Efficient Asymmetric Inclusion Between Regular Expression Types

Dario Colazzo* Giorgio Ghelli† Carlo Sartiani†

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 [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, 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 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.

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 [11, 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.

*Laboratoire de Recherce en Informatique (LRI) - Université Paris Sud - France

†Dipartimento di Informatica - Università di Pisa - Italy

[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)*"; our conflict-free types are more expressive than CHAREs; similar results, in the high range of 90%, have been reported in [1] and [4]

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 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.*

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

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:

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

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 ($\mathbb{N}_* = \mathbb{N} \cup \{*\}$).

Note that expressions like $T\,[0..n]$ are not allowed due to the condition on m; of course, 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.*

Semantics of types is inductively defined by the following equations.

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

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

Types that contain the empty word ϵ are called *nullable* and are characterized as follows.

Definition 2.3 $\mathrm{N}(T)$ *is a predicate on types, defined as follows:*

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

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:*

$$\begin{array}{ll} [\![T]\!] \neq \emptyset & (1) \\ a \in S(T) \Rightarrow \exists w \in [\![T]\!].\ a \in S(w) & (2) \end{array}$$

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 as follows.

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

The following special cases of the above definition are worth noticing.

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

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 type satisfies a constraint if all of its words do. Set of constraints are interpreted as their conjunction.

Definition 2.5 ($T \models F$, $T \models S$) *For any type T, constraint F, set of constraints S:*

$$T \models F \quad \Leftrightarrow_{def} \quad \forall w \in [\![T]\!].\ w \models F \qquad\qquad T \models S \quad \Leftrightarrow_{def} \quad \forall F \in S.\ T \models S$$

A constraint denotes a set of words, as follows.

Definition 2.6 ($[\![F]\!]$) *For any type constraint F, set of constraints S:*

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

As a consequence, the relations $T \models F$ and $T \models S$ are equivalent to $[\![T]\!] \subseteq [\![F]\!]$ and $[\![T]\!] \subseteq [\![S]\!]$.

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 $a\,[1..2]\,[1..2]$. 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 $U = U'\,[m..n]$, 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 $U = U_1 + U_2$ or $U = U_1 \otimes U_2$, 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 $(a + \ldots + z)^{\mathrm{op}}$ (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 exact types, plus the following 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.*

An exact function can be used to define a mixed-subtyping algorithm, as follows. The algorithm is *mixed* because U must admit an exact constraint-extraction function,[2] but T can be any type.

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

Proof. ($\Rightarrow$) Assume $[\![T]\!] \subseteq [\![U]\!]$ and F in $\mathcal{C}(U)$, hence $U \models F$, hence $[\![U]\!] \subseteq [\![F]\!]$, hence $[\![T]\!] \subseteq [\![F]\!]$, hence $T \models F$.

($\Leftarrow$) Assume that $T \models \mathcal{C}(U)$. Hence, $[\![T]\!] \subseteq [\![\mathcal{C}(U)]\!]$, hence $[\![T]\!] \subseteq [\![U]\!]$ because $\mathcal{C}$ is exact for U. ■

To exploit this simple, albeit surprising, 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 general type U that is exactly described by $\mathcal{C}(U)$.

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 $\mathit{ZeroMinMax}(U)$, *lower-bound* constraints $\mathit{SIf}(U)$, and *upper-bound* constraints $\mathit{upperS}(U)$, that is, the exact function that we are going to use is defined as

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

To apply Theorem 2.8, we have now to exhibit, for each component $\mathcal{C}_i$ (where $\mathcal{C}_i$ is one of $\mathcal{CC}()$, $\mathcal{OC}(U)$, etc.), an algorithm to verify that, 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 $A^+ \mapsto B^+$. 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 $\Sigma^+ \mapsto B^+$. In this section only, we use A^{++} as an abbreviation for $\Sigma^+ \mapsto A^+$; by definition, $[\![A^{++}]\!] = [\![A^+]\!] \cup \{\epsilon\}$, hence $T \models A^{++} \Leftrightarrow [\![T]\!] \setminus \{\epsilon\} \models A^+$.

We first observe the following facts (for reasons of space, most proofs are omitted and can be found in the Appendix).

Lemma 3.1 *For any type $T_1 \otimes T_2$:*

$$\begin{array}{lcl} T_1 \otimes T_2 \models A^+ & \Leftrightarrow & T_1 \models A^+ \ \vee\ T_2 \models A^+ \\ T_1 \otimes T_2 \models A^{++} & \Leftrightarrow & T_1 \models A^+ \ \vee\ T_2 \models A^+ \ \vee\ (T_1 \models A^{++} \wedge T_2 \models A^{++}) \\ T_1 \otimes T_2 \models a^+ \mapsto A^+ & \Leftrightarrow & (T_1 \models a^+ \mapsto A^+ \ \wedge\ T_2 \models a^+ \mapsto A^+) \ \vee\ T_1 \otimes T_2 \models A^+ \end{array}$$

We can now define the functions $\mathcal{SC}(T)$, which computes $\{A \mid T \models A^+\}$, and $\mathcal{SC}^!(T)$, which computes $\{A \mid T \models A^{++}\}$.

Definition 3.2 (Set constraints)

$$\begin{array}{rcl@{\qquad}rcl} \mathcal{SC}(T_1 + T_2) & =_{def} & \mathcal{SC}(T_1) \cap \mathcal{SC}(T_2) & \mathcal{SC}^!(T_1 + T_2) & =_{def} & \mathcal{SC}^!(T_1) \cap \mathcal{SC}^!(T_2) \\ \mathcal{SC}(T_1 \otimes T_2) & =_{def} & \mathcal{SC}(T_1) \cup \mathcal{SC}(T_2) & \mathcal{SC}^!(T_1 \otimes T_2) & =_{def} & \mathcal{SC}(T_1) \cup \mathcal{SC}(T_2) \\ & & & & & \cup(\mathcal{SC}^!(T_1) \cap \mathcal{SC}^!(T_2)) \\ \mathcal{SC}(T\,[m..n]) & =_{def} & \mathcal{SC}(T) & \mathcal{SC}^!(T\,[m..n]) & =_{def} & \mathcal{SC}^!(T) \\ \mathcal{SC}(T!) & =_{def} & \mathcal{SC}^!(T) & \mathcal{SC}^!(T!) & =_{def} & \mathcal{SC}^!(T) \\ \mathcal{SC}(a) & =_{def} & \{A \mid A \in 2^\Sigma,\ a \in A\} & \mathcal{SC}^!(a) & =_{def} & \{A \mid A \in 2^\Sigma,\ a \in A\} \\ \mathcal{SC}(\epsilon) & =_{def} & \emptyset & \mathcal{SC}^!(\epsilon) & =_{def} & 2^\Sigma \end{array}$$

These functions are complete over set constraints A^+ and over "weak set constraints" A^{++}. The crucial case $T_1 \otimes T_2$ was proved in Lemma 3.1, the others are trivial.

Theorem 3.3 (Completeness of $\mathcal{SC}(T)$ and $\mathcal{SC}^!(T)$) *For any type T:*

$$A \in \mathcal{SC}(T) \quad \Leftrightarrow \quad T \models A^+ \tag{3.1}$$

$$A \in \mathcal{SC}^!(T) \quad \Leftrightarrow \quad T \models A^{++} \tag{3.2}$$

We can now present the main result of this section, Theorem 3.5. It specifies that $T \models a^+ \Rrightarrow B^+$ 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.

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

Property 3.4 (Union) *For any word w and constraint $A^+ \Rrightarrow B^+$:*

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

Theorem 3.5 ($T \models a^+ \Rrightarrow B^+$ from $T' \models B^+$) *For any type T, $T \models a^+ \Rrightarrow B^+$ 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 $T \models a^+ \Rrightarrow B^+$ 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 $T \models a^+ \Rrightarrow B^+$. We prove the thesis by induction and by cases on the shape of T.

$T = T_1 + T_2$. Hence, $T_1 \models a^+ \Rrightarrow B^+$ and $T_2 \models a^+ \Rrightarrow B^+$. 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 3.1, 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, 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.

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

$T = a$: $T \models a^+ \Rrightarrow B^+$ 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 $T \models a^+ \Rrightarrow B^+$.

If $T \models B^+$, by definition $T \models a^+ \Rrightarrow B^+$, hence we are done. If $a \notin S(T)$, then $T \models a^+ \Rrightarrow B^+$ also 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 $T_i \models a^+ \Rrightarrow B^+$, hence $T \models a^+ \Rrightarrow B^+$.

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 $A^+ \Rrightarrow B^+ \in \mathcal{CC}(U)$, $T \models A^+ \Rrightarrow B^+$. $\mathcal{CC}(U)$ 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.

$$\begin{array}{lll@{\qquad}lll}
\mathcal{CC}(\epsilon) & =_{def} & \emptyset & \mathcal{CC}(a\,[m..n]) & =_{def} & \emptyset \\
\mathcal{CC}(U!) & =_{def} & \mathcal{CC}(U) & \mathcal{CC}(U_1 + U_2) & =_{def} & \mathcal{CC}(U_1) \cup \mathcal{CC}(U_2)
\end{array}$$

$$\begin{array}{lll}
\mathcal{CC}(U_1 \otimes U_2) & =_{def} & \{S(U_1)^+ \Rrightarrow S(U_2)^+ \mid \neg\mathrm{N}(U_2)\} \,\cup\, \{S(U_2)^+ \Rrightarrow S(U_1)^+ \mid \neg\mathrm{N}(U_1)\} \\
 & & \cup\, \mathcal{CC}(U_1) \cup\, \mathcal{CC}(U_2)
\end{array}$$

The soundness of $\mathcal{CC}(U)$ is an immediate consequence of Theorem 3.5, if one observes that $S(T) \in \mathcal{SC}(T)$ iff $\neg \mathrm{N}(T)$. Its completeness is far subtler, since, for each $U_1 \otimes U_2$, we only extract $S(U_1)^+ \mapsto S(U_2)^+$, which looks much weaker than $\{S(U_1)^+ \mapsto B^+ \ \mid\ B \in \mathcal{SC}(U_2)\}$. Informally, we can inductively assume that $\mathcal{CC}(U_2)$ implies $S(U_2)^+ \mapsto B^+$ for any $B \in \mathcal{SC}(U_2)$, hence $S(U_1)^+ \mapsto S(U_2)^+ \cup \mathcal{CC}(U_2)$ implies $S(U_1)^+ \mapsto B^+$.

We can now present the constraint implication algorithm (Figure 1). For space reasons, we present a version that only works in absence of the $T!$ constructor, so that we can compute $\mathcal{SC}(T)$ with no reference to $\mathcal{SC}^!(T)$. This version is all we need to discuss the time bound; generalization to the full language is easy.

The algorithm uses the auxiliary function CoCheck, which verifies that $T \models A^+ \mapsto B^+$ 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. 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 $O(|T|)$. The marking process is performed by the auxiliary functions MarkBPlus and MarkAll. The first marks the subterms T' where $B \in \mathcal{SC}(T')$, while MarkAll marks all the descendants.

After preprocessing A and B so that membership can be checked in constant time, MarkB-Plus 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 $A^+ \mapsto B^+ \in \mathcal{CC}(U)$, i.e., at most twice for each $\otimes$ in U, hence CoImplies has $O((|T| + |U|) * |U|)$ 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 symbols (a, b) such that exists a word in $[\![T]\!]$ where an a comes before a b.

Definition 3.6 (Pairs) $\quad \mathcal{P}(T) \ =_{def}\ \{(a,b) \mid \exists w_1, w_2, w_3.\ w_1 \cdot a \cdot w_2 \cdot b \cdot w_3 \in [\![T]\!]\}$

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

Property 3.7 $\quad T \models a \prec b \ \Leftrightarrow\ a \neq b$ *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); 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.

For example, in $a_1 + b_2$ the LCA of a_1 and b_2 is $+$, meaning that a and b never appear together, hence $(b, a) \notin \mathcal{P}(T)$. 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 $\circledast \neq \cdot$.

$LCA_{L(T)}[a_i, b_j] \in \{\&, \cdot^{\rightarrow}\}$ 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 $\_\ [m..n]$ operator with $n > 1$,

```
MarkBPlus(Type $T$, Set $B$)
  boolean $result$;
  case $T$  when $T_1\,[m..n]$: $result$ = MarkBPlus($B, T_1$);
            when $T_1 \otimes T_2$:   $result$ = MarkBPlus($B, T_1$) $\vee$ MarkBPlus($B, T_2$);
                                if $result$ then MarkAll($T_1$); MarkAll($T_2$);
            when $T_1 + T_2$:   $result$ = MarkBPlus($B, T_1$) $\wedge$ MarkBPlus($B, T_2$);
            when $\epsilon$:            $result$ = false;
            when $a$:            $result$ = $a \in B$;
  $marked[T]$ = $result$;
  return $result$;

MarkAll(Type $T$):
  if $marked[T]$ then return;  else $marked[T]$ = true;
  case $T$  when $T_1!$ or $T = T_1\,[m..n]$: MarkAll($T_1$);
            when $T_1 + T_2$ or $T_1 \otimes T_2$:   MarkAll($T_1$);  MarkAll($T_2$);
            when $\epsilon$ 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 $A$, $T_a$ in $NodesOfSymbol[a]$  satisfy $marked[T_a]$

CoImplies(Type $T$, Type $U$):
  every $A^+ \Rightarrow B^+$ in $\mathcal{CC}(U)$ satisfy CoCheck($T, A, B$)
```

Figure 1: Algorithm for implication of co-occurrence constraints.

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 $\_\,[m..n]$ 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 3.8 ($L(T)$) *$L(T)$ 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 $\_\,[m..n]$ (with $n > 1$) as $\circledast_r$; every other instance of a binary operator is rewritten as $\circledast_1$.*

Property 3.9 (Pairs)

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

Property 3.9 allows the definition of the following constraint-extraction function.

Definition 3.10 ($\mathcal{OC}_g(T)$)

$$\mathcal{OC}_g(T) \ =_{def}\ \{a \prec b \mid a, b \in S(T),\ a \neq b,\ \forall a_i, b_j \in L(T).\ LCA_{L(T)}[a_i, b_j] \in \{+_1, \overrightarrow{\cdot_1}\}\}$$

Property 3.11 (Completeness of $\mathcal{OC}_g(T)$) *$\mathcal{OC}_g(T)$ is complete for the set of constraints with shape $a \prec b$ and $\{a, b\} \subseteq S(T)$.*

The order constraints component $\mathcal{OC}(U)$ of the exact constraint extraction function $\mathcal{C}(U)$ is indeed $\mathcal{OC}_g(U)$. More precisely, since 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.

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

While $\mathcal{OC}(U)$ is only complete for constraints $a \prec b$ with $\{a, b\} \subseteq S(U)$, the function $\mathcal{C}(U)$ is complete for every constraint $a \prec b$, because it also contains a component upper($S(U)$) (introduced in Section 3.4), which implies $a \prec b$ for all the pairs $\{a, b\}$ such that $\{a, b\} \not\subseteq S(U)$: if a cannot appear in a word of U, then both $U \models a \prec b$ and $U \models b \prec a$ hold trivially.

Property 3.9 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 2. 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 array $\mathit{NodesOfSymbol}_T$ maps each symbol a to the list of its instances in T, while $\mathit{NodeOfSymbol}_U$ maps each symbol in $S(U)$ to the only corresponding leaf in U; the array $\mathit{SymbolOfNode}_T$, 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 each pair (a, b) whose lowest common ancestor is $\{+_1$ or $\cdot^{\rightarrow}_1\}$. Then, we scan each pair of nodes in $\mathit{NodesOfSymbol}_T$ and, for each pair of distinct symbols (a, b), we record **true** if all occurrences of (a, b) in T meet the LCA predicate, **false** otherwise. Finally, we scan each pair of symbols (a, b) in *filters* and verify that all occurrences of (a, b) satisfy the LCA predicate. 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.

```
OrderImplies(Type $T$, Type $U$):
  $LCA_T$, $\mathit{NodesOfSymbol}_T$, $\mathit{SymbolOfNode}_T$ = PreprocessGeneralType($T$);
  $LCA_U$, $\mathit{NodeOfSymbol}_U$ = PreprocessCFType($U$);
  Set $\mathit{filters} = \emptyset$
  for each $(a, b)$ in $S(U) \times S(U)$
      if($LCA_U[\mathit{NodeOfSymbol}_U(a), \mathit{NodeOfSymbol}_U(b)]$ in $\{+_1, \cdot^{\rightarrow}_1\}$)
      then $\mathit{filters}.\mathit{add}((a, b))$
  Array $\mathit{filtered}$ = [true]
  for each $n_1$ in $\mathit{NodesOfSymbol}_T$
      $a = \mathit{SymbolOfNode}[n_1]$
      for each $n_2$ in $\mathit{NodesOfSymbol}_T$
          $b = \mathit{SymbolOfNode}[n_2]$
          $\mathit{filtered}[(a, b)] = \mathit{filtered}[(a, b)] \wedge (LCA_T[n_1, n_2] \in \{+_1, \cdot^{\rightarrow}_1\})$
  for each $(a, b)$ in $\mathit{filters}$
      if (not $\mathit{filtered}[(a, b)]$)
      return false
  return true
```

Figure 2: Algorithm for implication of order constraints.

[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 $\mathit{ZeroMinMax}(U)$, defined as follows; $\mathit{ZeroMinMax}(U)$ is trivially complete for conflict-free types and for constraints with shape $a?[m..n]$ and $a \in S(U)$ [9]:

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

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

Definition 3.12 ($\mathrm{Min}^*(T,a)$)

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

Let us define $|w|_a$ as the number of occurrences of a in w, and $\mathsf{SMin}^*(T,a)$ as the semantic counterpart to $\mathrm{Min}^*(T,a)$, 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).

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

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

Lemma 3.13 *For any type T and symbol a:* $\mathsf{SMin}^*(T,a) = \mathrm{Min}^*(T,a)$

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

Definition 3.14 ($\mathrm{Max}^0(T, a)$)

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

The semantic counterpart of $\mathrm{Max}^0(T, a)$ is defined as follows.

Definition 3.15 ($\mathsf{SMax}^0(T, a)$)

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

The following lemma is immediate.

Lemma 3.16 *For any type T and symbol a:* $\mathsf{SMax}^0(T, a) = \mathrm{Max}^0(T, a)$

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

Lemma 3.17 $\quad T \models a?[m..n] \ \Leftrightarrow\ m \leq \mathrm{Min}^*(T, a)\ \wedge\ \mathrm{Max}^0(T, a) \leq n$

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 $a\,[1..2]\,[1..2]$, 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 3.18 (Cardinality Constraints for General Types)

$$\mathit{ZeroMinMax}_g(T) \ \ =_{def} \ \ \{a?[\mathrm{Min}^*(T, a)..\mathrm{Max}^0(T, a)] \ \mid\ a \in S(T)\}$$

Corollary 3.19 *$\mathit{ZeroMinMax}_g(T)$ 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 3; 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 $\mathrm{Max}^0(T, a)$.

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

3.4 Upper Bound and Lower Bound Constraints

Upper bound and lower bound constraints are defined below. The function *SIf*(U), in isolation, is far from complete for the whole set of B^+ constraints (which are specified by $\mathcal{SC}(U)$), but it becomes complete once united with $\mathcal{CC}(U)$. More precisely, if $\mathrm{N}(U)$, then $\mathcal{SC}(U)$ is empty, hence *SIf*(U) *is* complete. Otherwise, *SIf*$(U) = S(U)^+$. Since $\mathcal{C}(U)$ is complete for constraints with

```
CardImplies(Type $T$, Type $U$):
  every $a\,[m..n]$ in $U$ satisfy let $(\_,\ Min,\ Max)$ = PlusMinMax($T$, $a$);
                                   return $Min \geq m \wedge Max \leq n$

PlusMinMax(Type $T$, Symbol $a$):
  case $T$
  when $T_1 \otimes T_2$:   case (PlusMinMax($T_1$, $a$), PlusMinMax($T_2$, $a$))
                  when ((true, $Min_1$, $Max_1$), (true, $Min_2$, $Max_2$)):
                          return(true, $Min_1 + Min_2$, $Max_1 + Max_2$);
                  when ((true, $Min_1$, $Max_1$), (false,$Min_2$, $Max_2$)):
                          return(true, $Min_1$, $Max_1 + Max_2$);
                  when ((false, $Min_1$, $Max_1$), (true, $Min_2$, $Max_2$)):
                          return(true, $Min_2$, $Max_1 + Max_2$);
                  when ((false, $Min_1$, $Max_1$), (false, $Min_2$, $Max_2$)):
                          return(false, min($Min_1, Min_2$), $Max_1 + Max_2$);
  when $T_1 + T_2$:   let ($Plus_1$, $Min_1$, $Max_1$) = PlusMinMax($T_1$, $a$);
                  let ($Plus_2$, $Min_2$, $Max_2$) = PlusMinMax($T_2$, $a$);
                  return($Plus_1 \wedge Plus_2$, min($Min_1, Min_2$), max($Max_1, Max_2$));
  when $T_1\,[m..n]$: let ($Plus$, $Min$, $Max$) = PlusMinMax($T_1$, $a$);
                  if $Plus$ then return(true, $Min \times m$, $Max \times n$)
                          else return(false, $Min$, $Max \times n$)
  when $\epsilon$:          return(false, *, 0)
  when $a$:          return(true, 1, 1)
  when $b \neq a$:      return(false, *, 0)
```

Figure 3: Algorithm for implication of cardinality constraints.

shape $A^+ \mapsto\!\!\!\Rightarrow B^+$, then, for any $B \in \mathcal{SC}(U)$, $\mathcal{C}(U)$ implies the weaker constraint $S(U)^+ \mapsto\!\!\!\Rightarrow B^+$, which, together with $\mathit{SIf}(U)$, implies B^+.

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: the separate test for $T \models S(U)^+$ would involve the computation of something along the lines of $\mathcal{SC}(T)$; however, by restricting ourselves to the case when $T \models \mathit{upperS}(U)$, we only have to check that $\mathrm{N}(T) \Rightarrow \mathrm{N}(U)$, as proved below.

Definition 3.20 (Upper and Lower constraints)

$$\begin{array}{lll} \textit{Lower-bound:}\ \ \mathit{SIf}(U) & =_{\mathit{def}} & \textit{if}\ \neg\mathrm{N}(U)\ \textit{then}\ \{S(U)^+\}\ \textit{else}\ \emptyset \\ \textit{Upper-bound:}\ \ \mathit{upperS}(U) & =_{\mathit{def}} & \{\mathrm{upper}(S(U))\} \end{array}$$

Theorem 3.21 (Implication of $\mathit{SIf}(U)$ and $\mathit{upperS}(U)$) *For any two types T and U:*

$$T \models \mathit{SIf}(U) \cup \mathit{upperS}(U) \quad \Leftrightarrow \quad (\mathrm{N}(T) \Rightarrow \mathrm{N}(U)) \ \wedge\ S(T) \subseteq S(U)$$

Proof. ($\Rightarrow$) $T \models \mathit{upperS}(U)$ implies $S(T) \subseteq S(U)$, by Lemma 2.4. We prove now that $\neg\mathrm{N}(U) \Rightarrow \neg\mathrm{N}(T)$. Assume $\neg\mathrm{N}(U)$; then $T \models \mathit{SIf}(U)$ means $T \models S(U)^+$, hence $\epsilon \notin [\![T]\!]$, hence $\neg\mathrm{N}(T)$.

($\Leftarrow$) The implication $S(T) \subseteq S(U) \ \Rightarrow T \models \mathit{upperS}(U)$ is trivial. If $\mathrm{N}(U)$ is true, then $T \models \mathit{SIf}(U)$ holds trivially. If $\neg\mathrm{N}(U)$, then, by $\mathrm{N}(T) \Rightarrow \mathrm{N}(U)$, $\mathrm{N}(T)$ is false as well, hence every word of T contains a symbol from $S(T)$, hence a symbol from $S(U)$. ∎

The corresponding function UpperLowerImplies simply executes the test of Theorem 3.21.

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 3.22 *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 $\_\ [m..n]$ 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 [11], 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 [11], 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 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. By taking the lateral step of considering asymmetric inclusion, we have been able to get a swooping widening of the applicability of our approach, arriving at relieving the subtype from any limitation. Moreover, the resulting algorithm retains the quadratic complexity of the pure case, which is, frankly, quite amazing.

References

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

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

[3] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise DTDs from XML data. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 115–126. ACM, 2006.

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

[5] Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. Accepted for publication in *Information Systems*, 2008.

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

[7] J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. A logic your typechecker can count on: Unordered tree types in practice. In *PLAN-X, informal proceedings*, January 2007.

[8] Wouter Gelade, Wim Martens, and Frank Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. In *Proceedings of the 11th International Conference on Database Theory - ICDT 2007, Barcelona, Spain, January 10-12, 2007.*

[9] Giorgio Ghelli, Dario Colazzo, and Carlo Sartiani. Efficient inclusion for a class of XML types with interleaving and counting. In *Proceedings of the 11th International Symposium on Database Programming Languages, DBPL 2007, Vienna, Austria, September 23-24, 2007.*

[10] Giorgio Ghelli, Dario Colazzo, and Carlo Sartiani. Linear time membership for a class of XML types with interleaving and counting. To Appear in *ACM Conference on Information and Knowledge Management (CIKM)*, 2008.

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

A Proofs of Section 2.1

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

$$\begin{array}{ll} [\![T]\!] \neq \emptyset & (1) \\ a \in S(T) \Rightarrow \exists w \in [\![T]\!].\ a \in S(w) & (2) \end{array}$$

Proof. We prove both properties by mutual induction on the structure of T and by cases.

(1) The only non trivial case is $T = T'!$. Here, since T' contains at least one symbol a, by (2) and induction we have that there exists $w \in [\![T]\!]$ such that $a \in S(w)$, hence $w \neq \epsilon$, hence $[\![T'!]\!]$ contains w, hence $[\![T'!]\!] \neq \emptyset$.

(2) The only non trivial case is $T = T_1 \otimes T_2$ where only T_1 or T_2 contains a. Assume $a \in S(T_1)$. By induction we have that $\exists\ w_1 \in [\![T_1]\!].\ a \in S(w_1)$, and that T_2 is not empty (by induction and (1)). Hence, there exists a word $w \in w_1 \& w_2$, where $w_2 \in [\![T_2]\!]$, $w \in [\![T_1 \otimes T_2]\!]$ and $a \in S(w)$. ■

B Proofs of Section 2.3

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.*

Proof. Hereafter, we consider $A = \{F \in \mathscr{F} \mid U \models F\}$. So, if $\mathcal{C}()$ is a constraint-extraction function that is complete for our constraint language, we have $[\![\mathcal{C}(U)]\!] = [\![A]\!]$.

In Theorems 1 and 2 of [9] we proved that there exists a constraints-extraction function $\mathcal{C}'()$ that is exact for each conflict-free type. In particular, we proved that, for each $F \in \mathcal{C}'(U)$, we have $U \models F$. Hence, $\mathcal{C}'(U) \subseteq A$. Now, we have

$$[\![\mathcal{C}(U)]\!] = [\![A]\!] = [\![\mathcal{C}'(U) \cup (A \setminus \mathcal{C}'(U)]\!] = [\![\mathcal{C}'(U)]\!] \cap [\![A \setminus \mathcal{C}'(U)]\!]\ (*)$$

Since $\forall F \in (A \setminus \mathcal{C}'(U)).\ U \models F$, we have $[\![U]\!] \subseteq [\![A \setminus \mathcal{C}'(U)]\!]$, which, together with $(*)$ and $[\![U]\!] = [\![\mathcal{C}'(U)]\!]$ (recall that $\mathcal{C}'()$ is exact for U), yields $[\![\mathcal{C}(U)]\!] = [\![U]\!] \cap [\![A \setminus \mathcal{C}'(U)]\!] = [\![U]\!]$.

■

C Proofs of Section 3.1

Lemma 3.1 *For any type $T_1 \otimes T_2$:*

$$\begin{array}{lcl} T_1 \otimes T_2 \models A^+ & \Leftrightarrow & T_1 \models A^+\ \vee\ T_2 \models A^+ \\ T_1 \otimes T_2 \models A^{++} & \Leftrightarrow & T_1 \models A^+\ \vee\ T_2 \models A^+\ \vee\ (T_1 \models A^{++} \wedge T_2 \models A^{++}) \\ T_1 \otimes T_2 \models a^+ \Rrightarrow A^+ & \Leftrightarrow & (T_1 \models a^+ \Rrightarrow A^+\ \wedge\ T_2 \models a^+ \Rrightarrow A^+)\ \vee\ T_1 \otimes T_2 \models A^+ \end{array}$$

Proof. $(\Leftarrow)$ is immediate.

For $(\Rightarrow)$, assume that (a) $T_1 \otimes T_2 \models A^+$ and (b) $T_1 \not\models A^+$; we want to prove that $T_2 \models A^+$. By (b), $\exists w_1 \in [\![T_1]\!]$ such that $w_1 \not\models A^+$. Assume $w \in [\![T_2]\!]$; from (a), we deduce $w_1 \cdot w \models A^+$, hence, by $w_1 \not\models A^+$, we conclude that $w \models A^+$.

For $T_1 \otimes T_2 \models A^{++}$, assume (a) $T_1 \otimes T_2 \models A^{++}$, (b) $T_1 \not\models A^+$, and (c) $T_2 \not\models A^+$; we want to prove that $T_1 \models A^{++} \wedge T_2 \models A^{++}$. 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 $T_1 \models A^{++}$. $T_2 \models A^{++}$ is proved in the same way.

For $T_1 \otimes T_2 \models a^+ \mapsto A^+$, assume (a) $T_1 \otimes T_2 \models a^+ \mapsto A^+$ and (b) $\neg(T_1 \models a^+ \mapsto A^+ \ \wedge \ T_2 \models a^+ \mapsto A^+)$; we want to prove that (i) $T_1 \models A^+$ or (ii) $T_2 \models A^+$. By (b), either (iii) $T_1 \not\models a^+ \mapsto A^+$ or (iv) $T_2 \not\models a^+ \mapsto A^+$.

Assume (iii), hence $\exists w_1 \in [\![T_1]\!]$ such that $w_1 \models a^+$ and $w_1 \not\models A^+$; we now prove that (ii) follows. Assume $w \in [\![T_2]\!]$; from $w_1 \models a^+$ and (a), we deduce $w_1 \cdot w \models A^+$, hence, by $w_1 \not\models A^+$, we conclude that $w \models A^+$, hence (ii) holds. In the same way we prove that (iv) implies (i). From $T_1 \models A^+ \ \vee \ T_2 \models A^+$ we immediately deduce $T_1 \otimes T_2 \models A^+$: assume, wlog, $T_1 \models A^+$; any word of $T_1 \otimes T_2$ includes a word from T_1, which contains some symbols from A.

■

Theorem 3.3 (Completeness of $\mathcal{SC}(T)$ and $\mathcal{SC}^!(T)$].) *For any type T and $A \subseteq \Sigma$ with $A \neq \emptyset$:*

$$\begin{array}{lcll} A \in \mathcal{SC}(T) & \Leftrightarrow & T \models A^+ & (3.1) \\ A \in \mathcal{SC}^!(T) & \Leftrightarrow & T \models A^{++} & (3.2) \end{array}$$

Proof. We prove 3.1 and 3.2 by mutual induction and by cases.
Proof of (3.1).

Case $T = T_1 \otimes T_2$.

$$\begin{array}{rcll} A \in \mathcal{SC}(T_1 \otimes T_2) & \Longleftrightarrow & A \in \mathcal{SC}(T_1) \ \vee \ A \in \mathcal{SC}(T_2) & \text{by definition of } \mathcal{SC}(T_1 \otimes T_2) \\ & \Longleftrightarrow & T_1 \models A^+ \ \vee \ T_2 \models A^+ & \text{by induction} \\ & \Longleftrightarrow & T_1 \otimes T_2 \models A^+ & \text{by Lemma 3.1} \end{array}$$

Case $T = T_1 + T_2$. We first observe that $T_1 \models A^+ \ \wedge \ T_2 \models A^+ \iff T_1 + T_2 \models A^+$ (*), which we prove as follows. We have the following double implications: $T_1 \models A^+ \ \wedge \ T_2 \models A^+$ iff (for $i = 1, 2$) $\forall f \in [\![T_i]\!].f \models A^+$ iff $\forall f \in ([\![T_1]\!] \cup [\![T_2]\!]).f \models A^+$ iff $T_1 + T_2 \models A^+$. So,

$$\begin{array}{rcll} A \in \mathcal{SC}(T_1 + T_2) & \Longleftrightarrow & A \in \mathcal{SC}(T_1) \ \wedge \ A \in \mathcal{SC}(T_2) & \text{by definition of } \mathcal{SC}(T_1 + T_2) \\ & \Longleftrightarrow & T_1 \models A^+ \ \wedge \ T_2 \models A^+ & \text{by induction} \\ & \Longleftrightarrow & T_1 + T_2 \models A^+ & \text{by the above property (*)} \end{array}$$

Case $T = T_1\,[m..n]$. We first observe that $T_1\,[m..n] \models A^+ \iff T_1 \models A^+$ (**), which we prove as follows. Assume $T_1\,[m..n] \models A^+$, and, by aiming at a contradiction, assume $T_1 \not\models A^+$. This means that $\exists f \in [\![T_1]\!]$. $T \not\models A^+$, hence, if $f^m = f_1 \cdot \ldots \cdot f_m$ with $f_i = f$, we have $f^m \in [\![T_1\,[m..n]]\!]$ and $f^m \not\models A^+$, which contradicts $T_1\,[m..n] \models A^+$. Assume now that $T_1 \models A^+$ and, still by aiming at a contradiction, that $T_1\,[m..n] \not\models A^+$; this means that there exists $f \in [\![T_1\,[m..n]]\!]$ such that $f \not\models A^+$. Since $f = f_1 \cdot \ldots \cdot f_m$ with $f_i \in [\![T_1]\!]$, we have that $f_1 \not\models A^+$, hence the contradiction $T_1 \not\models A^+$. That said, we have now

$$\begin{array}{rcll} A \in \mathcal{SC}(T_1\,[m..n]) & \Longleftrightarrow & A \in \mathcal{SC}(T_1) & \text{by definition of } \mathcal{SC}(T_1\,[m..n]) \\ & \Longleftrightarrow & T_1 \models A^+ & \text{by induction} \\ & \Longleftrightarrow & T_1\,[m..n] \models A^+ & \text{by the previous property (**)} \end{array}$$

Case $T = T_1!$. By definition, we have $\mathcal{SC}(T_1!) = \mathcal{SC}^!(T_1)$. Hence, by induction, $T_1 \models A^{++}$, hence $\forall f \in ([\![T_1]\!] \setminus \{\epsilon\})$. $f \models A^+$, that is $\forall f \in [\![T_1!]\!]$. $f \models A^+$

Case $T = a$. The case is trivial since, for each $A \in \mathcal{SC}(T)$, we have $a \in A$, and since the only word in T is a.

Case $T = \epsilon$. The case is trivial since, for every A, $A \notin \mathcal{SC}(\epsilon)$ and $\epsilon \not\models A$

Proof of (3.2).

Case $T = T_1 \otimes T_2$. The case it straightforward:

$$\begin{array}{rcll}
T_1 \otimes T_2 \models A^{++} & \iff & T_1 \models A^+ \ \vee \ T_2 \models A^+ & \\
 & & \vee \ (T_1 \models A^{++} \wedge T_2 \models A^{++}) & \text{by Lemma 3.1} \\
 & \iff & A \in \mathcal{SC}(T_1) \ \vee \ A \in \mathcal{SC}(T_2) & \\
 & & \vee \ (A \in \mathcal{SC}^!(T_1) \ \wedge \ A \in \mathcal{SC}^!(T_2)) & \text{by induction} \\
 & \iff & A \in \mathcal{SC}^!(T_1 \otimes T_2) & \text{by definition of } \mathcal{SC}^!(T_1 \otimes T_2)
\end{array}$$

Case $T = T_1 + T_2$. The case is similar to the previous correspondent one. We first observe that $T_1 \models A^{++} \ \wedge \ T_2 \models A^{++} \iff T_1 + T_2 \models A^{++}$ (*), which we prove as follows. We have the following double implications: $T_1 \models A^{++} \ \wedge \ T_2 \models A^{++}$ iff (for $i = 1, 2$) $\forall f \in (\llbracket T_i \rrbracket \setminus \{\epsilon\}).f \models A^+$ iff $\forall f \in (\llbracket T_1 \rrbracket \cup \llbracket T_2 \rrbracket) \setminus \{\epsilon\}.\ f \models A^+$ iff $T_1 + T_2 \models A^{++}$. So,

$$\begin{array}{rcll}
A \in \mathcal{SC}^!(T_1 + T_2) & \iff & A \in \mathcal{SC}^!(T_1) \ \wedge \ A \in \mathcal{SC}^!(T_2) & \text{by definition of } \mathcal{SC}^!(T_1 + T_2) \\
 & \iff & T_1 \models A^{++} \ \wedge \ T_2 \models A^{++} & \text{by induction} \\
 & \iff & T_1 + T_2 \models A^{++} & \text{by the above property (*)}
\end{array}$$

Case $T = T\,[m..n]$. Again, the case is similar to the previous correspondent one. We first observe that $T_1\,[m..n] \models A^{++} \iff T_1 \models A^{++}$ (**), which we prove as follows. Assume $T_1\,[m..n] \models A^{++}$, and, by aiming at a contradiction, assume $T_1 \not\models A^{++}$. This means that $\exists f \in \llbracket T_1 \rrbracket.\ f \neq \epsilon \wedge T \not\models A^+$, hence, if $f^m = f_1 \cdot \ldots \cdot f_m$ with $f_i = f$, we have $f^m \in \llbracket T_1\,[m..n] \rrbracket$ and $f^m \neq \epsilon$ and $f^m \not\models A^+$, which contradicts $T_1\,[m..n] \models A^{++}$. Assume now that $T_1 \models A^{++}$ and, still by aiming at a contradiction, that $T_1\,[m..n] \not\models A^{++}$. This means that there exists $f \in \llbracket T_1\,[m..n] \rrbracket$ such that $f \neq \epsilon$ and $f \not\models A^+$. Since $f = f_1 \cdot \ldots \cdot f_m$ with $f_i \in \llbracket T_1 \rrbracket$, we have that at least one f_j is such that $f_j \neq \epsilon$ and $f_j \not\models A^+$, hence the contradiction $T_1 \not\models A^{++}$. That proved, we have now

$$\begin{array}{rcll}
A \in \mathcal{SC}^!(T_1\,[m..n]) & \iff & A \in \mathcal{SC}^!(T_1) & \text{by definition of } \mathcal{SC}^!(T_1\,[m..n]) \\
 & \iff & T_1 \models A^{++} & \text{by induction} \\
 & \iff & T_1\,[m..n] \models A^{++} & \text{by the previous property (**)}
\end{array}$$

Case $T = T'!$. Simple induction.

Case $T = a$. Trivial since, for each $A \in \mathcal{SC}^!(T)$, we have $a \in A$, and since the only word in T is a.

Case $T = \epsilon$. Trivial since, for each A in Σ such that $A \neq \emptyset$, we have $A \in \mathcal{SC}^!(\epsilon)$ as well, and $\epsilon \models A^{++}$, which trivially holds since $\llbracket T \rrbracket \setminus \{\epsilon\} = \emptyset$.

∎

Property 3.4 (Union) *For any word w and constraint $A^+ \Rrightarrow B^+$:*

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

Proof. $(\Rightarrow)$

We distinguish two cases: $S(w) \cap A = \emptyset$ and $S(w) \cap A \neq \emptyset$. In the first case we have that $\forall a \in A.\ S(w) \cap \{a\} = \emptyset$, which directly implies the thesis. In the second case, by hypothesis, we have that $S(w) \cap B \neq \emptyset$. So, if $A_1 = S(w) \cap A$ and $A_2 = A \setminus A_1$, we have $\forall a \in A_i.\ w \models a^+ \Rrightarrow B^+$, for $i = 1, 2$, hence the thesis, since $A = A_1 \cup A_2$.

$(\Leftarrow)$

Again, we distinguish two cases: $S(w) \cap A = \emptyset$ and $S(w) \cap A \neq \emptyset$. In the first case, $w \models A^+ \mapsto B^+$ trivially holds. In the second case, by hypothesis we have that there exists $a \in S(w) \cap A$ such that $w \models a^+ \mapsto B^+$. This means that $A \cap S(w) \neq \emptyset$ and $B \cap S(w) \neq \emptyset$, which directly implies $w \models A^+ \mapsto B^+$ ■

D Proofs of Section 3.2

Property 3.7 $\quad T \models a \prec b \;\Leftrightarrow\; a \neq b$ *and* $(b,a) \notin \mathcal{P}(T)$

Proof. Recall that, by definition, for each formula $a \prec b$ we have $a \neq b$ (see Section 2.2). The proof proceeds as follows:

$$\begin{array}{rcl} T \models a \prec b & \Longleftrightarrow & \forall w \in [\![T]\!].\ w \models a \prec b \\ & \Longleftrightarrow & \nexists w \in [\![T]\!].\ w \not\models a \prec b \\ & \Longleftrightarrow & \nexists w \in [\![T]\!].\ w = w1 \cdot b \cdot w_2 \cdot a \cdot w_3 \\ & \Longleftrightarrow & (b,a) \notin \mathcal{P}(T) \end{array}$$

■

Property 3.9 (Pairs)

$$\begin{array}{lll} 1) \ (a,b) \in \mathcal{P}(T) & \Leftrightarrow & \exists a_i, b_j \in L(T).\ LCA_{L(T)}[a_i, b_j] \in \{+_r, \otimes_r, \&_1, \cdot_1^{\rightarrow}\} \\ 2) \ (b,a) \notin \mathcal{P}(T) & \Leftrightarrow & \forall a_i, b_j \in L(T).\ LCA_{L(T)}[a_i, b_j] \in \{+_1, \cdot_1^{\rightarrow}\} \\ 3) \ T \models a \prec b & \Leftrightarrow & \forall a_i, b_j \in L(T).\ LCA_{L(T)}[a_i, b_j] \in \{+_1, \cdot_1^{\rightarrow}\} \end{array}$$

Proof. (1)($\Rightarrow$)

By induction on the structure of T and by case distinction.

Case $T = T_1 + T_2$, hence $L(T) = L(T_1) +_1 L(T_2)$. We have that $(a,b) \in \mathcal{P}(T)$ implies $(a,b) \in \mathcal{P}(T_1)$ or $(a,b) \in \mathcal{P}(T_2)$. Assume the first, wlog, and the thesis follows by induction.

Case $T = T_1 \cdot T_2$, hence $L(T) = L(T_1) \cdot_1 L(T_2)$. Here, we have $(a,b) \in \mathcal{P}(T)$ iff $(a,b) \in \mathcal{P}(T_1)$ or $(a,b) \in \mathcal{P}(T_2)$ or $a \in S(T_1) \wedge b \in S(T_2)$. In the first two cases, we conclude by induction. In the third case, we have at least one a_i in T_1 and one b_j in T_2, and their LCA is $\cdot_1^{\rightarrow}$.

Case $T = T_1 \& T_2$, hence $L(T) = L(T_1) \&_1 L(T_2)$. Here, we have $(a,b) \in \mathcal{P}(T)$ iff $(a,b) \in \mathcal{P}(T_1)$ or $(a,b) \in \mathcal{P}(T_2)$ or $a \in S(T_1) \wedge b \in S(T_2)$ or $a \in S(T_2) \wedge b \in S(T_1)$. In the first two cases, we conclude by induction. In the third case, we have at least one a_i in T_1 and one b_j in T_2, and their LCA is $\&_1$. The fourth case is similar.

Case $T = T_1\,[m..n]$. If $n = 1$, then the case easily follows by induction. If $n > 1$, then the LCA of any pair of symbols is in $\{+_r, \otimes_r\}$.

The other cases are trivial.

(1)($\Leftarrow$) $\exists a_i, b_j \in L(T).\ LCA_{L(T)}[a_i, b_j] \in \{+_r, \otimes_r, \&_1, \cdot_1^{\rightarrow}\} \Rightarrow (a,b) \in \mathcal{P}(T)$.

By induction on the structure of T and by case distinction.

Case $T = T_1 + T_2$, hence $L(T) = L(T_1) +_1 L(T_2)$, hence a_i and b_j are in the same side of T (their LCA is not $+_1$, by hypothesis), hence, by induction, that side contains the pair (a,b), hence we conclude.

Case $T = T_1 \cdot T_2$, hence $L(T) = L(T_1) \cdot_1 L(T_2)$, hence either a_i and b_j are in the same side of T or a_i is in T_1 and b_j is in T_2. In the first case, we conclude by induction, in the second case, by Lemma 2.4, we have two words $w_a \in [\![T_1]\!]$ and $w_b \in [\![T_2]\!]$ with $w_a \models a^+$ and $w_b \models b^+$, hence $w_a \cdot w_b$ is in $[\![T]\!]$ and contains the pair (a,b)

Case $T = T_1 \& T_2$, hence $L(T) = L(T_1) \&_1 L(T_2)$, hence either a_i and b_j are in the same side of T or a_i is in T_1 and b_j is in T_2 or a_i is in T_2 and b_j is in T_1. In the first case we conclude by induction, in the second and in the third case we reason as in the second case of $T_1 \cdot T_2$.

Case $T = T_1\,[m..n]$. If $n = 1$, then the case easily follows by induction. Otherwise, $L(T)$ is not equal to $(L(T_1))\,[m..n]$, since all operators have shape $\circledast_r$ in $L(T)$, while they may have shape $\circledast_1$ in $L(T_1)$, hence we cannot conclude by induction. However, from $\exists a_i, b_j \in L(T)$ we deduce, by Lemma 2.4, that we have two words w_a and w_b in $[\![T_1]\!]$ with $w_a \models a^+$ and $w_b \models b^+$, hence $w_a \cdot w_b$ is in $[\![T_1\,[m..n]]\!]$ and contains the pair (a, b).

The other cases are trivial.

Property 2) follows immediately from 1); Property 3) follows from 2) and by Property 3.7.

■

Property 3.11 (Completeness of $\mathcal{OC}_g(T)$) *$\mathcal{OC}_g(T)$ is complete for the set of constraints with shape $a \prec b$ and $\{a, b\} \subseteq S(T)$.*

Proof. By Property 3.9 and by definition of $\mathcal{OC}_g(T)$ we have that

$$\mathcal{OC}_g(T) = \{a \prec b \mid \{a, b\} \subseteq S(T) \wedge T \models a \prec b\}$$

hence $[\![\mathcal{OC}_g(T)]\!] = [\![\{a \prec b \mid T \models a \prec b\}]\!]$.

■

E Proofs of Section 3.3

Lemma 3.13 *For any type T and symbol a: $\mathsf{SMin}^*(T, a) = \mathrm{Min}^*(T, a)$*

Proof. By induction on the structure of T and by case distinction.

Case $T = T_1 + T_2$. We have $\mathsf{SMin}^*(T, a) = \min(\mathsf{SMin}^*(T_1, a), \mathsf{SMin}^*(T_2, a))$, so the case easily follows by induction.

Case $T = T_1 \otimes T_2$. We distinguish four cases:

- $T_1 \models a^+ \wedge T_2 \models a^+$. Every word of T_1 contains at least $\mathsf{SMin}^*(T_1, a)$ a's, and the same for T_2, hence every word of T contains at least $\mathsf{SMin}^*(T_1, a) + \mathsf{SMin}^*(T_2, a)$, hence $\mathsf{SMin}^*(T, a) = \mathsf{SMin}^*(T_1, a) + \mathsf{SMin}^*(T_2, a)$, and we conclude by induction.

- $T_1 \models a^+ \wedge T_2 \not\models a^+$. Since every word of T_1 contains at least $\mathsf{SMin}^*(T_1, a)$ a's, we have that $\mathsf{SMin}^*(T, a) \geq \mathsf{SMin}^*(T_1, a)$. Since we have a word $w_2 \in [\![T_2]\!]$ that contains no a's, by concatenating w_2 with a word from T_1 with $\mathsf{SMin}^*(T_1, a)$ a's we prove that $\mathsf{SMin}^*(T, a) \leq \mathsf{SMin}^*(T_1, a)$, hence $\mathsf{SMin}^*(T, a) = \mathsf{SMin}^*(T_1, a)$, and we conclude by induction.

- $T_1 \not\models a^+ \wedge T_2 \models a^+$. As in the case above.

- $T_1 \not\models a^+ \wedge T_2 \not\models a^+$. If $a \notin S(T)$, then $\mathsf{SMin}^*(T_i, a) = \mathrm{Min}^*(T_i, a) = *$, for $i = 1, 2$, so the case follows by induction, since $\min(*, *) = *$. If $a \in S(T)$, we distinguish two sub-cases. If $a \in S(T_1)$ and $a \in S(T_2)$, then $\min_{w \in ([\![T]\!] \cap [\![a^+]\!])} |w|_a = \min(\mathsf{SMin}^*(T_1, a), \mathsf{SMin}^*(T_2, a))$, and we conclude by induction. If a appears in one subterm only, let's say T_1, then $\mathsf{SMin}^*(T, a) = \mathsf{SMin}^*(T_1, a) = min(\mathsf{SMin}^*(T_1, a), *)$, hence, by induction $\mathsf{SMin}^*(T, a) = min(\mathrm{Min}^*(T_1, a), \mathrm{Min}^*(T_2, a))$.

Case $T = T_1\,[m..n]$. If $a \notin S(T_1)$, then both $\mathsf{SMin}^*(T,a) = *$ and $\mathsf{SMin}^*(T_1,a) = *$, and the thesis follows by induction, since $m \times * = *$. If $a \in S(T_1)$, then we distinguish two sub-cases. If $T \models a^+$, then each T_1 word contains at least $\mathsf{SMin}^*(T_1,a)$ occurrences of a, hence each word of T contains at least $\mathsf{SMin}^*(T_1,a) \times m$ occurrences of a, and the thesis follows by induction. If $T \not\models a^+$, then we have a word w_a in T_1 with $\mathsf{SMin}^*(T_1,a)$ occurrences of a and a word w' with no occurrences of a; the concatenation of one w_a with many w' gives us a word of T with $\mathsf{SMin}^*(T_1,a)$ occurrences of a, and we have no way to find a word in $[\![T]\!] \cap [\![a^+]\!]$ with less occurrences of a, hence $\mathsf{SMin}^*(T,a) = \mathsf{SMin}^*(T_1,a)$, and the thesis follows by induction.

The other cases are trivial.

■

Lemma 3.16 *For any type T and symbol a:* $\mathsf{SMax}^0(T,a) = \mathrm{Max}^0(T,a)$

Proof. By induction on the structure of T and by case distinction.

Case $T = T_1 + T_2$. In the case that $\mathsf{SMax}^0(T_1+T_2,a) = *$, we have that either $\mathsf{SMax}^0(T_1,a) = *$ or $\mathsf{SMax}^0(T_2,a) = *$. Assume, wlog, that $\mathsf{SMax}^0(T_1,a) = *$. By induction we have $\mathrm{Max}^0(T_1,a) = *$, and therefore $\mathrm{Max}^0(T_1+T_2,a) = *$ by definition.

In the case that $\mathsf{SMax}^0(T_1+T_2,a) \in \mathbb{N}$, we have $\mathsf{SMax}^0(T_i,a) \in \mathbb{N}$ for $i = 1,2$, and the following equivalences are immediate:

$$\begin{array}{rcll} \mathsf{SMax}^0(T_1+T_2,a) & = & \max(\mathsf{SMax}^0(T_1,a), \mathsf{SMax}^0(T_2,a)) & \\ & = & \max(\mathrm{Max}^0(T_1,a), \mathrm{Max}^0(T_2,a)) & \text{by induction} \\ & = & \mathrm{Max}^0(T_1+T_2,a) & \end{array}$$

Case $T = T_1 \otimes T_2$. In the case that $\mathsf{SMax}^0(T_1 \otimes T_2,a) = *$, we have that either $\mathsf{SMax}^0(T_1,a) = *$ or $\mathsf{SMax}^0(T_2,a) = *$. Assume, wlog, that $\mathsf{SMax}^0(T_1,a) = *$. By induction we have $\mathrm{Max}^0(T_1,a) = *$, and therefore $\mathrm{Max}^0(T_1 \otimes T_2,a) = *$ by definition.

In the case that $\mathsf{SMax}^0(T_1 \otimes T_2,a) \in \mathbb{N}$, we have $\mathsf{SMax}^0(T_i,a) \in \mathbb{N}$ for $i = 1,2$, and the following equivalences are immediate:

$$\begin{array}{rcll} \mathsf{SMax}^0(T_1 \otimes T_2,a) & = & \mathsf{SMax}^0(T_1,a) + \mathsf{SMax}^0(T_2,a) & \\ & = & \mathrm{Max}^0(T_1,a) + \mathrm{Max}^0(T_2,a) & \text{by induction} \\ & = & \mathrm{Max}^0(T_1+T_2,a) & \end{array}$$

The other cases are immediate.

■

Lemma 3.17 $T \models a?[m..n] \;\Leftrightarrow\; m \leq \mathrm{Min}^*(T,a) \;\wedge\; \mathrm{Max}^0(T,a) \leq n$

Proof. By Lemmas 3.13 and 3.16, we can substitute $\mathrm{Min}^*(T,a)$ with $\mathsf{SMin}^*(T,a)$ and $\mathsf{SMax}^0(T,a)$ with $\mathsf{SMax}^0(T,a)$.

($\Rightarrow$) Let $T \models a?[m..n]$. If $a \notin S(T)$, then $\mathsf{SMin}^*(T,a) = *$ and $\mathsf{SMax}^0(T,a) = 0$, hence the thesis. Assume $a \in S(T)$; $T \models a?[m..n]$ implies that the minimum number of times a appears in any word of $[\![T]\!] \cap [\![a^+]\!]$ is greater than m, hence $m \leq \mathsf{SMin}^*(T,a)$, and similarly for $\mathsf{SMax}^0(T,a)$.

($\Leftarrow$) Assume $m \leq \mathsf{SMin}^*(T,a) \;\wedge\; \mathsf{SMax}^0(T,a) \leq n$. If $a \notin S(T)$, then $T \models a?[m..n]$ holds trivially, since no word in T may violate it. Assume $a \in S(T)$. Then, $\mathsf{SMin}^*(T,a)$ counts the minimum number of times that a appears in a word of $[\![T]\!] \cap [\![a^+]\!]$, hence $m \leq \mathsf{SMin}^*(T,a)$ implies that $T \models a?[m..*]$. Similarly, $\mathsf{SMax}^0(T,a)$ counts the maximum number of times that a appears in a word of T, hence $\mathsf{SMax}^0(T,a) \leq n$ implies that $T \models a?[1..n]$. ■

Corollary 3.19 *$ZeroMinMax_g(T)$ is complete for constraints with shape $a?[m..n]$ and $a \in S(T)$.*

Proof. Consider the set $S = \{a?[m..n] \mid T \models a?[m..n] \;\wedge\; a \in S(T)\}$. By Lemma 3.17 we have that $a?[m..n] \in S \iff m \leq \mathrm{Min}^*(T,a) \;\wedge\; \mathrm{Max}^0(T,a) \leq n$. From this it follows that

$$[\![S]\!] = \cap_{a?[m..n]\in S}[\![a?[m..n]]\!] = \cap_{a?[m..n]\in ZeroMinMax_g(T)}[\![a?[m..n]]\!] = [\![ZeroMinMax_g(T)]\!]$$

■

F Proofs of Section 3.5

Theorem 3.22 *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**.*

Proof. We have that, for the conflict-free type U, the *exact* constraint characterization is given by

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

where each constraint extraction function is defined as illustrated in the body of the paper (and in [9]). We have the following double implications:

$CoImplies$(T,U) returns **true**	$\iff$	$\forall A^+ \mapsto B^+ \in \mathcal{CC}(U).\ T \models A^+ \mapsto B^+$
		by Theorems 3.5 and 3.4
$OrderImplies$(T,U returns **true**	$\iff$	$\forall a \prec b \in \mathcal{OC}(U).\ T \models a \prec b$
		by Property 3.9
$CardImplies$(T,U) returns **true**	$\iff$	$\forall a?[m..n] \in ZeroMinMax(U).T \models a?[m..n]$
		by Lemma 3.17
$UpperLowerImplies$(T,U) returns **true**	$\iff$	$T \models SIf(U)$
		by Theorem 3.21

So we have that all of CoImplies(T, U), OrderImplies(T, U), CardImplies(T, U), and UpperLowerImplies(T, U) return **true** if and only if $T \models \mathcal{C}(U)$ if and only if (Mixed Subtyping Theorem 2.8) $[\![T]\!] \subseteq [\![U]\!]$. ■