a in A, Ta in NodesOfSymbol[a]  satisfy marked[Ta]

CoImplies(Type T, Type U):
  every A+ ⇛ B+ in CC(U) satisfy CoCheck(T, A, B)
```

Figure 2: Algorithm for implication of co-occurrence constraints.

Hence, $(b,a) \in \mathcal{P}(T)$ because $LCA_{L(T)}[b_2, a_3] = +_r$, and $(a,b) \in \mathcal{P}(T)$ because $LCA_{L(T)}[a_1, b_2] = \cdot_1^{\rightarrow}$, but also because $LCA_{L(T)}[a_3, b_2] = +_r$. Hence, $T \not\models a \prec b$ and $T \not\models b \prec a$. ∎

Property 3.10 allows the definition of the following constraint-extraction function.

Definition 3.12 ($\mathcal{OC}_g^A(T)$)

$$\mathcal{OC}_g^A(T) \stackrel{def}{=} \{a \prec b \mid a,b \in A, \forall a_i, b_j \in L(T).\ LCA_{L(T)}[a_i, b_j] \in \{+_1, \cdot_1^{\rightarrow}\}\}$$

Property 3.13 (Completeness of $\mathcal{OC}_g^A(T)$) *$\mathcal{OC}_g^A(T)$ is complete for the set of constraints with shape $a \prec b$ and $\{a,b\} \subseteq A$.*

The order constraints component $\mathcal{OC}(U)$ of the exact constraint extraction function $\mathcal{C}(U)$ is indeed $\mathcal{OC}_g^{S(U)}(U)$. More precisely, since in a conflict-free type no symbol appears twice, and no operator is in the scope of a counting operator apart from a, $\mathcal{OC}(U)$ can be defined as follows, without any marking.

$$\mathcal{OC}(U) \stackrel{def}{=} \{a \prec b \mid a,b \in S(U),\ LCA_U[a,b] \in \{+, \cdot^{\rightarrow}\}\}$$

Property 3.10 also suggests an efficient algorithm to verify whether, for all $F \in \mathcal{OC}(U)$, we have $T \models F$. The algorithm is in Figure 3. It first builds, for each type, its decoration and a data structure to compute the LCA of any two leaves in constant time. This preprocessing phase can be done in linear time using the algorithm in [2].[3] The list $Leaves_T$ contains the leaves of T, while $NodeOfSymbol_U$ maps each symbol in $S(U)$ to the only corresponding leaf in U; the array $SymbolOfNode_T$, in turn, maps each leaf node a_i in T to the corresponding symbol a. After preprocessing, we access LCA_U once for each pair of leaves of U, and we save in OC_U every pair (a,b) such that $U \models a \prec b$. Then, we scan each pair of nodes in $Leaves_T$ and, for each pair of distinct symbols (a,b), we record **true** if all occurrences of (a,b) in T meet the LCA-based $\mathcal{OC}_g()$-test, **false** otherwise. Finally, we scan each pair of symbols (a,b) in OC_U and verify that $a \prec b \in \mathcal{OC}_g(T)$. It is easy to see that the algorithm complexity is $O(|T|^2 + |U|^2)$. Hence, also in this case, the extension from conflict-free inclusion to mixed inclusion adds no time complexity to the algorithm.

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

3.3 Cardinality Constraints

The cardinality constraints for a conflict-free type simply correspond to the instances of the counting operator. In particular, the cardinality constraint component of $\mathcal{C}(U)$ is $ZeroMinMax(U)$, defined as follows; $ZeroMinMax(U)$ is trivially complete for conflict-free types and for constraints with shape $a?[m..n]$ and $a \in S(U)$ [9]:

$$ZeroMinMax(U) = \{a?[m..n] \mid a\,[m..n] \text{ subterm of } U\}$$

General types are trickier, because of symbol repetition and generalized counting. In particular, the lowest allowed cardinality for a in T may depend on the validity of a^+ on some subterm of T. Consider, for example, the type $a\,[2..*] \cdot a\,[3..*]$: it clearly satisfies $a?[5..*]$. However, the type $(a\,[2..*] + \epsilon) \cdot (a\,[3..*] + \epsilon)$ only satisfies $a?[2..*]$: since a is optional, we consider here $\min(2,3)$ rather than $2+3$. Finally, $(a\,[2..*] + \epsilon) \cdot (a\,[3..*])$ satisfies $a?[3..*]$: since a is optional in the first subterm, we have to consider the bound of the second. In the same way, while $a\,[3..*]\,[4..*]$ satisfies $a?[12..*]$, the type $(a\,[3..*]+\epsilon)\,[4..*]$ only satisfies $a?[3..*]$. Exploiting this observation, we are able to define the following function $\mathrm{Min}^*(T,a)$, which, for a type T, computes the minimum number of

```
OrderImplies(Type T, Type U):
  LCA_T, Leaves_T, SymbolOfNode_T = PreprocessGeneralType(T);
  LCA_U, S(U), NodeOfSymbol_U = PreprocessCFType(U);
  Set OC_U = ∅
  for each (a, b) in S(U) × S(U)
      if(LCA_U[NodeOfSymbol_U(a), NodeOfSymbol_U(b)] in {+_1, ·_1^→})
      then OC_U.add((a, b))
  Array OC_T = {true_{1,1}, ..., true_{|S(U)×S(U)|}}
  for each n_1 in Leaves_T, n_2 in Leaves_T
      a = SymbolOfNode[n_1]
      b = SymbolOfNode[n_2]
      OC_T[(a, b)] = OC_T[(a, b)] ∧ (LCA_T[n_1, n_2] ∈ {+_1, ·_1^→})
  for each (a, b) in OC_U
      if (not OC_T[(a, b)])
      return false
  return true
```

Figure 3: Algorithm for implication of order constraints.

times that the symbol a appears in a word w of T such that $w \models a^+$. The $*$ in $\text{Min}^*(T,a)$ indicates that, when a appears in no word of T, then $\text{Min}^*(T,a)$ returns $*$; in the definitions below, we assume that all of $n+*$, $*+n$, $n \times *$, $* \times n$ return $*$. The condition $T \models a^+$ may be computed as described in Section 3.1, but it may also be computed together with $\text{Min}^*(T,a)$, as we will do later on.

Definition 3.14 ($\text{Min}^*(T,a)$)

$$
\begin{array}{rcl}
Min^*(T_1+T_2,a) & \stackrel{def}{=} & \min(Min^*(T_1,a), Min^*(T_2,a)) \\
Min^*(T_1 \otimes T_2,a) & \stackrel{def}{=} & \\
\multicolumn{3}{l}{\quad if\ \ T_1 \models a^+ \wedge T_2 \models a^+} \\
\multicolumn{3}{l}{\qquad then\ Min^*(T_1,a) + Min^*(T_2,a)} \\
\multicolumn{3}{l}{\quad elsif\ \ T_1 \models a^+ \wedge T_2 \not\models a^+} \\
\multicolumn{3}{l}{\qquad then\ Min^*(T_1,a)} \\
\multicolumn{3}{l}{\quad elsif\ \ T_1 \not\models a^+ \wedge T_2 \models a^+} \\
\multicolumn{3}{l}{\qquad then\ Min^*(T_2,a)} \\
\multicolumn{3}{l}{\quad elsif\ \ T_1 \not\models a^+ \wedge T_2 \not\models a^+} \\
\multicolumn{3}{l}{\qquad then\ \min(Min^*(T_1,a), Min^*(T_2,a))} \\
Min^*(b,a) & \stackrel{def}{=} & if\ \ b=a\ then\ 1\ else\ * \\
Min^*(T\,[m..n]\,,a) & \stackrel{def}{=} & \\
\multicolumn{3}{l}{\quad if\ \ T \models a^+\ \ then\ \ Min^*(T,a) \times m\ \ else\ \ Min^*(T,a)} \\
Min^*(T!,a) & \stackrel{def}{=} & Min^*(T,a) \\
Min^*(\epsilon,a) & \stackrel{def}{=} & *
\end{array}
$$

To reason about this approach, we need a clear specification of what $\text{Min}^*(T,a)$ is expected to compute. Let us define $|w|_a$ as the number of occurrences of a in w, and $\mathsf{SMin}^*(T,a)$ as the formal specification of $\text{Min}^*(T,a)$, as follows, where $[\![a^+]\!]$ are the words where a appears (notice that, by Lemma 2.4, $a \in S(T)$ iff $[\![T]\!] \cap [\![a^+]\!]$ is not empty).

$$
\begin{array}{rcll}
\mathsf{SMin}^*(T,a) & \stackrel{def}{=} & \min_{w \in ([\![T]\!] \cap [\![a^+]\!])} |w|_a & \text{if } a \in S(T) \\
\mathsf{SMin}^*(T,a) & \stackrel{def}{=} & * & \text{if } a \notin S(T)
\end{array}
$$

The following lemma is a bit tedious but very easy to prove.

Lemma 3.15 *For any type T and symbol a: $Min^*(T,a) = \mathsf{SMin}^*(T,a)$*

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

Definition 3.16 ($\text{Max}^0(T,a)$)

$$
\begin{array}{rcl}
Max^0(T_1+T_2,a) & \stackrel{def}{=} & \max(Max^0(T_1,a), Max^0(T_2,a)) \\
Max^0(T_1 \otimes T_2,a) & \stackrel{def}{=} & Max^0(T_1,a) + Max^0(T_2,a) \\
Max^0(b,a) & \stackrel{def}{=} & if\ \ b=a\ then\ 1\ else\ 0 \\
Max^0(T\,[m..n]\,,a) & \stackrel{def}{=} & Max^0(T,a) \times n \\
Max^0(T!,a) & \stackrel{def}{=} & Max^0(T,a) \\
Max^0(\epsilon,a) & \stackrel{def}{=} & 0
\end{array}
$$

The specification of $\text{Max}^0(T,a)$ is defined as follows.

Definition 3.17 ($\mathsf{SMax}^0(T,a)$)

$$
\begin{array}{rcll}
\mathsf{SMax}^0(T,a) & \stackrel{def}{=} & \max_{w \in [\![T]\!]} |w|_a & if\ (\max_{w \in [\![T]\!]} |w|_a) \in \mathbb{N} \\
\mathsf{SMax}^0(T,a) & \stackrel{def}{=} & * & if\ (\max_{w \in [\![T]\!]} |w|_a) = \infty
\end{array}
$$

The following lemma is immediate.

Lemma 3.18 *For any type T and symbol a: $Max^0(T,a) = \mathsf{SMax}^0(T,a)$*

As a consequence, cardinality constraint implication can be decided as follows.

Lemma 3.19

$$T \models a?[m..n] \ \Leftrightarrow\ m \le Min^*(T,a)\ \wedge\ Max^0(T,a) \le n$$

We can also extend the $ZeroMinMax(U)$ function to general types. The result is complete, but we have no hope of exactness once we abandon the symbol-counting limitation. For example, if you consider a type $(aa)\,[1..2]$, our constraint language has no way to specify that aa and $aaaa$ both belong to the type while the intermediate word aaa does not (remember that $F \vee F$ is not part of the language).

Definition 3.20 (Cardinality Constraints)

$$
\begin{array}{l}
ZeroMinMax_g(T) \\
\quad \stackrel{def}{=}\ \{a?[Min^*(T,a)..Max^0(T,a)]\ \mid\ a \in S(T)\}
\end{array}
$$

Corollary 3.21 *$ZeroMinMax_g(T)$ is complete for constraints with shape $a?[m..n]$ and $a \in S(T)$.*

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 Figure 4; the function *PlusMinMax* computes, in one pass, a boolean specifying whether $T \models a^+$, the value of $\mathrm{Min}^*(T, a)$, and the value of $\mathrm{Max}^0(T, a)$. We omit case $T!$ for simplicity; its inclusion requires the computation of a second boolean specifying whether $T \models a^{++}$.

$PlusMinMax(T, a)$ can be computed in time $O(|T|)$. *Card Implies* invokes it on T once for each symbol of U, hence it can be computed in time $O(|U| \times |T|)$.

3.4 Upper Bounds and Lower Bounds

The upper bound and lower bound components of $\mathcal{C}(U)$ are defined below.

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

Definition 3.22 (Upper and Lower components of $\mathcal{C}(U)$)

$$\begin{aligned} &\textit{Lower-bound:} \\ &\quad SIf(U) \stackrel{def}{=} \textit{if } \neg N(U) \textit{ then } \{S(U)^+\} \textit{ else } \emptyset \\ &\textit{Upper-bound:} \\ &\quad upperS(U) \stackrel{def}{=} \{upper(S(U))\} \end{aligned}$$

Theorem 3.23 (Implication of $SIf(T_2)$ and $upperS(T_2)$) *For any two types T_1 and T_2:*

$$\begin{aligned} & T_1 \models SIf(T_2) \cup upperS(T_2) \\ \Leftrightarrow \quad & (N(T_1) \Rightarrow N(T_2)) \ \wedge \ S(T_1) \subseteq S(T_2) \end{aligned}$$

Proof. ($\Rightarrow$) $T_1 \models upperS(T_2)$ implies $S(T_1) \subseteq S(T_2)$, by Lemma 2.4. 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 $T_1 \models S(T_2)^+$, hence $\epsilon \notin [\![T_1]\!]$, hence $\neg\mathrm{N}(T_1)$.

($\Leftarrow$) The implication $S(T_1) \subseteq S(T_2) \ \Rightarrow T_1 \models upperS(T_2)$ is trivial. 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 $S(T_1)$, hence a symbol from $S(T_2)$.

The corresponding function UpperLowerImplies simply executes the test of Theorem 3.23.

3.5 Summing up

We have recalled each of the five components of the function $\mathcal{C}(U)$, 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, the following theorem holds.

Theorem 3.24 *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), UpperLowerImplies(T, U) return* ***true****.*

CoImplies, OrderImplies, and CardImplies have quadratic time-complexity, while UpperLowerImplies is linear. The only case whose complexity is affected by the presence of general types in the subtype position is that of cardinality constraints, where the presence of multiple occurrences of a symbol and the nesting of $T\,[m..n]$ operators both concur in making the problem less trivial.

4. RELATED WORK

The problem of inclusion of regular expressions with interleaving has been studied in many papers. In [12], the complexity of membership, inclusion, and inequality was studied 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 [12], Gelade et al. [8] 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 [8], 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.

As we specified many times, in [9] 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.

The properties of a commutative type language for XML data have been discussed in [7]. 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.

To the best of our knowledge, the only paper dealing with asymmetric inclusion of XML types is [6]. Here, Colazzo and Sartiani showed that complexity of inclusion can be lowered from EXPSPACE to EXPTIME when a weaker form of conflict-freedom is satisfied by the supertype.

5. CONCLUSIONS

In [9] we introduced the idea of representing REs with interleaving and counting as sets of constraints, and of using this representation as a way to check inclusion. Inclusion of such extended REs has an appalling EXPSPACE complexity in general, while our approach produced a cubic algorithm, later reduced to $O(n^2)$, for an important subclass. Unfortunately, while the subclass fits well the common practice of XML schema definitions, it is 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.

We had hopes of extending our class of PTime comparable REs, since it does not contain all the types that can be exactly defined with our constraints. However, any attempt

```
CardImplies(Type T, Type U):
  every a [m..n] in U satisfy let (_, Min, Max) = PlusMinMax(T, a);
                              return Min ≥ m ∧ Max ≤ n

PlusMinMax(Type T, Symbol a):
  case T
  when T1 ⊗ T2:   case (PlusMinMax(T1, a), PlusMinMax(T2, a))
                  when ((true, Min1, Max1), (true, Min2, Max2)):
                        return(true, Min1 + Min2, Max1 + Max2);
                  when ((true, Min1, Max1), (false,Min2, Max2)):
                        return(true, Min1, Max1 + Max2);
                  when ((false, Min1, Max1), (true, Min2, Max2)):
                        return(true, Min2, Max1 + Max2);
                  when ((false, Min1, Max1), (false, Min2, Max2)):
                        return(false, min(Min1, Min2), Max1 + Max2);
  when T1 + T2:   let (Plus1, Min1, Max1) = PlusMinMax(T1, a);
                  let (Plus2, Min2, Max2) = PlusMinMax(T2, a);
                  return(Plus1 ∧ Plus2, min(Min1, Min2), max(Max1, Max2));
  when T1 [m..n]: let (Plus, Min, Max) = PlusMinMax(T1, a);
                  if Plus then return(true, Min ×m, Max ×n)
                        else return(false, Min, Max ×n)
  when ϵ:         return(false, *, 0)
  when a:         return(true, 1, 1)
  when b ≠ a:     return(false, *, 0)
```

Figure 4: Algorithm for implication of cardinality constraints.

to weaken the restrictions on single occurrence or counting immediately allows the definition of types which admit no exact description in the constraint language. The extension of the type language with intersection does not suffer this problem, as constraints are closed by intersection. However, we proved in [9] that even one instance of binary intersection is enough to make inclusion NP-hard, because it makes the $T \models F$ problem much harder.

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. Moreover, the resulting algorithm retains the quadratic complexity of the pure case, which is, frankly, quite amazing.

6. REFERENCES

[1] D. Barbosa, G. Leighton, and A. Smith. Efficient incremental validation of XML documents after composite updates. In *XSym*, volume 4156 of *LNCS*, pages 107–121. Springer, 2006.

[2] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In G. H. Gonnet, D. Panario, and A. Viola, editors, *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[3] G. J. Bex, F. Neven, T. Schwentick, and 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, and Y.-K. Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 115–126. ACM, 2006.

[4] B. Choi. What are real DTDs like? In *WebDB*, pages 43–48, 2002.

[5] D. Colazzo, G. Ghelli, and C. Sartiani. Efficient inclusion for a class of xml types with interleaving and counting. *Information Systems*, 2008. To Appear.

[6] D. Colazzo and C. Sartiani. Efficient subtyping for unordered XML types. Technical report, Dipartimento di Informatica - Università di Pisa, 2007.

[7] J. N. Foster, B. C. Pierce, and A. Schmitt. A logic your typechecker can count on: Unordered tree types in practice. In *Workshop on Programming Language Technologies for XML (PLAN-X), informal proceedings*, Jan. 2007.

[8] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for xml: Numerical constraints and interleaving. In T. Schwentick and D. Suciu, editors, *Proceedings of the 11th International Conference on Database Theory - ICDT 2007, Barcelona, Spain, January 10-12, 2007*, volume 4353 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2007.

[9] G. Ghelli, D. Colazzo, and C. Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. In M. Arenas and M. I. Schwartzbach, editors, *Proceedings of the 11th International Symposium on Database Programming Languages, DBPL 2007, Vienna, Austria, September 23-24, 2007, Revised Selected Papers*, volume 4797 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2007.

[10] G. Ghelli, D. Colazzo, and C. Sartiani. Linear time membership for a class of XML types with interleaving and counting. In *ACM Conference on Information and Knowledge Management (CIKM)*, 2008.

[11] W. Martens, F. Neven, T. Schwentick, and G. J. Bex. Expressiveness and complexity of xml schema. *ACM Trans. Database Syst.*, 31(3):770–813, 2006.

[12] A. J. Mayer and L. J. Stockmeyer. Word problems — this time with interleaving. *Inf. Comput.*, 115(2):293–311, 1994.