Linear Inclusion for XML Regular Expression Types

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

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

Luca Pardini
Dipartimento di Informatica - Università di Pisa - Italy
pardini@di.unipi.it

Carlo Sartiani
DMI - Università della Basilicata - Italy
sartiani@gmail.com

ABSTRACT
Type inclusion is a fundamental operation in every type-checking compiler, but it is quite expensive for XML manipulation languages. We recently presented an inclusion checking algorithm for an expressive family of XML type languages which is polynomial, but runs in quadratic time both in the best and in the worst cases. We present here an algorithm that has a linear-time backbone, and resorts to the quadratic approach for some specific parts of the compared types. Our experiments show that the new algorithm typically runs in linear time, hence can be used as a building block for a practical type-checking compiler.

Categories and Subject Descriptors
H.2 [Database Management]: Miscellaneous

General Terms
Theory

Keywords
XML, type inclusion

1. INTRODUCTION

Regular Expressions extended with counting and interleaving (Extended REs, or EREs) are an abstract model of the basic mechanism of typical type systems for XML manipulation languages, such as XML Schema Definition, DTDs, Relax NG [11, 3, 4, 5]. Such ERE-based languages are expressive and natural, but type inclusion is intractable, being EXPTIME-complete for basic REs, and EXPSPACE-complete once counting and interleaving are added [8].

Type inclusion is the basic operation of any type-checking compiler, and is typically invoked for every variable assignment, function definition, and function call, hence it must be extremely fast. We offered a first solution in [9, 6]. In these papers we introduced conflict-free EREs, which are a restricted form of EREs that closely correspond to the kind of EREs that schema designers actually use, and that admit a quadratic-time inclusion algorithm.

Unfortunately, that algorithm is quadratic both in the best and in the worst case, which is hardly acceptable, since in the most common situations a linear test is quite obviously sufficient. We present here a new algorithm that is linear-time in the common situations and resorts to the quadratic approach only for those specific portions of the two types where it seems to be necessary. Our experiments show that the new algorithm typically runs in linear time.

The algorithm we presented in [6] extracts a set of “constraints” from the candidate supertype and verifies that each constraint is satisfied by the subtype, which involves a quadratic amount of work, even in cases when the two types are very similar, or equal. Our new algorithm has a more traditional “structural” approach: it visits both types in parallel from the top, matching the topmost operator and recurring on the children. This approach is not complete, since EREs may be included in cases when the topmost operators are permuted in quite complex ways, hence the structural approach should be combined with the quadratic approach to yield a complete algorithm. A naïve algorithm could just apply an incomplete set of structural rules and, when these fail, go back to the original types and apply the quadratic algorithm. Hence the algorithm would be better than the quadratic one in those cases when the structural rules suffice, but would impose an overhead otherwise. Unfortunately, choosing the optimal set of structural rules is impossible under this approach. A simple set of rules would be very effective in only a small set of cases. A richer set of rules would enlarge the set of cases where the algorithm is effective, but would impose a higher overhead in those cases where the structural work is useless. Our understanding of the “typical” workload of a type-checking compiler is too limited for a reasoned choice of an optimal set of rules.

We overcome this problem by designing a set of no-backtracking structural rules: whenever these rules rewrite a comparison into a set of simpler comparisons, the new set is not just a sufficient condition for the previous comparison, but it is equivalent. In this way, once a comparison that
matches no rule is found, we do not need to go back to the initial types, but we can apply the quadratic algorithm to the smaller type fragments, so that the algorithm is always convenient over the baseline. These no-backtracking rules for EREs are the main contribution of this work, together with a technique to select the applicable rule in constant time.

Paper Outline
The paper is structured as follows. In Section 2 we describe the ERE type language we are using here. In Section 3 we describe the algorithm and prove its correctness. In Sections 5 and 6 we revise some related works and conclude.

2. TYPES

2.1 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 \(\epsilon\).

\[
v\&w \overset{\text{def}}{=} \{v_1\cdot w_1, \ldots, v_n\cdot w_n | v_1, \ldots, v_n = v, w_1, \ldots, w_n = w, v_i, w_i \in \Sigma^*, n > 0\}
\]

\[
L_1\&L_2 \overset{\text{def}}{=} \bigcup_{w_1 \in L_1, w_2 \in L_2} (v\&w)
\]

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 \(\Sigma\):

\[
T ::= \epsilon | a | T[m..n] | T + T | T \cdot T | T\&T | T!
\]

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 \(\mathbb{N}\cup \{\epsilon\}\), where \(\epsilon\) behaves as \(+\infty\), i.e., for any \(n \in \mathbb{N}\), \(\ast > n\).

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 type semantics (it is formalized shortly). The mandatory presence of an \(a\) subterm in \(T\) guarantees that \(T\) contains at least one word that is different from \(\epsilon\), hence \(T!\) is never empty, 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 \(T\).

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

\[
[\epsilon] = \{\epsilon\}
\]

\[
[T_1 + T_2] = [T_1] \cup [T_2]
\]

\[
[T_1 \cdot T_2] = [T_1] \cdot [T_2]
\]

\[
[T_1\&T_2] = [T_1]\&[T_2]
\]

\[
[T!] = [T] \setminus \{\epsilon\}
\]

\[
[T[m..n]] = \{w | w = w_1, \ldots, w_j, \forall i \in 1..j, w_i \in [T], m \leq j \leq n\}
\]

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

We use \(N(T)\) to indicate that \(T\) is nullable, that is \(\epsilon \in [T]\). \(N(T)\) has the following definition (observe that \(N(T[m..n]) = N(T)\) because \(m\) cannot be 0).

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

\[
N(\epsilon) = \text{true}
\]

\[
N(T[m..n]) = N(T)
\]

\[
N(T!) = \text{false}
\]

\[
N(T + T') = N(T) \text{ or } N(T')
\]

\[
N(T \cdot T') = N(T) \text{ and } N(T')
\]

From this definition, it easily follows that \(N(T)\) can be computed in \(O(|T|)\).

2.3 Conflict-free Types
In [6] we proved that inclusion for the above language, which is EXPSPACE-complete in general, can be verified in quadratic time when the right hand-side of the comparison (the supertype) is conflict-free. Conflict-free types are 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 \(U_1 \oplus U_2\), 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 DTs and XSD (XML Schema Definition) schemas use repetition almost exclusively as \(a^+\) or as \((a + \ldots + z)^+\) where \(op \in \{+, \cdot\}\), see [2]), which can be immediately translated to types that only count symbols, thank to the \(U_1 \oplus U_2\) and \(U_1\) operators. For instance, \((a + \ldots + z)^+\) can be expressed as \((a^\infty \& \ldots \& z^\infty)\), where \(a^\infty\) is a shortcut for \(a[1..\infty] + \epsilon\), while \((a + \ldots + z)^+\) can be expressed as \((a^\infty \& \ldots \& z^\infty)!\).

It should be observed that the quadratic algorithm imposes no restriction on the left hand-side of the comparison (the subtypes), hence it can be safely applied in any context where the subtypes is automatically inferred and the supertype is human-designed or human-declared. This situation is very common during the semantic analysis of programs being compiled: indeed, the compiler infers types for any
expression in the language and compares those types with a supertype declared in the program code in the three fundamental situations where subtype-checking is used: variable assignment, function definition, function call.

Our algorithm will adopt the same conflict-free restriction for the supertype, and will revert to the algorithm in [6] in those cases when it is not able to proceed.

3. STRUCTURAL APPROACH

3.1 Introduction

Our algorithm is based on the assumption that subtyping would typically be applied to types that are very similar, such as $b \leq a + b + c$, or $a - b < a - b$. Most of these cases may be proved by combining transitivity, associativity and commutativity with some obvious structural rules, such as:

- Monotonicity: $T_1 \leq U_1 \land T_2 \leq U_2 \implies T_1 \odot T_2 \leq U_1 \odot U_2$
- Union: $T_1 \leq T_1 + T_2$
- Product: $T_1 \cdot T_2 \leq T_1 \& T_2$

Our algorithm is defined, as usual, by a set of deduction rules which are meant to be used to deterministically reduce an expression to ($\odot$) by combining transitivity, associativity and commutativity with some obvious structural rules, such as:

- **Property**
  
  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.

  * **Definition**

  $\alpha \leq \beta$ is conflict-free, and if the symbols of $\alpha$ and $\beta$ have no common symbols, then $\alpha$ and $\beta$ can be separated.

  * **Property**

  $\forall \alpha \in \mathcal{T}$, $\exists \beta \in \mathcal{T}$ such that $\alpha \leq \beta$.
detail is the use of nullability in the premises. In the three
(DIVIDE⊗⊗) rules the nullability condition is needed for the
direct implication to be sound. If the first nullability condition
were violated, we would have ε ∈ [T1], a non empty word w2 in [T2]
and ε ∉ [U1]. Hence, w2 would belong to T1 ⊗ T2 and w2 would not contain any symbol from U1, hence
it could not belong to U1 ⊗ U2, which only contains words
that contain some symbol from U1. Observe that this compi-
lation derives from the use of ≤k, since T1 ⊆ U1 would
imply N(T1) ⇒ N(U1), and similarly for T2.
Nullability of U2 in the (NFOCUS⊗⊗) rules is not related to
≤k, but it is the kernel of the rule itself, which is based
on the observation that, if ε ∈ [U2], then [U1] ⊆ [U1 ⊗ U2].
The same observation, applied to both factors, justifies the
(NDIVIDE) rules.

Observe that this set of rules is by no means complete.
For example, one may add the following rule, to take com-
mutativity of ‘+’ into account.

\[
\begin{align*}
S(T_1) &\subseteq S(U_2) \land S(T_2) \subseteq S(U_1) \\
T_1 &\leq_k U_2, T_2 \leq_k U_1 \\
T_1 + T_2 &\leq_k U_1 + U_2 \\
\end{align*}
\]

(REV-DIV++)

Unfortunately, associativity is at least as important as com-
mutativity, but is far more difficult to deal with. We hence
present here just a minimal set of rules, to illustrate the ba-
sic ideas, and we discuss our approach to associativity and
commutativity later, in Section 3.4.

3.3 The algorithm

The algorithm is described below. It first calls the auxil-
ary algorithm preprocess(T,U), which prepares the types
for efficient subtype checking, and which we describe later
on. The algorithm then verifies whether a rule r exists such
that Cond_r(T,U) holds. If the rule exists, it is applied,
and the problem is split in simpler problems, to be solved in
subsequent iterations of the while-loop. When we find a sub-
problem where no rule is applicable, the algorithm resorts to
the quadratic algorithm Oracle(T,U) [6].

Check(T,U)
1  preprocess(T,U)
2  push (T,U) in todo
3  while (todo ≠ ∅)
4    do
5      pick (T,U) from todo
6      if (∃ r such that Cond_r(T,U))
7        then push Ti ≤ U₁, ... , Tn ≤ U₀ in todo
8        else if (not oracle(T,U))
9          then return true
10      return true

The following theorems specify some sufficient conditions
about Cond_r(T,U) which guarantee that the algorithm is
correct and is linearOracle, meaning that it runs in linear
time, apart from the time spent by ORACLE.

Theorem 3.3 (Correctness) The structural algorithm is
correct if, for any rule r and any pair of types T and U, the
following holds.

\[
\begin{align*}
\text{Cond}_r(T,U) &\Rightarrow (T \leq U \Leftrightarrow T_i \leq U_i \land \cdots \land T_n \leq U_n) \\
\end{align*}
\]

Table 1: The structural rules
Theorem 3.4 (Linearity\textsuperscript{Oracle}) The structural algorithm is linear\textsuperscript{Oracle} provided that:

- \textbf{Preprocess} \((T, U)\) is in \(O(|T| + |U|)\);

- every rule consumes some input, i.e., for any rule \(r\) an integer \(k_r > 0\) exists such that, for any pair of types \(T, U\):
  \(|(T[|T|] + |U[|U||] + \ldots + |T_n]| + |U_n]|)| \geq k_r\)

- the test “find \(r\) such that \(\text{Cond},(T, U)\) runs in time \(O(|T| + |U|)\) when is negative, and in time \(O(k_r)\) when it finds the rule \(r\).

We have now to show that the rules are correct, meaning that each rule corresponds to a double implication. Finally we show that the algorithm is linear, by showing that rule selection can be performed in constant time. Linearity\textsuperscript{Oracle} follows by Theorem 3.4, since every rule consumes one symbol, i.e., \(k_r = 1\) for every rule \(r\).

3.3.1 Correctness

In the following, we will use the notation \(S_1 \# S_2\) to indicate that the two symbol sets are disjoint.

\textbf{Lemma 3.5} Let \(U\) be a conflict-free type. The following 4 properties hold:

\(w \in (w_1 & w_2) \Rightarrow S(w_1) \subseteq S(U) \wedge S(w_2) \# S(U) \Rightarrow w \mid S(U) = w_1\) (1)

\(w \in [T_1 \otimes T_2] \Rightarrow S(T_1) \subseteq S(U) \wedge S(T_2) \subseteq S(U) \Rightarrow w \mid S(U) \subseteq [T_1]\) (2)

\(S(w) \subseteq S(T_1) \wedge S(T_2) \Rightarrow w \in [T_1]\) (3)

\(S(w_1) \subseteq S(T_1) \wedge S(w_2) \subseteq S(T_2) \Rightarrow w \in [T_1]\) (4)

\textbf{Lemma 3.6} For each type \(T\) we have that \([T] \neq \emptyset\).

We can now present our main technical result, the bidirectionality of all the rules. The result we present here is slightly stronger than needed, since we group the nullability conditions together with the \(\leq k\) premises, instead of the symbol inclusion precondition. In practice, this means that the algorithm may just use symbol inclusion to choose the rule, and, after the choice has been performed, if the nullability test went wrong the algorithm may return an inclusion failure. We do not elaborate more on this point, but it may represent an important optimization.

\textbf{Theorem 3.7 (decomposition)} If both \(S(T_1) \subseteq S(U_1)\) and \(S(T_2) \subseteq S(U_2)\) hold, then

- \((\text{divide}++): \quad T_1 + T_2 \leq k U_1 + U_2 \quad \Rightarrow \quad T_1 \leq k U_1 \wedge T_2 \leq k U_2 \wedge (N(T_1) \wedge (S(T_2) \neq \emptyset) \Rightarrow N(U_1)) \wedge (N(T_2) \wedge (S(T_1) \neq \emptyset) \Rightarrow N(U_2))\)

- \((\text{divide} \cdot \cdot )): \quad T_1 \cdot T_2 \leq k U_1 & U_2 \quad \Rightarrow \quad T_1 \leq k U_1 \wedge T_2 \leq k U_2 \wedge (N(T_1) \wedge (S(T_2) \neq \emptyset) \Rightarrow N(U_1)) \wedge (N(T_2) \wedge (S(T_1) \neq \emptyset) \Rightarrow N(U_2))\)

- \((\text{divide} \cdot \cdot \cdot )): \quad T_1 \cdot T_2 \leq k U_1 \cdot U_2 \quad \Rightarrow \quad T_1 \leq k U_1 \wedge T_2 \leq k U_2 \wedge (N(T_1) \wedge (S(T_2) \neq \emptyset) \Rightarrow N(U_1)) \wedge (N(T_2) \wedge (S(T_1) \neq \emptyset) \Rightarrow N(U_2))\)}
Let $w \neq \epsilon \in [T_1]$, and consider any $w' \in [T_2]$; such a $w'$ exists by Lemma 3.6. Since $w \neq \epsilon \in [T_1 \cdot T_2]$ and $(w \cdot w') \neq \epsilon$, we have that $w \cdot w' \in [U_1 \cup S_2]$, hence, by Lemma 3.5(2), $(w \cdot w')[S(w)] \in [U_1]$. By $(S(T_1) \subseteq S(U_1) \land S(T_2) \subseteq S(U_2) \land S(U_1) \# S(U_2)$, we deduce that $S(S(w)) \subseteq S(U_1)$ and $(w \cdot w)[S(U_1)] \subseteq S(U_2)$, hence, by Lemma 3.5(1), we have that $(w \cdot w')[S(w)] = w$, hence $w \in [U_1]$. In the same way we prove that $T_2 \subseteq U_2$.

We now prove that $N(U_1) \land (S(T_2) \neq \emptyset) \Rightarrow N(U_1)$. Assume $w \in [T_1]$, and consider a non empty word $w' \in [T_2]$, that exists, by the hypothesis $S(T_2) \neq \emptyset$. By $T_1 \cdot T_2 \subseteq U_1 \cup S_2$, since $\epsilon \cdot w' \neq \emptyset$, $w' \in [U_1 \cup S_2]$, hence $(\epsilon \cdot w')[S(U_1)] \subseteq [U_1]$, hence $w \in [U_1]$. In the same way we prove that $N(U_2) \land (S(T_1) \neq \emptyset) \Rightarrow N(U_2)$.

Now we want to prove that:

$$T_1 \cdot T_2 \subseteq U_1 \cup U_2 \Rightarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2$$

Assume $w \in [T_1]$, and choose one $w' \in [T_2]$; such a $w'$ exists by Lemma 3.6. Since $w \cdot w' \in [T_1 \cdot T_2]$, we have that $w \cdot w' \in [U_1 \cup S_2]$, hence, by Lemma 3.5(2), $(w \cdot w')[S(U_1)] \in [U_1]$. By $S(T_1) \subseteq S(U_1) \land S(T_2) \subseteq S(U_2) \land S(U_1) \# S(U_2)$, we deduce that $S(w) \subseteq S(U_1)$ and $S(w) \# S(U_2)$, hence, by Lemma 3.5(1), we have that $(w \cdot w')[S(U_1)] = w$, hence $w \in [U_1]$. Finally we want to prove that:

$$T_1 \cdot T_2 \subseteq U_1 \cup U_2 \Leftarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2$$

Assume $w \in [T_1 \cdot T_2]$, hence, exist $w_1 \in [T_1]$ and $w_2 \in [T_2]$ such that $w = w_1 \cdot w_2$. If both $w_1$ and $w_2$ are different from $\epsilon$ we have that $w_1 \in [U_1]$ and $w_2 \in [U_2]$, hence $w = w_1 \cdot w_2 \in [U_1 \cup S_2]$. If $w_1 = \epsilon$, then $w_2 \neq \epsilon$, hence we have that $N(U_1) \land (S(T_2) \neq \emptyset)$, hence $N(U_1)$, hence $w_2 \in [U_1]$, hence we still have that $w \in [U_1 \cup U_2]$. The same reasoning applies when $w_2 = \epsilon$.

Now we want to prove that:

$$T_1 \cdot T_2 \subseteq U_1 \cup U_2 \Rightarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2$$

Let $w \in [T_1]$, and choose one $w' \in [T_2]$; such a $w'$ exists by Lemma 3.6. Since $w \cdot w' \in [T_1 \cdot T_2]$, we have that $w \cdot w' \in [U_1 \cup S_2]$, hence, by Lemma 3.5(2), $(w \cdot w')[S(U_1)] \in [U_1]$. By $S(T_1) \subseteq S(U_1) \land S(T_2) \subseteq S(U_2) \land S(U_1) \# S(U_2)$, we deduce that $S(w) \subseteq S(U_1)$ and $S(w) \# S(U_2)$, hence, by Lemma 3.5(1), we have that $(w \cdot w')[S(U_1)] = w$, hence $w \in [U_1]$. Finally we want to prove that:

$$T_1 \cdot T_2 \subseteq U_1 \cup U_2 \Leftarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2$$

Assume $w \in [T_1 \cdot T_2]$, hence, exist $w_1 \in [T_1]$ and $w_2 \in [T_2]$ such that $w = w_1 \cdot w_2$, hence $w_1 \in [U_1] \land w_2 \in [U_2]$, hence $w = w_1 \cdot w_2 \in [U_1 \cup U_2]$. By $S(T_1) \subseteq S(U_1) \land S(T_2) \subseteq S(U_2) \land S(U_1) \# S(U_2)$, we deduce that $S(w) \subseteq S(U_1)$ and $S(w) \# S(U_2)$, hence, by Lemma 3.5(1), we have that $(w \cdot w')[S(U_1)] = w$, hence $w \in [U_1]$. Finally we want to prove that:

$$T_1 + T_2 \subseteq U_1 \cup U_2 \Rightarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2 \land N(U_1) \land N(U_2)$$

Consider any $w \neq \epsilon \in [T_1]$; by $T_1 + T_2 \subseteq U_1 \cup U_2$ we have any $w \in [U_1 \cup U_2]$. By $S(w) \subseteq S(T_1) \subseteq S(U_1)$ and by Lemma 3.5(4) we have $w \in [U_1]$, hence $T_1 \subseteq U_1$. By Lemma 3.5(3), we also know that $w[S(U_2)] \in [U_2]$; since $w[S(U_2)] = w$, we deduce that $N(U_2)$. By considering any $w \neq \epsilon \in [T_2]$ we prove that $T_2 \subseteq U_2$ and that $N(U_2)$.

Now we want to prove that:

$$T_1 + T_2 \subseteq U_1 \cup U_2 \Leftarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2 \land N(U_1) \land N(U_2)$$

Assume $w \neq \epsilon \in [T_1 + T_2]$, hence $w \in [T_1]$ or $w \in [T_2]$. Without loss of generality assume $w \in [T_1]$. By hypothesis $w \in [U_1]$ and $\epsilon \in [U_2]$, hence $w \epsilon \in [U_1 \cup U_2]$. Now we want to prove:

$$T_1 + T_2 \subseteq U_1 \cup U_2 \Rightarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2 \land N(U_1) \land N(U_2)$$

Assume any $w \in [T_1]$; by $T_1 + T_2 \subseteq U_1 \cup U_2$ we have any $w \in [U_1 \cup U_2]$. By $S(w) \subseteq S(T_1) \subseteq S(U_1)$ and by Lemma 3.5(4) we have $w \in [U_1]$, hence $T_1 \subseteq U_1$. By Lemma 3.5(3), we also know that $w[S(U_2)] \in [U_2]$; since $w[S(U_2)] = \epsilon$, we deduce that $N(U_2)$.

Finally we want to prove that:

$$T_1 + T_2 \subseteq U_1 \cup U_2 \Leftarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2 \land N(U_1) \land N(U_2)$$

Assume $w \in [T_1 + T_2]$, hence $w \in [T_1]$ or $w \in [T_2]$. Without loss of generality assume $w \in [T_1]$. By hypothesis $w \in [U_1]$ and $\epsilon \in [U_2]$, hence $w \epsilon \in [U_1 \cup U_2]$. Now we want to prove:

$$T_1 + T_2 \subseteq U_1 \cup U_2 \Rightarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2 \land N(U_1) \land N(U_2)$$

Assume any $w \in [T_1]$; by $T_1 + T_2 \subseteq U_1 \cup U_2$ we have any $w \in [U_1 \cup U_2]$. By $S(w) \subseteq S(T_1) \subseteq S(U_1)$ and by Lemma 3.5(4) we have $w \in [U_1]$, hence $T_1 \subseteq U_1$. By Lemma 3.5(3), we also know that $w[S(U_2)] \in [U_2]$; since $w[S(U_2)] = w$, we deduce that $N(U_2)$.

Finally we want to prove that:

$$T_1 + T_2 \subseteq U_1 \cup U_2 \Leftarrow T_1 \subseteq U_1 \land T_2 \subseteq U_2 \land N(U_1) \land N(U_2)$$
Assume $w \in [T]$, by hypothesis $w \in [U_1]$ and $\epsilon \in [U_2]$, hence $w \notin \kappa \in [U_1 \cup U_2]$.

### 3.3.2 Linearity

Since every rule consumes a bit of input, by Theorem 3.4, it suffices to prove that, after a linear-time preprocessor, the applicable rule can be selected in constant time. Since we have a finite number of rules, it suffices to show that the applicability of each rule can be tested in constant time.  

Applicability conditions are a combination of the following components:

1. pattern matching, such as $T = T_1 \circ T_2$
2. boolean combination of nullability and symbol emptiness, such as $N(T_1) \land (S(T_2) \neq \emptyset) \Rightarrow N(U_1)$
3. symbol set inclusion, such as $S(T_1) \subseteq S(U_1)$

Component (1) is obviously in $O(1)$. A linear time bottom-up traversal can be used to decorate each node of $T$ and $U$ with attributes recording NULLability and emptiness of symbol set, hence solving component (2). Component (3) requires a bit more of work, which we describe next.

Given a type $T$, represented by its syntax tree, we introduce the following definitions:

- for any node $n$ of $T$, we use $S(n)$ to denote the symbols in the subtree of $T$ rooted in $n$; we may write $S_T(n)$ to emphasize that $n$ is a node of $T$;
- we write $Desc(n, m)$, when $n$ is a descendant of $m$, including the case $n = m$;
- for any node $n$ of $T$ with $S(n) \neq \emptyset$, and for any conflict-free type $U$, we use $cap_T(n)$ to denote the lowest node $m$ of $U$, if it exists, such that $S_U(m) \supseteq S_T(n)$ and we use $nodeOfSymbol_T(a)$ to denote the only node of $U$, if any, that corresponds to an atom $a$ in $U$.

**Lemma 3.8**

Given two types $T$ and $U$, with $S(T) \subseteq S(U)$, and where $U$ is conflict free:

1. for any node $n$ of $T$ whose children are $n_1$ and $n_2$, if $cap_T(n_1)$ and $cap_T(n_2)$ are well defined, then $cap_T(n) = LCA_T[cap_T(n_1), cap_T(n_2)]$, where $LCA_T[m_1, m_2]$ is the Lowest Common Ancestor of $m_1$ and $m_2$ in $U$;
2. for any node $n$ of $T$ with $S(n) \neq \emptyset$, $cap_T(n)$ is well defined.

**Proof.** (1) Any common ancestor $m$ of $cap_T(n_1)$ and $cap_T(n_2)$ enjoys $S(n_1) \cup S(n_2) \subseteq S(m)$, hence $S(n) \subseteq S(LCA_T[cap_T(n_1), cap_T(n_2)])$. Now consider any node $m$ in $U$ such that $S(n) \subseteq S(m)$, hence $S(n_1) \subseteq S(m)$ and $S(n_2) \subseteq S(m)$, hence $m$ is an ancestor of both $cap_T(n_1)$ and $cap_T(n_2)$.

(2) We reason by induction on the size of the subtree rooted in $n$. We have three cases: either $n$ corresponds to a symbol $a$, or it is a unary operator applied to a node $n'$ with $S(n') \neq \emptyset$, or it is a binary operator applied to $n_1$ and $n_2$. In the first case, since $U$ is conflict-free, $cap_T(n) = nodeOfSymbol_T(a)$. In the second case, $S(n) = S(n')$, hence $cap_T(n) = cap_T(n')$, which is well defined by induction. In the third case, we have to distinguish whether both $n_1$ and $n_2$ enjoy $S(n_1) \neq \emptyset$, or not. In the first case, we apply induction plus property (1). In the second case, let us assume that $S(n_1) = \emptyset$, hence $S(n_2) \neq \emptyset$ and $S(n) = S(n_2)$, hence $cap_T(n) = cap_T(n_2)$, which is well defined by induction.

Lemma 3.8 suggests a way to implement a constant-time test for $S_T(n) \subseteq S_U(m)$ after an $O(|T| + |U|)$ preprocessing. We first build, in time $O(|U|)$, the data structures needed to compute $LCA_U[m_1, m_2]$ and $nodeOfSymbol_U(a)$ in constant time [1]. We then decorate each node $n$ of $T$ such that $S(n) \neq \emptyset$ with a pointer to $cap_T(n)$ through a bottom-up visit of $T$, as follows: each leaf $a$ of $T$ is decorated with a pointer to $nodeOfSymbol_T(a)$, each node of $T$ with only one child $n_1$ with $S(n_1) \neq \emptyset$ is decorated with $cap_T(n_1)$, and each node of $T$ with two children $n_1$ and $n_2$ with $n_1 \neq \emptyset$ is decorated with $LCA_U[cap_T(n_1), cap_T(n_2)]$. Each step of the decoration phase takes constant time, hence $T$ can be decorated in time $O(|T|)$. Now, a test $S_T(n) \subseteq S_U(m)$ can be reduced to $Desc(cap_T(n), m)$, and $Desc(m', m)$ is equivalent to $LCA_U[m', m] = m$.

### 3.4 The Flat Version

The algorithm as presented embodies our main ideas, but it is too rigid, because it ignores basic commutativity and associativity properties of our type operators. For example, it would fail on all of the following examples:

\[ a + b \leq b + a \]
\[ a + b + a \leq b + a + c \]
\[ a \cdot (b - c) \leq (a \cdot b) - c \]
\[ ak(bkc) \leq (ckb)\&a \]
\[ a + (b + c) \leq b?(&(a + d) + c) \]

The first example shows that commutativity should be taken care of, and the second one elaborates a bit on this. The third example illustrates associativity. The fourth example shows that the simple approach of normalizing how operators are associated is not sufficient, because associativity and commutativity should be treated together. We solve this issue by adopting a flat version of all type operators, where every operator has an arbitrary number of arguments. This approach solves associativity, but leaves commutativity open; we may solve this by reordering all addends alphabetically, but that would require more than linear time. Moreover, the last example shows that flattening and then reordering is not enough: since the product of nullable factors is a supertype of union, one would need to consider some pairs of operators together.

The approach we implemented in the algorithm is a bit more elaborated. First of all, we generalize all binary operators to their n-ary version, and we preprocess the types, in linear time, to collapse all consecutive application of the same binary operator into one application of an n-ary operator. Second, when applying a divide/ndivide rule to a pair of types $T_1 \circ \ldots \circ T_n$ and $U_1 \circ \ldots \circ U_m$, we find, for each $T_i$, the minimum subterm $U_{\min}(i)$ of some $U_j$ that contains all of its symbols, so that we may recur on the pair $(T, U_{\min}(i))$. $U_{\min}(i)$ does not need to coincide with $U_i$, provided that some conditions (specified later) hold on its path to $U_j$, and this solves the issue presented in the fifth example.

To formalize this algorithm, we first define the notions of $+$-child and $+$-descendant. The intuition is that, when a subterm $U'$ of $U$ is a $+$-descendant of $U$, then $[U'] \subseteq [U]$.

**Definition 3.9**

PlusChild$(U, U')$ holds if one of the following conditions hold:

1. $U = U_1 + \ldots + U_i + \ldots + U_n$
2. $U = U_1 \otimes \ldots \otimes U_i \otimes \ldots \otimes U_n$, and $\forall j \in \{1, \ldots, i-1, i+1, \ldots, n\}, N(U_j)$.

The relation $\text{PlusDesc}(\_)$ is the reflexive and transitive closure of $\text{PlusChild}(\_)$.

The flat algorithm starts by first building the same data structures that we described for the binary algorithm, and then finding the node $m = \text{cap}_{T_i}(n)$, where $n$ is the root of $T_i$ and starting its work on the pair $n, m$. In this way, the algorithm always operates on a pair of nodes $n, m$ such that $m = \text{cap}_{T_i}(n)$, meaning that it never needs a focus rule. The algorithm, instead, repeatedly applies the following four rules which, guided by the topmost operators of $n$ and $m$, combine the tasks of divide and focus together.

With a slight abuse of notation, in the rules we write $\text{cap}_{T_i}(T_i)$ rather than $\text{cap}_{m}(n)$, by identifying each node with the type that is rooted in that node. We remove all $\epsilon$ types from the encoding, but we carry the corresponding information through an $N(T)$ mark at every node; in this way, the expression $\text{cap}_{T_i}(T_i)$ is always well-defined, since $S(T_i)$ is never empty, and $S(T_i) \subseteq S(U)$. The final case (SYMBOL) is not completely trivial, but is derived from the approach described in [6], and only requires a linear-time bottom-up visit of $T$.

The linearity of the algorithm follows from the fact that each rule has a cost that is proportional to the amount of symbols that it removes. We already described how to compute $\text{cap}_{T_i}(T_i)$ in constant time, and the test $\text{PlusDesc}(\_)$ can also be run in constant time, provided that one builds the relevant data structures during preprocessing, in linear time. The same is true for all the $N(T)$ tests.

$$T = T_1 + \ldots + T_n$$

$$U = U_1 \otimes \ldots \otimes U_m$$

$\forall i \in \text{1..n.} \text{ PlusDesc}(\text{cap}_{T_i}(T_i), U)$

$\forall i \in \text{1..n.} \text{ T}_i \leq_k \text{ cap}_{T_i}(T_i)$

$T \leq_k U$ \hspace{1cm} (⊕)

$$T = T_1 \otimes \ldots \otimes T_n$$

$$U = U_1 \& \ldots \& U_m$$

$\forall i \in \text{1..n.} \text{ PlusDesc}(\text{cap}_{T_i}(T_i), U_{\phi(i)})$

$\forall i \in \text{1..n.} \text{ N}(T_i) \Rightarrow N(U_{\phi(i)})$

$\forall j \in 1..m. \text{ j} \in \text{ϕ}(1), \ldots, \text{ϕ}(n) \text{ j} \& N(U_j)$

$\forall i \in 1..n. \text{ T}_i \leq_k \text{ cap}_{T_i}(T_i)$

$T \leq_k U$ \hspace{1cm} (⊗&)

$$T = T_1 \cdot \ldots \cdot T_n$$

$$U = U_1 \cdot \ldots \cdot U_m$$

$\forall i \in \text{1..n.} \text{ PlusDesc}(\text{cap}_{T_i}(T_i), U_{\phi(i)})$

$\forall i \in \text{1..n.} \text{ N}(T_i) \Rightarrow N(U_{\phi(i)})$

$\forall j \in 1..m. \text{ j} \in \text{ϕ}(1), \ldots, \text{ϕ}(n) \text{ j} \& N(U_j)$

$\forall i, j, i \leq j \Rightarrow \text{ϕ}(i) \leq \text{ϕ}(j)$

$\forall i \in 1..n. \text{ T}_i \leq_k \text{ cap}_{T_i}(T_i)$

$T \leq_k U$ \hspace{1cm} (−)

$$U = a [m..n] \hspace{1cm} S(T) \subseteq \{a\}$$

$\text{cardinality}_n(T) \subseteq \{m..n\}$

$T \leq_k U$ \hspace{1cm} (SYMBOL)

The rules differ because, when the subtype operator is $+$, we only have to check that every addend of the subtype corresponds to a $+$-descendant of the supertype. In the $(\otimes&)$ case, each factor of $T$ must correspond to a $+$-descendant of a factor of $U$, but, moreover, no factor in the subtype must be duplicated, and the missing factors in the supertype must be nullable. The condition $N(T_i) \Rightarrow N(U_{\phi(i)})$ that is found in the product rules corresponds to the $\epsilon$-inclusion condition that is found in the binary product rules. Finally, in the $-$ case, the factors of $T$ must also respect the order of the corresponding factors of $U$.

4. EXPERIMENTAL EVALUATION

In this paper we describe an improved algorithm for asymmetric inclusion between regular expression types. The new algorithm exploits a linear structural comparison technique whenever possible, and reverts to the quadratic approach of [6] when the linear comparison is no longer applicable.

As for any improvement, we must show that the “optimized” algorithm is more efficient than the original one and that its applicability conditions can be easily satisfied, so to justify its implementation.

To this purpose, starting from the observation in [8] that most human designed XML types are in conjunctive normal form, where each factor has the form $(a_1 + \ldots + a_k)$, $(a_1 + \ldots + a_k)?$, $(a_1 + \ldots + a_k)^+$, or $(a_1 + \ldots + a_k)^{\ast}$, we focus our experiments on CNF types and compare the performance of our algorithm with that of the quadratic algorithm on the four main kinds of factors.

To make these experiments more significant, the conflict-freedom restriction has been enforced on the supertype only, hence the subtype contains repeated labels.

4.1 Experimental Setup

Both the structural linear algorithm and the quadratic one [6] have been tested in Java 1.5, and all experiments were performed on a 2.16 Ghz Intel Core 2 Duo machine (3 GB main memory) running Mac OS X 10.5.7. To avoid issues related to independent system activities, we ran each experiment ten times, discarded both the highest (worst) and the lowest (best) processing times, and reported the average processing time of the remaining runs.

4.2 Experiments

As already stated, in our experiments we evaluate the performance of our algorithm on CNF types, with four main categories of factors: $(a_1 + \ldots + a_k)$, $(a_1 + \ldots + a_k)?$, $(a_1 + \ldots + a_k)^+$, and $(a_1 + \ldots + a_k)^{\ast}$. For the sake of completeness, we also evaluate our algorithm on a DNF types scenario, where types are in disjunctive normal form (e.g., the supertype and the supertype are a union of products).

In the supertype we impose the conflict freedom constraint, hence terminal symbols are unique and counting is applied only to terminal symbols, while these restrictions are relaxed in the subtype, which can be any legal type.

In our experiments we compared the performance of the structural algorithm with that of the plain mixed algorithm of [6]; in particular, we evaluate the scalability of the algorithms by increasing the number of addenda in each factor of both the supertype and the supertype from 10 to 100. To make the experiments even more realistic and test the flat algorithm, the supertype contains a 20% of randomly distributed labels. We only generated pairs of types which satisfy the subtype test, since this is the dominating situation when the algorithm is run by a compiler.

The results of our experiments on CNF types are shown...
in Figures 1, 2, 3, 4, and 5. As it can be seen, except for the case for $(a_1 + \ldots + a_k)^+$ (see Figure 4), the structural algorithm definitely outperforms the plain algorithm, that exposes its quadratic behaviour, while the structural algorithm is able to perform most of the work without resorting to the quadratic fallback. Hence, the structural algorithm is a definite improvement over the previous one.

In the case for $(a_1 + \ldots + a_k)^+$ the two algorithms show essentially the same performance. The problem is related to the encoding of $(a_1 + \ldots + a_k)^+$ factors as conflict-free types; this encoding is based on the use of a bang operator $!$, which is currently not covered by any applicability condition for the structural algorithm. Adding the relevant condition is a trivial task, which we plan to complete very shortly, in order to make this case uniform with the others.

In Figure 5 we illustrate a variation of the case for $(a_1 + \ldots + a_k)$, where ordered products have been replaced by commutative products. As for the base case, the performance is satisfactory and the speed gain is significant.

For the sake of completeness, in Figure 6 we report the results of an experiment with disjunctive normal form types: in both the subtype and the supertype each addendum is a product of factors, whose number ranges from 20 to 200. As
it can be seen, the structural algorithm significantly outperforms the plain one also in this case.

Figure 6: Structural vs quadratic algorithm: DNF types.

5. RELATED WORK

The inclusion of regular expressions with interleaving has been studied in many papers. In particular, in [10] the complexity of membership, inclusion, and inequality was studied for several classes of regular expressions with interleaving and intersection, and authors proved that inclusion, in the presence of interleaving, is EXPSPACE-complete.

Starting from the results of [10], 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.

In [9] we defined a quadratic algorithm for inclusion of conflict-free types, while in [6] we presented a new algorithm that can be applied when the subtype is not constrained (asymmetrical inclusion). This algorithm forms the basis for the present work.

Asymmetrical inclusion of XML types has been studied in [7] too. 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.

6. CONCLUSIONS

In [6] we proposed an algorithm to check subtyping among ERE types with the only restriction that the supertype must be conflict-free, as it commonly happens while type-checking XML programs. This algorithm has quadratic complexity, both in the best and worst cases, it strongly exploits the conflict-free restriction over the supertype, but does not exploit any structural similarities between the subtype and the supertype to further accelerate inclusion checking.

In this paper we have provided a more efficient algorithm, still dealing with the kind of mixed comparisons of [6], but which also exploits possible structural similarities between the types being compared. The new algorithm proceeds in a top-down fashion, and is based on a set of structural subtyping rules, that are applied whenever a structural similarity is detected; when these similarity conditions are not satisfied, the algorithm just resorts to the quadratic algorithm.

We have proved that the new algorithm is correct, and that it runs in linear time whenever the quadratic algorithm is not invoked. In order to ensure linearity we have defined a specific pre-processing technique, and a set of structural subtyping rules whose application can be decided in constant time, and that need no backtracking; in particular, to this end we have defined each subtyping rule so that its premises are equivalent to the conclusion. To verify the effectiveness of the new algorithm, we have performed several tests comparing an implementation of the new structural algorithm with respect to the one we proposed in [6]. As illustrated by our experiments, in most of the considered cases the new algorithm exhibits a linear behavior, while the other one is clearly quadratic.

7. REFERENCES