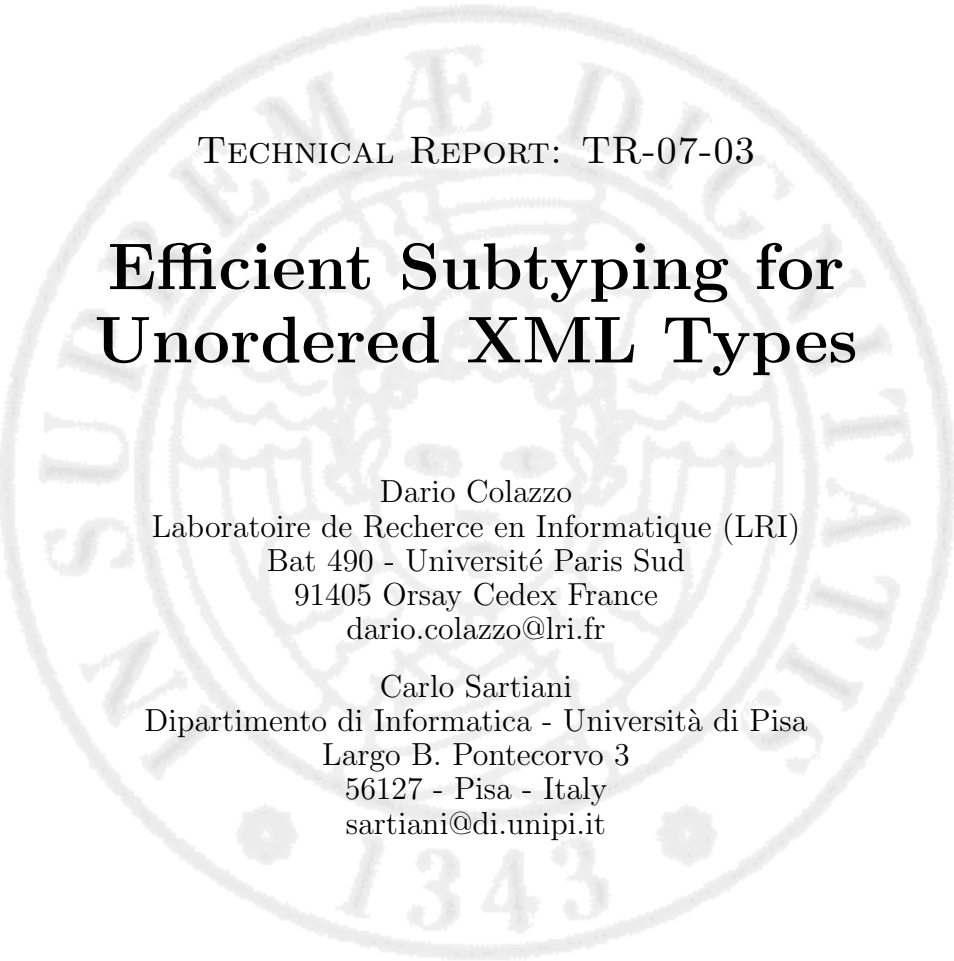
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-07-03

Efficient Subtyping for Unordered XML Types

Dario Colazzo
Laboratoire de Recherce en Informatique (LRI)
Bat 490 - Université Paris Sud
91405 Orsay Cedex France
dario.colazzo@lri.fr

Carlo Sartiani
Dipartimento di Informatica - Università di Pisa
Largo B. Pontecorvo 3
56127 - Pisa - Italy
sartiani@di.unipi.it

February 2, 2007

ADDRESS: Largo B. Pontecorvo 3, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Efficient Subtyping for Unordered XML Types

Dario Colazzo
Laboratoire de Recherce en Informatique (LRI)
Bat 490 - Université Paris Sud
91405 Orsay Cedex France
dario.colazzo@lri.fr

Carlo Sartiani
Dipartimento di Informatica - Università di Pisa
Largo B. Pontecorvo 3
56127 - Pisa - Italy
sartiani@di.unipi.it

February 2, 2007

Abstract

While XML is an ordered data format, many applications outside the document processing area just drop ordering and manipulate XML data as they were unordered. In these contexts, hence, XML is essentially used as a way for representing unordered, unranked trees. The wide use of unordered XML data should be coupled with a careful and detailed analysis of their theoretical properties.

One of the operations that is mostly affected by the presence of a global ordering relation is semantic subtype-checking, i.e., language inclusion. In an unordered context, inclusion has been proved to be inherently more complex than in the ordered case: in particular, subtype-checking for ordered single-type EDTDs is in PSPACE, while the same operation for single-type EDTDs with unordered types is in EXPSPACE (the same complexity result holds for unordered DTDs). Comparing two unordered XML types for inclusion, hence, is very expensive; as a consequence, it becomes very important to identify restrictions defining type classes for which inclusion is tractable or, at least, less complex.

This paper identifies two large subclasses of unordered XML types for which inclusion can be computed by an EXPTIME and a PTIME algorithm, respectively. These classes are defined by restrictions on the use of element, repetition, and union types, and comprise many DTDs and XML Schemas used in practice.

1 Introduction

The last few years have seen an increasing diffusion of the *eXtensible Markup Language* (XML). Given its ability to represent both structured and unstructured data, XML is currently being used in several contexts: data integration

and data exchange systems, where XML is exploited for reconciling data coming from multiple and heterogeneous (both in the structure and in the data model) data sources; document processing applications, where XML flexibility is essential for manipulating complex documents; and, more generally, any data interchange process between heterogeneous applications, e.g., a spreadsheet and a DBMS. XML is an inherently ordered data representation format, as it imposes a global ordering relation among elements; this reflects on the XML data model and schema language (XML Schema [18, 4]), which are ordered, as well as on query and transformation languages for XML, like XSLT [7] and XQuery [5], which strongly take ordering into account. As a consequence, many theoretical studies about XML focus on the ordered nature of XML.

While XML is an ordered data format, many applications outside the document processing area just drop ordering and manipulate XML data as they were unordered. In these contexts, hence, XML is essentially used as a way for representing unordered, unranked trees: for instance, in data integration or data exchange scenarios, where multiple and autonomous data sources are combined together, no global order relation can be imposed on the combined (virtual) data, as local order relations cannot be reconciled. Hence, it is worth to deeply analyze the theoretical properties of unordered XML data.

One of the operations that is mostly affected by the presence of a global ordering relation is semantic subtype-checking, i.e., language inclusion. While ordered and unordered semantic subtyping share the same definition, their formal properties are quite different, in particular for what concerns computational complexity. In the ordered context, subtype-checking among ordered XML types has been widely studied: for *single-type extended DTDs* (an abstraction capturing XML Schemas), subtype-checking has been proved to be in PSPACE [17], and several "optimized" algorithms, based on special-purpose heuristics, have been designed in the context of the XDuce [16] and CDuce [2] projects, as well as in some XQuery implementations [12].

In an unordered context, instead, inclusion has been proved to be inherently more complex than in the ordered case (see [10] and [14]): in particular, subtype-checking for single-type EDTDs with unordered types is in EXPSPACE (the same complexity result holds for unordered DTDs). Comparing two unordered XML types for inclusion, hence, is very expensive; as a consequence, it becomes very important to identify restrictions defining type classes for which inclusion is tractable or, at least, less complex.

Our Contribution This paper identifies two large subclasses of unordered XML types for which inclusion can be decided in EXPTIME and PTIME, respectively. We first impose a restriction on the use of tree and repetition types inside type regular expressions, so to define a class of types with (at most) EXPTIME inclusion; this restriction is met by many practical DTDs and XML Schemas, hence it does not represent a severe limitation for the applicability of the whole approach. For types in this class, subtyping can be captured, in a correct and complete way, by a symbolic, simulation-based relation, which is then used for deriving an *efficient* algorithm; this algorithm, inspired by the type projection algorithm shown in [9], has a worst case EXPTIME complexity. It is worth to notice that the same result holds when this restriction is met by the right hand-side of the inclusion only; in other words, when comparing T and

U for inclusion, T can be an arbitrary type.

By imposing a further restriction on the grammar of types (both on the right and the left hand-side of the comparison), we identify a second class of types, for which the proposed approach is in PTIME. This class is described by a property that requires union types to be guarded by repetition types or by element types, and that is satisfied by a large fragment of DTDs used in practice (see [3]).

We chose a simulation-based approach for a twofold reason. First of all, we found simulation a convenient way for discovering critical aspects of subtyping, in particular for what concerns backtracking issues. Furthermore, the axiomatization of subtyping allows for an easy implementation of subtyping, which is usually quite hard to implement.

Paper Outline The paper is organized as follows. Section 2 describes the reference type language and its semantics, and recalls the notion of *semantic* subtyping. Section 3, then, introduces *type compactness*, a non-ambiguity property enjoyed by many practical DTDs and XML Schemas; this property allows for the definition of an EXPTIME correct and complete algorithm. This algorithm is obtained by an axiomatization of subtyping, hence Section 4 illustrates how semantic subtyping can be encoded into symbolic subtyping, and shows that this encoding is correct and complete over compact types. Section 5, next, presents our subtype-checking algorithm as well as its main properties. Section 6 describes *element/star-guarded types* (ESG types) and shows that inclusion for unordered ESG types is in PTIME. In Sections 7 and 8, finally, we review some related works and draw our conclusions.

2 Type Language

We represent an XML document as an *unranked*, *unordered*, node-labeled tree, as shown by the following grammar.

$$f \quad ::= \quad () \quad | \quad b \quad | \quad l[f] \quad | \quad f, \ldots, f$$

f is a forest that may comprise the empty sequence ($()$), base values (b), element nodes ($l[f]$), as well as the concatenation of other forests $(f, \ldots, f)$; hence, concatenation $(f, \ldots, f)$ is commutative, associative, and has $()$ as neutral element.

Our type language, based on XDuce [16] and XQuery [11] type languages, is shown in Figure 1, where $()$ is the type for the empty sequence value, B denotes the type for base values (without loss of generality, we only consider string base values), types T, U and $T \mid U$ are, respectively, product and union types, $T*$ is the type for 0-or-many repetitions, $T+$ is the type for 1-or-many repetitions, and, finally, $T?$ is the optional type. As we assume XML forests to be unordered, our XML types are unordered too. Our type language, inspired by data-centric applications, does not include recursive types (only repetition is permitted). This is a significant difference from XDuce and XQuery type languages, where types are ordered and recursion is allowed; however, our results can be easily extended to the case of recursive types, where recursion is guarded by element types and each type equation contains at most one recursion variable. Furthermore, early results based on sheaves automata [13] and NFA(&) automata [14]

$$
\begin{array}{llllllll}
\textbf{Types} & T & ::= & () & | & \mathsf{B} & | & l[T] \quad | \\
 & & & T,T & | & T \mid T & | & T* \quad | \\
 & & & T+ & | & T? & & \\
\textbf{Base Type} & \mathsf{B} & ::= & \texttt{String} & & & & \\
\end{array}
$$

Figure 1: Type language

seem suggesting that vertical recursion is not a key issue in determining the complexity of inclusion for type languages close to ours.

For the sake of simplicity, in the following we will consider B as equivalent to $\_\mathsf{B}[()]$, i.e., an element type with a special purpose label $\_\mathsf{B}$ containing the empty sequence as its only child. Therefore, any property or result related to element types will be valid also for B.

As types are unordered, in the following we will consider a product type $T_1, \ldots, T_n$ as identical to all its possible permutations $T_{\pi(1)}, \ldots, T_{\pi(n)}$, where π is a permutation over $\{1, \ldots n\}$. Moreover, as our types actually are XDuce unordered types, we also have that $T, ()$ is identical to T, and that $(T, T'), T''$ is identical to $T, (T', T'')$. This conforms to the corresponding laws over the data model.

The semantics of types is standard: as usual, $[\![\_]\!]$ is the minimal function from types to sets of forests that satisfies the following monotone equations:

$$
\begin{array}{lcl}
[\![()]\!] & \triangleq & \{()\} \\
[\![\mathsf{B}]\!] & \triangleq & \{b \mid b \textit{ is a base value}\} \\
[\![l[T]]\!] & \triangleq & \{l[f] \mid f \in [\![T]\!]\} \\
[\![T_1 \mid T_2]\!] & \triangleq & [\![T_1]\!] \cup [\![T_2]\!] \\
[\![T_1, T_2]\!] & \triangleq & \{f_1, f_2 \mid f_i \in [\![T_i]\!]\} \\
[\![T*]\!] & \triangleq & [\![T]\!]^* \\
[\![T+]\!] & \triangleq & [\![T*, T]\!] \\
[\![T?]\!] & \triangleq & [\![T \mid ()]\!]
\end{array}
$$

In the following we will write $f : T$ in place of $f \in [\![T]\!]$, and we will write f_T to indicate a forest in $[\![T]\!]$.

From the definition of type semantics, it follows an interesting property of *unordered*, non-recursive regular expression types, that will be used later on in the paper.

Lemma 2.1 *Given two types T and U:*

$$[\![T*, U*]\!] = [\![(T \mid U)*]\!]$$

This property is a key issue in our work as it permits to restrict the subtype-check to types with only one $*$ type at the top level, so to trim the possible cases in the symbolic subtype comparison.

Subtyping is defined as language inclusion, as shown below.

Definition 2.2 (Subtyping) *Given two types T_1 and T_2, we say that T_1 is a subtype of T_2 iff the semantics of T_1 is included in the semantics of T_2:*

$$T_1 < T_2 \quad \Longleftrightarrow \quad [\![T_1]\!] \subseteq [\![T_2]\!]$$

3 Type Compactness

Our study is based on a non-ambiguity property for XML unordered types, that allows us to identify a large class of types for which subtype-checking can be efficiently enforced. Before introducing type compactness, we need three preliminary definitions, describing *type contexts*, *top level type terms*, and *top level tags*, respectively.

Definition 3.1 (Type contexts) *We denote as $C[\cdot]$ a one-hole type context defined by following grammar:*

$$C[\cdot] = [\cdot] \;\mid\; l[C[\cdot]] \;\mid\; T, C[\cdot] \;\mid\; T \;\mid C[\cdot] \;\mid\; C[\cdot]*$$

Moreover, we say that $C[\cdot]$ is a light *type context, and denote it as $C_{\mathcal{L}}[\cdot]$, if it can be defined by the above grammar without using the case $l[C'[\cdot]]$.*

Light contexts are useful to define the set of *top level terms* of a type, i.e., subterms not occurring inside the content of an element type

Definition 3.2 (Top level type terms) *Given a type T, $Top(T)$ is the set of terms occurring at the top level of T, and it is defined as follows.*

$$Top(T) \quad = \quad \{U \mid \exists C_{\mathcal{L}}[\cdot].\ T = C_{\mathcal{L}}[U]\}$$

Definition 3.3 (Top level tags) *Given a forest f, $topLT(f)$ is the set of tags occurring at the top level of f, and it is defined as follows.*

$$topLT(f) = \{l \mid f = f', l[f''], f'''\}$$

$topLT()$ can be extended in a natural way to types, as shown below.

Definition 3.4 *Given a type T, $topLT(T)$ is the set of tags occurring at the top level of T, and it is defined as follows.*

$$topLT(T) = \{l \mid l[U] \in Top(T)\}$$

Finally, in the following we will use symbols $\mathcal{A}, \mathcal{B}, \mathcal{C}$ to denote sequence of tree types.

Given these preliminary notions, we can state our definition of *compact types*.

Definition 3.5 (Pre-compact types) *A type T is* pre-compact *if for each type S such that $S = T$ or $T = C[l[S]]$:*

- *(i) for each $m \in topLT(S)$, there is only one occurrence of m at the top level of S; formally if $S = C_{\mathcal{L}}[m[U]]$, then $m \notin topLT(C_{\mathcal{L}}[()])$;*
- *(ii) for each $U+ \in Top(S)$, it is $U = \bigcup_1^n \mathcal{A}_i$;*

- *(iii) for each* $U* \in Top(S)$*, it is* $U = \bigcup_1^n \mathcal{A}_i$.

Definition 3.6 (Compact types) *A type* T *is* compact *if there exists a pre-compact type* T' *such that* T *can be obtained from* T' *by just replacing sub-terms* $U+$ *and* $U?$ *by, respectively,* $U*, U$ *and* $U \mid ()$*. Moreover, we require that, if* $T = C[S|()]$ *with* S *not a union type, then* $S = S', l[S'']$ *and* $l \notin topLT(S')$.

Type compactness comprises three syntactical restrictions, addressing several ambiguity issues. The essence of these restrictions is to prevent backtracking phenomena while comparing two given types for inclusion. It is worth to notice that similar backtracking issues arise in automaton-based approaches ([13]), and are someway independent from the simulation-based approach we use here. We do not claim here that these restrictions identify the maximal type class for which our approach is correct and complete; we are aware that these restrictions can be someway relaxed without altering the properties of our approach. However, relaxed restrictions are so complex that we prefer to use a simpler, even if coarser, property.

It is worth to observe that our algorithm is still *sound* on non-compact types, i.e., if the subtype-check succeeds, then the subtyping relation among the types being compared actually holds. As we will show later, there are cases where the compactness property is systematically violated and for which our algorithm is correct.

As an example of type compactness, consider the following (pre-compact) type: $(a[\mathsf{B}]|(b[\mathsf{B}], c[\mathsf{B}]))+, d[\mathsf{B}]$. This type meets the first restriction, as no tag is used twice inside the same $Top(S)$ set. Furthermore, restriction (ii) is satisfied as well, since the argument of the + constructor is a union of sequence types.

Type compactness is a syntactical property that can be enforced by a simple visit of the type in polynomial time. In [1] a similar restriction, called conflict freedom, has been described and in [6] it has been shown that most XML Schemas and DTDs used in practice meet that requirement. Type compactness is actually a refinement of conflict freedom, as restriction (ii) serves the purpose of closing the class of pre-compacts types under the equivalence $T+ = T*, T$, which is crucial for the completeness of our approach, and restriction (iii) allows one to check $T*, A < U*, B$ without backtracking. The restriction on optional types meets the purpose of disambiguating comparisons of the kind $T < U \mid V$, hence avoiding backtracking phenomena; indeed, no simulation approach can be complete on union types if one cannot prove that either $T < U$ or $T < V$. This restriction is actually very light as optional types are usually exploited to guard element types.

Given these restrictions, in the following we will focus on a smaller type language, where + and ? type constructors are canonically rewritten.

As we will see later, in a subtype-check $T < U$ we only require that the right-hand side type (i.e., U) is compact, while no restriction is imposed on the left-hand side one (i.e., T). This is a very important property of our approach. Indeed, in many cases a type T is automatically inferred [11, 8] from a query or a transformation function, and it is compared against a predefined, human-designed schema U; given the inner complexity of type inference for XML queries and transformations, it is very unlike that T would meet restrictions like conflict freedom or type compactness, while the human-designed schema U can be tuned to satisfy those requirements.

Table 4.1. *norm() function.*

$norm(())$	$\triangleq$	$()$
$norm(\mathsf{B})$	$\triangleq$	B
$norm(l\,[T])$	$\triangleq$	$l\,[norm(T)]$
$norm(T \mid U)$	$\triangleq$	$norm(T) \mid norm(U)$
$norm(T'*, U'*, U)$	$\triangleq$	$norm((T' \mid U')*, U)$
$norm(T, U)$	$\triangleq$	$\begin{cases} \bigcup_1^n norm(A_i, U) & \text{if } norm(T) = (A_1 \mid ..... \mid A_n) \\ \bigcup_1^n norm(T, A_i) & \text{if } norm(U) = (A_1 \mid ..... \mid A_n) \\ norm(T), norm(U) & \text{otherwise} \end{cases}$
$norm(T*)$	$\triangleq$	$norm(T)*$

4 Symbolic Subtyping

We claimed in the Introduction that semantic subtyping on unordered compact types can be decided in EXPTIME. To prove this claim we introduce the notion of *symbolic* subtyping, and show the equivalence between symbolic and semantic subtyping over unordered compact types. Through this equivalence, we can use the EXPTIME symbolic subtype-checking of Section 5 for semantic subtyping, hence proving the EXPTIME upper bound. We stress that, while our approach is in EXPTIME, we do not know if the inclusion problem for compact types is actually in EXPTIME; we just provide here an EXPTIME algorithm for a problem that may have lower complexity.

4.1 Preliminary Definitions

Symbolic subtyping is defined among types in *disjunctive normal form*, i.e., types where products are distributed across unions. A type T can be normalized by applying the normalization function $norm(T)$, defined as shown in Table 4.1.

$norm()$ works by transforming types, while preserving their semantics (Lemma 4.1), so that the transformed types can be easily compared by the symbolic relation (and by the corresponding algorithm). For instance, $norm(T'*, U'*, U)$ transforms a product of repetition types, which is hard to formalize in the simulation rules, into a *-guarded union, for which much easier simulation rules exist (correctness of this transformation is entailed by Lemma 2.1).

To eliminate some ambiguity, the rules of the $norm()$ function must be applied in the order in which they are defined. $norm()$ can be applied to any type to obtain a corresponding normalized type, and its relevance resides in the proof of equivalence between symbolic and semantic subtyping, as it will be clear in the rest of the paper.

$norm()$ is essentially equivalent to the transformation of logic formulas in DNF (see rules for $norm(T, U)$), hence its computational complexity is in EXPTIME. Despite this upper bound, for a vast class of types $norm()$ can be computed in PTIME, as it will be shown in Section 6.

The following lemma proves that $norm()$ preserves the semantics of types.

Lemma 4.1 *For each type T, $[\![T]\!] = [\![norm(T)]\!]$.*

Proof. The proof easily follows by induction on $size(T)$, defined as follows

$$\begin{array}{lcl} size(()) & = & 0 \\ size(\mathsf{B}) & = & 0 \\ size(T'*) & = & 1 + size(T') \\ size(l[T']) & = & 1 + size(T') \\ size(T', U') & = & 1 + size(T') + size(U') \\ size(T' \mid U') & = & 1 + size(T') + size(U') \end{array}$$

■

Definition 4.2 (Prime types) *A type T is* prime *if and only if $T = norm(T)$ and $T \neq A \mid B$.*

The following lemma proves an important property concerning *maximal prime* subterms of $norm(T)$ with T compact. First of all we say that U is a *maximal prime* subterm of $norm(T)$ if either $norm(T) = C[l[U]]$ or $norm(T) = C[U \mid U']$, where U is prime.

Lemma 4.3 *If U is a* maximal prime *subterm of $norm(T)$, with T compact, then U meets the following property: U is either $()$ or a type $(\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n)*$, or $(\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n)*, \mathcal{A}$ or simply a type $\mathcal{A}$, with $\mathcal{A}$ having unique top-level tags and such that*

- *either $\mathcal{A} = \mathcal{B}'_1, \ldots, \mathcal{B}'_k$, or $\mathcal{A} = \mathcal{B}'_1, \ldots, \mathcal{B}'_k, \mathcal{D}$, or $\mathcal{A} = \mathcal{D}$*

  with each $\mathcal{B}_i$ and $\mathcal{A}$ having unique top level tags, and
- *$n \geq 1$ and $topLT(\mathcal{B}_i) \cap topLT(\mathcal{B}_j) = \emptyset$, for $i \neq j$.*
- *$\forall \mathcal{B}'_s . \exists \mathcal{B}_j .\ \mathcal{B}'_s = \mathcal{B}_j$*
- *$topLT(\mathcal{D}) \cap topLT(\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n) = \emptyset$*

Proof. The proofs follows by using an alternative and more verbose definition of normalization for the case of product type, $norm(T', U') \triangleq$

$$\begin{cases} \bigcup_1^n (norm(\mathcal{C}_i), (norm(T'))*) & \text{if } T' = (\mathcal{C}_1 \mid ..... \mid \mathcal{C}_n) \\ & \quad U' = (T'*) \\ \bigcup_1^n norm(A_i, U') & \text{if } norm(T') = (A_1 \mid ..... \mid A_n) \ \wedge \\ & \quad topLT(T') \cap topLT(U') = \emptyset \\ \bigcup_1^n norm(T', A_i) & \text{if } norm(U') = (A_1 \mid ..... \mid A_n) \wedge \\ & \quad topLT(T') \cap topLT(U') = \emptyset \\ norm(T'), norm(U') & \text{otherwise} \end{cases}$$

This redefinition is equivalent to the previous one over compact types.

The proof then continues by induction on $size(T)$, without presenting particular technical issues. ■

The lemma that follows will play a central role in the following, and explains what are the effects of compactness from the point of view of type semantics.

Lemma 4.4 *If U is a* maximal prime *subterm of $norm(T)$ with T compact and $U = S*, \mathcal{A}$ with $\mathcal{A}$ and $S = (\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n)$ meeting properties stated in the previous lemma, then*

- *$topLT(\mathcal{A}) \subseteq topLT(\mathcal{S}) \Rightarrow (f : U \Rightarrow f : S*)$*
- *otherwise $\mathcal{A} = \mathcal{B}, \mathcal{D}$ and $topLT(\mathcal{D}) \cap topLT(S) = \emptyset$ and $topLT(\mathcal{B}) \subseteq topLT(S) \wedge \ (f : U \Rightarrow f = f_{S*}, f_{\mathcal{D}})$*

Proof. The proof follows by properties stated in Lemma 4.3. ∎

The following definitions and lemmas characterize the properties of normalized types; in particular, the Upward Closure Lemma expresses the fact that in prime types, obtained by normalization, there are not exclusive labels. This property is crucial to allow for a top-down structural analysis to check subtyping of union types.

Lemma 4.5 (Upward closure) *If T is prime then $\forall\ f_1, f_2 \in [\![T]\!]$. $\exists f \in [\![T]\!]$. $topLT(f_i) \subseteq topLT(f)$*

Proof. We reason by case analysis on the form of T. As T is prime, $T = norm(T)$ and $T \neq U \mid V$. Hence, T is a product of tree types and/or repetition types. The main case is $T = (V)*$. Here the main observation that directly leads to the proof is that $\forall f_1,\ f_2 \in [\![T]\!].(f_1, f_2) \in [\![T]\!]$.

∎

The following lemma will serve to prove Lemma 4.7.

Lemma 4.6 *If T is compact and $T = \bigcup_1^n \mathcal{A}_i$ then*

$$norm(T+) = norm(T, (T*)) = \bigcup_1^n (norm(\mathcal{A}_i), (norm(T))*)$$

and for $i \neq j$:

- *$topLT(norm(\mathcal{A}_i)) \cap topLT(norm(\mathcal{A}_j)) = \emptyset$;*
- *$topLT(norm(\mathcal{A}_i), (norm(T))*)) \ = \ topLT(norm(\mathcal{A}_j), (norm(T))*)$.*

Proof. The proof easily follows once we observe that over product types $\mathcal{B}$ normalization preserves top level tags and the product structure. ∎

Restriction (i) of type pre-compactness requires that a pre-compact type T has no repeated tags at the top level of any type subterm S such that $S = T$ or $T = C[l[S]]$. This tag uniqueness property is of course lost after normalization, which is always applied before subtype-checking. However, uniqueness of tag occurrences in T implies some properties about $norm(T)$ that are able to prevent backtracking during the subtype comparison, as well as to correctly decompose union types (i.e., $T < U \mid V$ if $T < U$ or $T < V$).

For instance, we have $norm((l[\mathsf{B}]* \mid m[\mathsf{B}]*), b[\mathsf{B}]) = l[\mathsf{B}]*, b[\mathsf{B}] \mid m[\mathsf{B}]*, b[\mathsf{B}]$ where the tag b repeats twice. However here we are able to distinguish the two types in union by the fact that, if not $f_{l[\mathsf{B}]*, b[\mathsf{B}]} : m[\mathsf{B}]*, b[\mathsf{B}]$, then there is a

top level tag of $f_{l[\mathsf{B}]*,b[\mathsf{B}]}$ which is not a top level tag of $m[\mathsf{B}]*, b[\mathsf{B}]$. As we will show later, if during the subtype-check we have $S < l[\mathsf{B}]*, b[\mathsf{B}] \mid m[\mathsf{B}]*, b[\mathsf{B}]$, with S prime, thanks to this property we will know that either $S < l[\mathsf{B}]*, b[\mathsf{B}]$ or $S < m[\mathsf{B}]*, b[\mathsf{B}]$.

The same holds for $norm((l[\mathsf{B}] \mid m[\mathsf{B}])+, a[\mathsf{B}]) = norm((l[\mathsf{B}] \mid m[\mathsf{B}])*, (l[\mathsf{B}] \mid m[\mathsf{B}]), a[\mathsf{B}])$ which is $(l[\mathsf{B}] \mid m[\mathsf{B}])*, l[\mathsf{B}], a[\mathsf{B}] \mid (l[\mathsf{B}] \mid m[\mathsf{B}])*, m[\mathsf{B}], a[\mathsf{B}]$. So here the two types in union have the same set of top level tags, but we have the part out of the star which makes the distinction between the two types in union after normalization, even if there is $a[\mathsf{B}]$ that repeats.

Finally consider $norm(l[\mathsf{B}], (m[\mathsf{B}], n[\mathsf{B}])?) = l[\mathsf{B}] \mid l[\mathsf{B}], m[\mathsf{B}], n[\mathsf{B}])$. Here we have $l[\mathsf{B}]$ that repeats, but $m[\mathsf{B}], n[\mathsf{B}]$ in the second part that makes the distinction.

These three examples are generalized in the following lemma, which, we recall, will be crucial to prove that $T < U \mid V$ implies $T < U$ or $T < V$ (Lemma 4.9).

Lemma 4.7 *Let T be compact, if $norm(T) = C[S \mid V]$, with S and V prime, then one of the following three properties holds:*

1. *$topLT(S) \bigtriangleup topLT(V) \neq \emptyset$ implies that if $f : S$ and not $f : V$, then $\exists\, l \in topLT(f)$ such that $l \notin topLT(V)$, and vice versa, where $\bigtriangleup$ is the symmetrical difference operator;*
2. *$topLT(S) \subset topLT(V)$ implies $V = V', l[V'']$ and $l \notin topLT(S) \cup topLT(V')$;*
3. *$topLT(S) = topLT(V)$ implies $S = \mathcal{A}, (S')*$ and $V = \mathcal{B}, (V')*$ with $topLT(\mathcal{A}) \bigtriangleup topLT(\mathcal{B}) \neq \emptyset$.*

Proof. To easy the proof we reformulate the statement in the following equivalent way, where in the fourth point we add some properties implied by Lemma 4.3:

1. $topLT(S) \bigtriangleup topLT(V) \neq \emptyset$ and, if $f : S$ and not $f : V$, then $\exists\, l \in topLT(f)$ such that $l \notin topLT(V)$, and vice versa, where $\bigtriangleup$ is the symmetrical difference operator;
2. $topLT(S) \subset topLT(V)$ and $V = V', l[V'']$ and $l \notin topLT(S) \cup topLT(V')$;
3. $topLT(V) \subset topLT(S)$ and $S = S', l[S'']$ and $l \notin topLT(V) \cup topLT(S')$;
4. $topLT(S) = topLT(V)$ and $S = \mathcal{A}, \mathcal{D}, (S')*$ and $V = \mathcal{B}, \mathcal{D}, (V')*$ with
   - $topLT(\mathcal{A}) \bigtriangleup topLT(\mathcal{B}) \neq \emptyset$
   - $\mathcal{A} = \mathcal{A}_1, \ldots, \mathcal{A}_n$ and $\mathcal{B} = \mathcal{B}_1, \ldots, \mathcal{B}_m$
   - $S' = (\bigcup_1^n A_i) \mid (\bigcup_1^m B_i) \mid S'' = V'$
   - $topLT(\mathcal{D}) \cap topLT(S') = \emptyset$ and $(topLT(\mathcal{A}) \cup topLT(\mathcal{B})) \cap topLT(S'') = \emptyset$

Again, we use the alternative definition $norm(T)$ provided in the proof of Lemma 4.3.

We then proceed by case distinction on the form of T and by induction on $size(T)$. Base cases (i.e., $T = ()$ and $T = B$), for which $size(T) = 0$, are trivial.

Concerning the inductive cases, the easiest one is that for $T = T'*$. In this case it is sufficient to observe that $norm(T'*) = norm(T')*$, and that all unions are in $norm(T')$, where the 4 properties in the thesis hold by the inductive hypothesis. The case $T = l[T']$ is similar.

Consider now $T = T_1 \mid T_2$. We recall that $size(T_i) < size(T)$, $norm(T) = norm(T_1) \mid norm(T_2)$, and that, by induction, the thesis holds for unions in $norm(T_i)$. So to prove the thesis it remains to prove that it holds for unions $V_1 \mid S_1$, with $norm(T_1) = V_1 \mid \ldots$ and $norm(T_2) = S_1 \mid \ldots$. But since T_1 and T_2 do not have common top-level tags, the same holds for V_1 and S_1, so that $topLT(V_1) \bigwedge topLT(V_2) \neq \emptyset$, meaning that for these unions $V_1 \mid S_1$ property 1 holds.

The case $T = T', U'$ is the most difficult one. We go by case distinction on the kind of product, as indicated in the previous re-definition of normalization.

The first case is immediate by Lemma 4.6. The second and third cases are similar, so we only treat the second.

We have that T' and U' do not have top-level common tags. Moreover:

$$norm(T', U') = \bigcup_{1}^{n} norm(A_i, U')$$

with $norm(T') = (A_1 \mid ..... \mid A_n)$, where each A_i prime.

Also assume that

$$norm(U') = (B_1 \mid ..... \mid B_m)$$

with each B_i prime. Note that types A_i's have no common top-level tags with respect to types B_h's.

Since, by induction, the thesis holds for unions in each $norm(A_i, U')$, we have to prove that the thesis holds for $V_1 \mid S_1$, with $norm(A_i, U') = V_1 \mid \ldots$ and $norm(A_j, U') = S_1 \mid \ldots$ with $i \neq j$. It is not difficult to prove that $V_1 = norm(A_i, B_h)$ and $S_1 = norm(A_j, B_k)$.

Now we have to observe and keep in mind this fact: since types $A_i,\ B_h,\ A_j,\ B_k$ are prime, normalization of $norm(A_i, B_h)$ and $norm(A_j, B_k)$ only requires a kind of top level reorganization of the types: tipically merge of * types. This operation does not alter properties stated in this lemma's thesis, that are 1, 2, 3, and 4.

We know that the lemma holds for pairs $A_i,\ A_j$ and $B_h,\ B_k$; this means that for $A_i,\ A_j$ we have one of the property 1, 2, 3, and 4 in the thesis statement hold; and similarly for $B_h,\ B_k$. But thanks to the previous fact, we can provide the following table showing that the lemma holds for all possible combination of the 4 cases of the lemma statement.

In the first column of the table we have the possible 4 cases for $A_i,\ A_j$ and on the first line the 4 possible cases for $B_h,\ B_k$. The body of the table contains the corresponding case (1, 2, 3, or 4) of the lemma statement that holds for the pair $norm(A_i, B_h)$ and $norm(A_j, B_k)$:

Table 4.2. *Symbolic subtyping.*

1)	()	$<:$	()	
2)	()	$<:$	$T*$	
3)	B	$<:$	B	
4)	$l[T]$	$<:$	$l[U]$	if $T <: U$
5)	$T*, \mathcal{A}$	$<:$	$U*, \mathcal{B}$	if $\mathcal{A} <: U*, \mathcal{B}$ and $T* <: U*$
6)	T_1, T_2	$<:$	U_1, U_2	if $T_1 <: U_1$ and $T_2 <: U_2$
7)	T_1	$<:$	$U_2 \mid U_3$	if $T_1 <: U_2$ or $T_1 <: U_3$ with $T_1 \neq V_1 \mid V_2$
8)	$T_1 \mid T_2$	$<:$	U	if $T_1 <: U$ and $T_2 <: U$
9)	T	$<:$	$U*$	if $T <: U$
10)	$T*$	$<:$	$U*$	if $T <: U*$
11)	T_1, T_2	$<:$	$U*$	if $T_1 <: U*$ and $T_2 <: U*$

	1	2	3	4
1	1	1	1	1
2	1	2	1	2
3	1	1	3	3
4	1	2	3	4

This ends the proof for the second case of re-definition provide in Lemma 4.3; the third case is similar to that, so it remains the fourth case. Here things are easier: we have $norm(T', U') = norm(T'), norm(U')$, which is not a union type. So all remaining unions types are inside element typed of $norm(T')$ and $norm(U')$, for which the four properties of the lemma statement hold by induction. ■

4.2 Symbolic Subtyping

Given the preliminary definitions of the previous Section, we can now propose our notion of symbolic subtyping. This notion is *symbolic* in the sense that it is defined by a list of *simulation rules* depending on the structure of types. We will use $<:$ to denote symbolic subtyping. In the following definition we use symbols N and M to indicate normalized tree types $l[T]$ or B .

Definition 4.8 (Symbolic subtyping) *Symbolic subtyping* $<:$ *among normalized types is defined as shown in Table 4.2.*

4.3 Properties of Symbolic Subtyping

Lemma 4.9 (Accumulation) *Let T be such that $norm(T') = C[T]$, and the union $S \mid V$ such that $norm(U') = C'[S \mid V]$, with U' compact and S and V prime. If $T < S \mid V$ then either $T < S$ or $T < V$.*

Proof. Consider $S \mid V$. By Lemma 4.7 we know that one of the three following cases must hold.

1. $topLT(S) \bigtriangleup topLT(V) \neq \emptyset$ and, if $f : S$ and not $f : V$, then $\exists\, l \in topLT(f)$ such that $l \notin topLT(V)$, and vice versa, where $\bigtriangleup$ is the symmetrical difference operator;
2. $topLT(S) \subset topLT(V)$ and $V = V', l[V'']$ and $l \notin topLT(S) \cup topLT(V')$;
3. $topLT(V) \subset topLT(S)$ and $S = S', l[S'']$ and $l \notin topLT(V) \cup topLT(S')$;
4. $topLT(S) = topLT(V)$ and $S = \mathcal{A}, \mathcal{D}, (S')*$ and $V = \mathcal{B}, \mathcal{D}, (V')*$ with
   - $topLT(\mathcal{A}) \bigtriangleup topLT(\mathcal{B}) \neq \emptyset$
   - $\mathcal{A} = \mathcal{A}_1, \ldots, \mathcal{A}_n$ and $\mathcal{B} = \mathcal{B}_1, \ldots, \mathcal{B}_m$
   - $S' = (\bigcup_1^n A_i) \mid (\bigcup_1^m B_i) \mid S'' = V'$
   - $topLT(\mathcal{D}) \cap topLT(S') = \emptyset$ and $(topLT(\mathcal{A}) \cup topLT(\mathcal{B})) \cap topLT(S'') = \emptyset$

We first deal with the fourth case, which is the hardest one. We define $\mathcal{H} \bowtie \mathcal{E}$ iff $topLT(\mathcal{H}) = topLT(\mathcal{E})$.

We first assume that $T = (T')*, \mathcal{Z}$. By hypothesis we have $\mathcal{Z} < S \mid V$. This means that $\mathcal{Z} = \mathcal{X}, \mathcal{Y}, \mathcal{G}$ with $\mathcal{X} \bowtie \mathcal{A}$ and not $\mathcal{X} \bowtie \mathcal{B}$, or vice versa, since $topLT(\mathcal{A}) \bigtriangleup topLT(\mathcal{B}) \neq \emptyset$; moreover in both cases we have $\mathcal{Y} \bowtie \mathcal{D}$. Wlog we assume the first case.

Now we prove that $\mathcal{X} < \mathcal{A}$. This follows by simply observing that $\mathcal{X} \bowtie \mathcal{A}$, that top level tags in $\mathcal{X}$ are not in $\mathcal{D}$, and that $\mathcal{A} = \mathcal{A}_1, \ldots, \mathcal{A}_m$ with the property : $S' = (\bigcup_1^n A_i) \mid (\bigcup_1^m B_i) \mid S'' = V'$ and $(topLT(\mathcal{A}) \cup topLT(\mathcal{B})) \cap topLT(S'') = \emptyset$. This means that for each $f = f', f''$ such that $f : S$, or $f : V$, and f' contains at the top level tags at the top level of $\mathcal{A}$, f' must be a product $f_{\mathcal{A}_1} \ldots f_{\mathcal{A}_n}$.

Moreover it is straightforward that $\mathcal{Y} < \mathcal{D}$. So it remains to prove that $\mathcal{G} < S'*$. To this end, suppose this is not true. We would have soon an absurd since this would mean that there exists $f : \mathcal{G}$ such that not $f : S'*$ and not $f : V'*$ (recall that $S' = V'$). Now, by Lemma 4.4, entailing that both $\mathcal{A}$ and $\mathcal{B}$ are subtype of $S'*$ and $V'*$, if we take an $f_{\mathcal{X},\mathcal{Y}}$ we have that not $f_{\mathcal{X},\mathcal{Y}}, f' : S \mid V$, which is an absurd.

So we can conclude that $\mathcal{Z} < \mathcal{S}$. To complete the proof we prove that $T'* < (S')*$ (that is $T'* < (V')*$). We first observe that $topLT(\mathcal{T}') \cap topLT(\mathcal{D}) = \emptyset$, by uniqueness of tags at the top level of $\mathcal{D}$ of forests in both S and V. This means that top level tags of $f : T'*$ are included in those of $S' = V'$, and since both $\mathcal{A}$ and $\mathcal{B}$ are subtypes of $S'*$ and $V'*$, we have that it must be $T'* < \mathcal{A}, (S')* < (S')*$ (and $T'* < \mathcal{B}, (V')* < (V')*$, with, we insist, $S' = V'$, so to obtain the desired property. This completes the case $T = (T')*, \mathcal{Z}$.

The case $T = \mathcal{Z}$ has indeed already been proved, while it case $T = (T')*$ implies that both $\mathcal{A}, \mathcal{D}$ and $\mathcal{B}, \mathcal{D}$ are the empty sequence type $()$. So the proof follows by reasonings similar to the previous ones.

Let us now consider the case that S and V meet property 2 (the third is strictly similar). In this case, we have $V = V', l[V'']$ and $l \notin topLT(S) \cup topLT(V')$. Now suppose that there exists $f, l[f'] : T$. This means that each forest f_T in T contains only one top level tag l since, due to $l \notin topLT(S) \cup topLT(V')$, otherwise we contradict the hypothesis $T < S \mid V$. Moreover this also means that in must be $f_T : V$ for each $f_T : T$.

Suppose now that does not exists $f, l[f'] : T$. Since $T < S \mid V$ this means that $T < S$, since each forest in V contains an l top level tag.

The only remaining case is that that S and V meet property 1. By aiming at a contradiction, assume the thesis not to hold. This means that $\exists f_S \in [\![T]\!] \cap [\![S]\!]$ and $\exists f_V \in [\![T]\!] \cap [\![V]\!]$ such that $f_S \notin [\![V]\!]$ and $f_V \notin [\![S]\!]$. By Lemma 4.7 we have that there exists a top level tag in f_S not belonging to $topLT(V)$ and that there exists a top level tag in f_V not belonging to $topLT(S)$. But Lemma 4.5 tells us that there exists $f : T$ such that $topLT(f_S) \cup topLT(f_V) \subseteq topLT(f)$, with neither $f : S$ nor $f : V$, because of the previous facts. So we have contradicted the hypothesis $T < S \mid V$ and, therefore, finished the proof. ■

Lemma 4.10 (Product Injection) *If $\mathcal{A} < \mathcal{B}$ with $\mathcal{A} = N_1, \ldots, N_n$, $\mathcal{B} = M_1, \ldots, M_m$ and $\mathcal{A}$ and $\mathcal{B}$ prime types such that there exists T compact, so that $norm(T) = C[\mathcal{B}]$, then there exists a bijection π such that $N_i < M_{\pi(i)}$.*

Proof. The proof follows by cardinality reasons and by observing that at top level of $\mathcal{B}$ we have unique tags. ■

Lemma 4.11 (Product Injection 1) *Given a prime type $\mathcal{A}$ and given a prime time $U*$ such that $norm(T) = C[U*]$ with T compact, if $\mathcal{A} < U*$, then*

$$\mathcal{A} < U \ \vee \ (\mathcal{A} = \mathcal{A}_1, \mathcal{A}_2 \ \wedge \mathcal{A}_i < U*)$$

Proof. We proceed by induction on the number of union types at the top level of U (not inside an element type of U). In the base case we have that $U = \mathcal{B}$, with $\mathcal{B}$ having unique top level tags. In this case either $\mathcal{A} < \mathcal{B}$ or $\mathcal{A} = \mathcal{A}^1, \ldots, \mathcal{A}^n$ with $\mathcal{A}^i < \mathcal{B}$.

Now suppose $U = \mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n$. We have $U = (\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n)*, \mathcal{B}_n*$. Now if $\mathcal{A} < (\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_{n-1})*$ we have the thesis by induction, since $(B_1 \mid \ldots \mid B_{n-1})* < U*$ and $(B_1 \mid \ldots \mid B_{n-1}) < U$. If this is not the case, this means that $\mathcal{A} = \mathcal{A}_1, \mathcal{A}_2$ with $\mathcal{A}_1 < (\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_{n-1})*$ and $\mathcal{A}_2 < \mathcal{B}_n*$, which gives us the desired decomposition as $(B_1 \mid \ldots \mid B_{n-1})* < U*$ and $\mathcal{B}_n* < U*$. Note that the decomposition for $\mathcal{A}$ is given by the fact that i) $\mathcal{A} < U*$, ii) $topLT(\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_{n-1}) \cap topLT(\mathcal{B}_n) = \emptyset$ and iii) not $\mathcal{A} < (\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_{n-1})*$. Indeed, iii) means that $\exists f : \mathcal{A}$ such that not $f : (\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_{n-1})*$. This togehter with ii) yields that $topLT(\mathcal{A}) \not\subseteq topLT((\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_{n-1})*)$: otherwise we would have an absurd since we end up with

$$\mathcal{A} < U* \iff \mathcal{A} < (\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_{n-1})*$$

which contradicts iii). So since $\mathcal{A} < U* \implies topLT(\mathcal{A}) \subseteq topLT(U*)$, we can obtain $\mathcal{A} = \mathcal{A}_1, \mathcal{A}_2$ with $topLT(\mathcal{A}_1) \subseteq topLT((\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_{n-1})*)$ and $topLT(\mathcal{A}_2) \subseteq topLT(\mathcal{B}_n*)$, Since i) and ii) these two properites imply that $\mathcal{A}_1 < (B_1 \mid \ldots \mid B_{n-1})*$ and that $\mathcal{A}_2 < \mathcal{B}_n*$, which concludes the proof.

■

Lemma 4.12 (Product Injection 2) *If $T*, \mathcal{C} < U*, \mathcal{B}$ and $T*, \mathcal{C}$ is a maximal prime subterm of $norm(T')$ and $U*, \mathcal{B}$ is a maximal prime subterm of $norm(U')$ with U' compact, then*

$$\mathcal{C} < U*, \mathcal{B} \ \wedge \ T* < U*$$

Proof. The part $\mathcal{C} < U*, \mathcal{B}$ is a direct consequence of the hypothesis $T*, \mathcal{C} < U*, \mathcal{B}$. For the second case we use Lemma 4.3. By this lemma we may have several cases concerning the right hand side type. Let us first consider the first case

$$U* = (\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n)*$$

and $\mathcal{B} = \mathcal{B}'_1, \ldots, \mathcal{B}'_k$ and

- $n \geq 1$, each $\mathcal{B}_i$ having unique top level tags, and $topLT(\mathcal{B}_i) \cap topLT(\mathcal{B}_j) = \emptyset$, for $i \neq j$.
- $\forall \mathcal{B}'_s . \exists \mathcal{B}_j .\ \mathcal{B}'_s = \mathcal{B}_j$.

In this case by Lemma 4.4 we know that each forest f in $U*, \mathcal{B}$ is a combination of $\mathcal{B}_i$'s forests. So the same holds for forests in $\mathcal{C}$, since $\mathcal{C} < U*, \mathcal{B}$. Now this means that the same property must hold for $T*$. Otherwise we contradict the hypothesis $T*, \mathcal{C} < U*, \mathcal{B}$. Indeed if we take a forests f_{T*} such that not $f_{T*} : U*, \mathcal{B}$, then we have not $f_{T*}, f_{\mathcal{C}} : U*, \mathcal{B}$, which is an absurd; this implication follows by induction on the number of f_{T*} top level trees.

Let's consider now the case that $\mathcal{B} = \mathcal{D}$ with

- $n \geq 1$, each $\mathcal{B}_i$ having unique top level tags, and $topLT(\mathcal{B}_i) \cap topLT(\mathcal{B}_j) = \emptyset$, for $i \neq j$.
- $topLT(\mathcal{D}) \cap topLT(\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n) = \emptyset$

This entails that $topLT(T) \cap topLT(\mathcal{D}) = \emptyset$. So this means that $T*$ can only contains combinations of $\mathcal{B}_i$'s forests, that are forests in $U*$.

The remaining case is that

- $\mathcal{B} = \mathcal{B}'_1, \ldots, \mathcal{B}'_k, \mathcal{D}$

with

- $n \geq 1$, each $\mathcal{B}_i$ having unique top level tags, and $topLT(\mathcal{B}_i) \cap topLT(\mathcal{B}_j) = \emptyset$, for $i \neq j$.
- $\forall \mathcal{B}'_s . \exists \mathcal{B}_j .\ \mathcal{B}'_s = \mathcal{B}_j$, and $topLT(\mathcal{B}'_r) \cap topLT(\mathcal{B}'_s) = \emptyset$ with $r \neq s$
- $topLT(\mathcal{D}) \cap topLT(\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n) = \emptyset$

This case actually is a combination of previous two cases, and follows with a similar reasoning. ■

Lemma 4.13 (Product Injection 3) *If $\mathcal{A} < U*, \mathcal{B}$ and $\mathcal{A}$ is a subterm of $norm(T')$ and $U*, \mathcal{B}$ is a maximal prime subterm of $norm(U')$ with U' compact, then*

$\mathcal{A} < \mathcal{B} \ \vee \ (\mathcal{A} = \mathcal{A}_1, \mathcal{A}_2 \ \wedge \mathcal{A}_1 < U* \ \wedge \mathcal{A}_2 < \mathcal{B})$

Proof. We may have $|\mathcal{A}| = |\mathcal{B}|$ or $|\mathcal{A}| > |\mathcal{B}|$, where $|\cdot|$ is the number of tree types at the top level of the argument type. The first case is obvious, so we assume $|\mathcal{A}| > |\mathcal{B}|$.

We use Lemma 4.3. By this lemma we may have several cases concerning the right hand side type. Let us first consider the first case

$$U* = (\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n)*$$

and $\mathcal{B} = \mathcal{B}'_1, \ldots, \mathcal{B}'_k$ and

- $n \geq 1$, each $\mathcal{B}_i$ having unique top level tags, and $topLT(\mathcal{B}_i) \cap topLT(\mathcal{B}_j) = \emptyset$, for $i \neq j$.
- $\forall \mathcal{B}'_s . \exists \mathcal{B}_j .\ \mathcal{B}'_s = \mathcal{B}_j$.

Now we proceed by induction on $|\mathcal{B}|$. For the base case we assume $|\mathcal{B}| = 0$, which trivially follows, as $\mathcal{A} = \mathcal{A}, ()$. Assume that V is tree type and that $\mathcal{B} = \mathcal{B}', V$.

By previous properties we have that there exists V' such that $\mathcal{A} = \mathcal{A}', V'$ with $V' < V$ (this is because for whatever tree type V'' at the top level of $U*, \mathcal{B}$ having the same top level tag as V, it is $V'' = V$). As a consequence we have $\mathcal{A}' < U*, \mathcal{B}'$, and then the thesis by induction.

The second case is that $\mathcal{B} = \mathcal{D}$ with

- $topLT(\mathcal{D}) \cap topLT(\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n) = \emptyset$

This case is similar to the previous one, even easier actually thanks to uniqueness of $\mathcal{D}$ tags.

The remaining case is that

- $\mathcal{B} = \mathcal{B}'_1, \ldots, \mathcal{B}'_k, \mathcal{D}$

with

- $n \geq 1$, each $\mathcal{B}_i$ having unique top level tags, and $topLT(\mathcal{B}_i) \cap topLT(\mathcal{B}_j) = \emptyset$, for $i \neq j$.
- $\forall \mathcal{B}'_s . \exists \mathcal{B}_j .\ \mathcal{B}'_s = \mathcal{B}_j$, and $topLT(\mathcal{B}'_r) \cap topLT(\mathcal{B}'_s) = \emptyset$ with $r \neq s$
- $topLT(\mathcal{D}) \cap topLT(\mathcal{B}_1 \mid \ldots \mid \mathcal{B}_n) = \emptyset$

This is a combination of previous two cases, and follows with similar reasonings.

■

Lemma 4.14 (* Elimination) *$T* < U*$ iff $T < U*$*

Proof. The only non trivial direction is $T* < U*$ if $T < U*$. To prove this it suffices to add a star to $T < U*$ so to obtain $T* < U**$, which means $T* < U*$.

■

4.4 Equivalence between symbolic and semantic subtyping

It is now possible to prove that our symbolic characterization of semantic subtyping is correct and that, when U in $T < U$ is compact, then it is also complete.

Theorem 4.15 (Soundness) *If* $norm(T) <: norm(U)$ *then* $T < U$.

Proof. The proof easily follows by cases inspection of Table 4.2. ∎

Theorem 4.16 (Completeness) *If* $T < U$ *and* U *is compact then* $norm(T) <: norm(U)$.

Proof. For this case all the work has already been done: the proof easily follows induction over the structure of types and by case distinction of compared types, and by applying Lemma 4.9 for the case when the left hand-side type is a union type, and lemmas 4.10, 4.11, 4.12, and 4.14 in the cases of product and $*$ types. ∎

5 Symbolic Subtype-checking

Our algorithm for symbolic subtype-checking (SIM(T_1,T_2)) is shown in Figures 2 and 3. It consists of three main phases. During Phase 1, the algorithm creates and populates a type matrix *simTypes* with boolean values or symbolic references, both obtained by repeatedly calling the SIMPLESUB algorithm of Figure 2. The matrix has as many rows as the type terms in T_1, and as many columns as type terms in T_2. To ensure the proper behavior of the algorithm, type terms from both T_1 and T_2 have to be extracted with a *pre-order* visit, so that, if Z_i and Z_j are type terms in T_1 and $i < j$, then Z_i precedes Z_j in the pre-order visit of T_1. The SIMPLESUB algorithm is called once per each cell in the type matrix, with the notable exception of those cells that correspond to types whose match is useless as they occupy incompatible positions in the type term hierarchy (this task is performed by the boolean function COMPARABLE). For instance, given $T_1 = l\,[\mathsf{B}, m\,[\mathsf{B}]]$ and $T_2 = l\,[(\mathsf{B}, p\,[\mathsf{B}]) \mid (\mathsf{B}, m\,[\mathsf{B}])]$, comparing the first B inside T_1 with the B inside $p\,[\mathsf{B}]$ in T_2 is a waste of time, as their different positions in the hierarchies of T_1 and T_2, respectively, prevent their matching.

SIMPLESUB returns a boolean value for any comparison that does not require any further type comparisons. If, instead, the comparison between T and U requires a further comparison, as in the type simulation rule for $l\,[Z] <: l\,[W]$ (Rule 4), then the algorithm returns a simple symbolic reference ($ref(U_x, V_y)$), a logical combination of simple references (e.g., $\bigwedge_{i=1}^{n} ref(U_i, T_2)$), or a *product* reference ($ref(\{U_1, \ldots, U_n\} \otimes \{V_1, \ldots, V_m\})$). A simple reference $ref(U_x, V_y)$ is a symbolic pointer to the content of the cell $simTypes[x][y]$, while a product reference indicates that the maximum flow algorithm must be executed on top of a bipartite graph built from the partitions $\{U_1, \ldots, U_n\}$ and $\{V_1, \ldots, V_m\}$. References are left unevaluated, and they will be solved during Phase 2.

The output of Phase 1, thus, is a partially instantiated type matrix. This matrix is the input for Phase 2, whose objective is to solve symbolic references. By visiting the matrix in reverse order, starting from the bottom right and

```
SimpleSub(Type T1, Type T2)
 1  switch
 2    case T1 = T2 :
 3         return true
 4    case T1 = () ∧ T2 = V* :
 5         return true
 6    case T1 = l[U1] ∧ T2 = l[U2] :
 7         return ref(U1, U2)
 8    case T1 = U1, ..., Un ∧ T2 = V1, ..., Vm :
 9         return ref({U1, ..., Un} ⊗ {V1, ..., Vm})
10    case T1 = U1, ..., Un ∧ T2 = V1 | ... | Vm :
11         return (⋁_{j=1}^{m} ref(T1, Vj))
12    case T1 = U1 | ... | Un ∧ T2 = V1, ..., Vm :
13         return (⋀_{i=1}^{n} ref(Ui, T2))
14    case T1 = U * ∧T2 = V* :
15         return ref(U, T2)
16    case T1 = U1, ..., Un ∧ T2 = V* :
17         return (⋀_{i=1}^{n} ref(Ui, T2))
18    case T1 = U ∧ T2 = V* :
19         return ref(U, V)
20    case default :
21         return false
```

Figure 2: SimpleSub algorithm.

going right to left, the algorithm proceeds by replacing references of the form $ref(U_x, V_y)$ with the content of $simType[x][y]$, by evaluating logical combiners, and by applying the maximum flow algorithm for $\otimes$ references; in the latter case, an auxiliary (and trivial) function GraphConstr is invoked with the aim of building the bipartite graph $\mathcal{G}$, while the maximum flow is computed with a standard maximum flow algorithm MaxFlow; MaxFlow returns the maximum flow traversing the bipartite graph as well as the set of nodes in the second partition that are reached by the flow. The result of this phase is a fully instantiated type matrix, since, as shown by Lemma 5.5, when a cell $simTypes[i][j]$ has been reached, all the cells that can be referenced by its content have already been visited and instantiated. It should be observed that, as prescribed by Rule 10 of the simulation relation, nodes in the second partition of $\mathcal{G}$, corresponding to *-types in the right hand-side of the comparison, are marked as *special*, since they can be used to map more than one term of the left hand-side; from a flow point of view, this means the edges connecting these nodes with the sink of the graph have unbounded capacity.

Phase 3 is very simple and consists in returning a boolean value describing the result of the whole simulation. Since the types being compared correspond to the first row and to the first column of the type matrix, the algorithm just returns the content of $simTypes[1][1]$.

The use of a maximum flow algorithm for implementing Rules 5 and 6 may

```
SIM(Type T1, Type T2)
 1  // we assume T1 to be composed by n terms and T2 by m terms
 2  // phase 1: type matrix construction
 3  Array[n][m] simTypes
 4  for each U_i in T1
 5  do for each V_j in T2
 6      do if COMPARABLE(U_i, V_j)
 7            then simType[i][j] = SIMPLESUB(U_i, V_j)
 8  // phase 2: reference resolution
 9  for i ← n to 1
10  do for j ← m to 1
11      do if simTypes[i][j] = ref(Z_p, U_q) ∧ simTypes[p][q] ∈ {true, false}
12            then simTypes[i][j] = simTypes[p][q]
13            else if simTypes[i][j] contains references different from ⊗
14                    then for each ref(Z_x, V_y) in simTypes[i][j]
15                          do replace ref(U_x, V_y) with simTypes[x][y]
16                             evaluate the logical expression in simTypes[i][j]
17                    else if simTypes[i][j] = ref({U_f, ..., U_p} ⊗ {V_g, ..., V_q})
18                            then P1 = {U_f, ..., U_p}
19                                 P2 = {V_g, ..., V_q}
20                                 mark as special P2 nodes corresponding to *-types
21                                 G = GRAPHCONSTR(P1, P2)
22                                 (flow, Pc) = MAXFLOW(G)
23                                 if flow = ||P1|| ∧ Pc = P2
24                                    then simTypes[i][j] = true
25                                    else simTypes[i][j] = false
26  // phase 3: result discovery
27  return simTypes[1][1]
```

Figure 3: Symbolic subtype-checking algorithm.

appear as an overkill since the restrictions imposed by type compactness make comparison of product types quite simple. In particular, as no element tags can be repeated at the top level of the right hand-side product, it is easy to see that a simple nested loop between the left hand-side factors and the right hand-side factors would suffice to implement the simulation rules. We chose to use a more powerful algorithm in order to broaden the class of *non-compact* types on which the algorithm is still complete; as Ford/Fulkerson maximum flow algorithm is still polynomial, we found this option a viable solution to extend the applicability of our approach. Several experiments on types with wide and systematic violations of the type compactness property validate this claim.

5.1 Correctness and Completeness

To prove correctness and completeness of the SIM algorithm wrt type simulation, we need the following Lemmas.

Lemma 5.1 *The* SIMPLESUB *algorithm satisfies the reflexivity, commutativity, associativity, and ()-neutrality properties.*

Proof. The lemma is proved by analyzing the behavior of the algorithm.
Reflexivity:
Reflexivity is directly proved by case 2/3;
()**-neutrality**:
Consider $T = T_1, ()$. By applying case 8/9, SIMPLESUB(T, T_1) returns a $\otimes$ reference of the form $ref(\{T_1, ()\} \otimes \{T_1\})$, which instructs the SIM to build a bipartite graph $\mathcal{G}$, where $\mathcal{P}_1 = \{T_1, ()\}$ and $\mathcal{P}_2 = \{T_1\}$. The maximum flow algorithm automatically discards () types during maximum flow analysis.
The case of SIMPLESUB(T_1,T) is trivial.
Commutativity:
Consider $T = T_1, T_2$ and $U = T_2, T_1$. SIMPLESUB(T,U) returns a $\otimes$ reference $ref(\{T_1, T_2\} \otimes \{T_2, T_1\})$, which instructs SIM to build a bipartite graph $\mathcal{G}$, whose partitions are equal.
Associativity: Consider types $T = (T_1, T_2), T_3$ and $U = T_1, (T_2, T_3)$. The algorithm just drops the parentheses from T and U, hence the proof is trivial. ■

Lemma 5.2 *The* SIMPLESUB *algorithm is* compatible *with the simulation relation, in the sense that it implements the simulation relation.*

Proof. The lemma is proved by analyzing the correspondence between the simulation rules and the algorithm cases. A preliminary observation is that side conditions of the form $T <: U$ are encoded by a simple reference of the form $ref(T, U)$.
Completeness
We want to prove that each simulation rule can be encoded in the algorithm. We proceed by induction on the simulation rules.

$() <: ()$**:** By reflexivity.

$() <: T*$**:** By case 4/5.

$B <: B$**:** By reflexivity.

$l\,[T] <: l\,[U]$**:** By case 6/7.

$N_1, \ldots, N_n <: M_1, \ldots, M_m$**:** Case 8/9 is applied; the algorithm returns a $\otimes$ reference of the form $ref(\{N_1, \ldots, N_n\}, \{M_1, \ldots, M_m\})$, which instructs the SIM algorithm to build a bipartite graph $\mathcal{G}$ where $\mathcal{P}_1 = \{N_1, \ldots, N_n\}$ and $\mathcal{P}_2 = \{M_1, \ldots, M_m\}$. The thesis follows from induction on the rule side conditions.

$U*, N_1, \ldots, N_n <: V*, M_1, \ldots, M_m$**:** As for the previous rule, case 8/9 is applied and the thesis follows from induction on the rule side conditions.

$T_1 <: U_2 \mid U_3$**:** Case 10/11 is applied, hence the algorithm returns $ref(T_1, V_1) \vee ref(T_1, V_2)$. The thesis follows from induction on the rule side conditions.

$T_1 \mid T_2 <: U$**:** Case 12/13 is applied, hence the algorithm returns $ref(T_1, U) \wedge ref(T_2, U)$. The thesis follows from induction on the rule side conditions.

$T <: U*$**:** Case 18/19 is applied.

$T* <: U*$**:** Case 14/15 is applied.

$T_1, T_2 <: U*$**:** Case 16/17 is applied.

Correctness

We want to prove each algorithm case is backed by the application of the simulation rules. We proceed by induction on the algorithm cases.

Case 2/3: By reflexivity.

Case 4/5: By ()-neutrality.

Case 6/7: By Rule 4.

Case 8/9: If both T_1 and T_2 contain a star type, Rule 6, combined with type equivalence modulo commutativity, is applied. Otherwise, Rule 5, combine with type equivalence modulo commutativity, is applied.

Case 10/11: If $n = 1$, it suffices to apply Rule 7. If $n > 1$, Rule 7 is applied to decompose the right hand-side of the comparison; then, Rules 5 and 6 are applied.

Case 12/13: Rule 8 is applied to decompose the left hand-side of the comparison; Rules 5 and 6 are then applied to the resulting terms.

Case 14/15: It suffices to apply Rule 10.

Case 16/17: Rule 11 is iteratively applied.

Case 18/19: Rule 9 is applied.

To conclude the proof, it should be observed that the algorithm does not contain any other case. ■

Lemma 5.3 *Given $i \in [1, \ldots, n]$, given $j \in [1, \ldots, m]$, if $simTypes[i][j]$ contains a reference $ref(Z_x, V_y)$, then $x \geqslant i$ and $y \geqslant j$.*

Proof. By a simple inspection of the SimpleSub algorithm. ∎

Lemma 5.4 *Given $i \in [1, \ldots, n]$, given $j \in [1, \ldots, m]$, if $simTypes[i][j]$ contains a reference of the form $ref(\{Z_f, \ldots, \mathcal{Z}_h\} \otimes \{V_p, \ldots, V_q\})$, then $i \leqslant f, \ldots, i \leqslant h$ and $j \leqslant p, \ldots, j \leqslant q$*

Proof. By a simple inspection of the SimpleSub algorithm. ∎

These lemmas allows us to prove the following Lemma about the second phase of the Sim algorithm.

Lemma 5.5 *Phase 2 of the* Sim *algorithm produces a fully instantiated type matrix.*

Proof. Phase 2 consists of a reverse order visit of the type matrix. When a cell $simTypes[i][j]$ is examined, by Lemmas 5.3 and 5.4, it may contain only forward references, i.e., references to already visited cells. These references have already been solved, hence the cell $simTypes[i][j]$ can be fully instantiated. ∎

We can now state the correctness and completeness of Sim wrt type simulation.

Theorem 5.6 *The* Sim *algorithm is correct and complete wrt the type simulation relation.*

Proof. We first observe that the algorithm terminates, as it does not make recursive calls nor it uses unbounded iterations.

We can now prove the thesis by analyzing each phase of the algorithm.

Phase 1

Assuming that T_1 is formed by n type terms (from Z_1 to Z_n), and that T_2 is formed by m type terms (from V_1 to V_m), Phase 1 creates a matrix $simTypes$ of $n \times m$ entries, where each entry contains a boolean value or a reference to other entries: $simTypes[i][j]$ indicates whether Z_i is similar to V_j (**true** or **false**); if the simulation cannot be directly computed, a reference to entries of nested terms is inserted. For each entry, Phase 1 calls the SimpleSub algorithm, which, as shown by Lemma 5.2, implements the rules of the definition of type simulation.

Phase 2

Phase 2 solves symbolic references in the matrix entries. The only potential source of incompleteness is the comparison among product types. However, we have already shown, even if informally, that this comparison is equivalent to a $0-1$ maximum flow problem on bipartite graphs, and that our algorithm is able to capture all matchings among product types.

Phase 3

Phase 3 just outputs the result of the algorithm.

∎

5.2 Complexity Analysis

To study the complexity of the SIM algorithm we must first analyze the complexity of the auxiliary algorithms SIMPLESUB.

Lemma 5.7 *The* SIMPLESUB *algorithm has* $O(1)$ *worst case complexity.*

Proof. We assume the pattern matching on types (e.g., **case** $T_1 = \ldots$) has $O(1)$ complexity.

Each case in the SIMPLESUB algorithm performs no recursive calls nor iterations; furthermore, each reference returned by the return clause is just a symbolic reference and does not involve any operation to be performed. As a consequence, each case requires just $O(1)$ operations. ■

Theorem 5.8 *The worst case complexity of the* SIM *algorithm, while comparing* T *and* U*, is* $O(nm(n+m)^3)$*, where* n *is the number of terms in* T*, and* m *is the number of terms in* U*.*

Proof. We prove the thesis by analyzing each phase of the algorithm.
Phase 1

Phase 1 creates a matrix of $n \times m$ entries, where each entry contains the projection relation between two types. For each entry, phase 1 calls the SIMPLESUB algorithm, which, as shown in Lemma 5.7, has $O(1)$ complexity. Moreover, each calls to the function COMPARABLE has $O(1)$ complexity, as it involves a single lookup in a compatibility matrix that can be built in $O(nm)$ time during type parsing. Phase 1, hence, performs $O(nm)$ operations.
Phase 2

Phase 2 performs the resolution of symbolic references by traversing the type matrix in reverse order. Simple references of the form $ref(Z_x, V_y)$ can be solved with a single access to the matrix, while each logical expression of the form $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} ref(U_i, V_j)$ can be evaluated in at most nm operations.

The resolution of $\otimes$-references requires the construction of the bipartite graph $\mathcal{G}$ ($O(nm)$ operations), and the execution of the $0-1$ maximum flow algorithm. The best $0-1$ maximum flow algorithm on bipartite graphs is a variant of the Ford-Fulkerson algorithm and has $O((n+m)^3)$ complexity [15], hence this phase has $O(nm(n+m)^3)$ worst case time complexity.
Phase 3

Phase 3 requires just one operation.

■

It should be noted that, while SIM has polynomial complexity, the overall approach has exponential complexity, as types must be normalized (via $norm()$) before the application of SIM.

6 Tractable Subtype-checking

In the previous Sections we described a class of unordered XML types for which subtype-checking can be performed in EXPTIME. As most of the complexity of our approach is confined into the $norm()$ function, which transforms a given

type into an equivalent disjunctive normal form representation, inclusion becomes tractable for any type class for which normalization can be performed in polynomial time. To this purpose, we identified a large class of types (ESG types) not in DNF for which normalization is polynomial. ESG types are defined by the following grammar.

Definition 6.1 (ESG types) *A type T is* element/star-guarded *if it can be generated by the following grammar:*

$$\begin{array}{ll} \textbf{\textit{ESG-Types}} & T ::= () \mid \mathsf{B} \mid l[U] \mid T,T \mid U* \\ \textbf{\textit{Union Types}} & U ::= T \mid T \mid U \end{array}$$

By the above grammar, only union types inside the immediate scope of a repetition or a tree type are legal. Hence, types $(l[\mathsf{B}] \mid m[\mathsf{B}])*$ and $l[\mathsf{B} \mid m[\mathsf{B}]]$ are ESG types, while type $(l[\mathsf{B}] \mid m[\mathsf{B}]), p[\mathsf{B}]$ does not satisfy this restriction.

As shown in [3], the restriction imposed by the grammar of ESG types is met by many practical DTDs: indeed, author considered a sample comprising DTDs extracted from the Web and found that about 77% of all the union types used in the sample were $*$-guarded.

It is easy to see that $norm()$ is polynomial when executed on ESG types.

Lemma 6.2 *$norm()$ is polynomial on ESG types.*

By combining this result with the complexity analysis of the previous Section, we can state the following theorem.

Theorem 6.3 *Given an ESG type T, and given an ESG, compact type U, $T < U$ can be decided in polynomial time.*

Proof. The thesis easily follows from theorems 4.16, 5.8, and from Lemma 6.2. ∎

It is worth to notice that Theorem 6.3 implies that equivalence and membership for ESG, compact types can be decided in PTIME too. We are currently investigating the complexity of the intersection problem.

7 Related Works

The properties of unordered XML types have become the subject of several recent works. In [13] authors discuss the techniques and heuristics they used in implementing a type-checker for unordered XML types. Their implementation is based on sheaves automata with Presburger arithmetic. Interestingly, authors describe the same backtracking phenomena we addressed here (see the accumulation lemmas of Section 4.3): this fact proves our claim that these phenomena are independent from the approach used for implementing the subtyping operation.

In [14] authors deeply analyze the complexity of basic decision problems (i.e., inclusion, intersection, and equivalence) for XML types with numerical occurrence constraints and interleaving. They prove that inclusion for unordered XML types is inherently more expensive than inclusion for ordered types, both

in the case of DTDs and in the case of EDTDs. Authors also describe a class of XML types for with numerical occurrence constraints (but without interleaving) for which inclusion is in PTIME; this class comprises types in conjunctive normal form with severe limitations on the use of repetition types. On the contrary, we focus on unordered types only, and describe a wider class of types (ESG types) for which inclusion can be decided in PTIME.

8 Conclusions and Future Work

In this paper we described a class of unordered XML types for which subtype-checking can be decided in EXPTIME, as well as a class for which subtyping is tractable. The first class is identified by a non-ambiguity property that is enjoyed by most DTDs and XML Schemas, and that represents only a mild restriction on the type language. We encoded semantic subtyping into a simulation-based, symbolic relation for which we provided a correct and complete algorithm.

Our future work moves along two lines. First, we want to analyze what happens when types compactness is combined with grammatical restrictions like CHARE[&] [14]: our intuition is that inclusion for CHARE[&] compact types can be performed in PTIME, but we have no dispositive evidence. Second, we want to more deeply analyze the properties of our approach, in particular for what concerns complexity lower bounds.

References

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

[2] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an xml-centric general-purpose language. In *ICFP*, pages 51–63, 2003.

[3] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. Dtds versus xml schema: A practical study. In Sihem Amer-Yahia and Luis Gravano, editors, *WebDB*, pages 79–84, 2004.

[4] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. Technical report, World Wide Web Consortium, Oct 2004. W3C Recommendation.

[5] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, Nov 2006. W3C Proposed Recommendation.

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

[7] James Clark. XSL Transformations (xslt) Version 1.0. Technical report, World Wide Web Consortium, Nov 1999. W3C Recommendation.

[8] Dario Colazzo., Giorgio Ghelli, Paolo Manghi., and Carlo Sartiani. Static analysis for path correctness of XML queries. *Journal of Functional Programming*, 16(4-5):621–661, 2006.

[9] Dario Colazzo and Carlo Sartiani. An efficient algorithm for xml type projection. In *Proceedings of the Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06), Venice, Italy, 10-12 July 2006.*, 2006.

[10] Silvano Dal-Zilio, Denis Lugiez, and Charles Meyssonnier. A logic you can count on. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 135–146. ACM, 2004.

[11] Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, November 2006. W3C Proposed Recommendation.

[12] Mary F. Fernández and Jérôme Siméon. Building an extensible xquery engine: Experiences with galax (extended abstract). In Zohra Bellahsene, Tova Milo, Michael Rys, Dan Suciu, and Rainer Unland, editors, *XSym*, volume 3186 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2004.

[13] J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. A logic your typechecker can count on: Unordered tree types in practice, September 2006. To appear in PLAN-X '07.

[14] Wouter Gelade, Wim Martens, and Frank Neven. Optimizing schema languages for xml: Numerical constraints and interleaving. In *To appear in Proceedings of the International Conference on Database Theory 2007 (ICDT 2007)*, 2007.

[15] Andrew V. Goldberg. Recent developments in maximum flow algorithms. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 1998.

[16] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

[17] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for simple regular expressions. In Jirí Fiala, Václav Koubek, and Jan Kratochvíl, editors, *MFCS*, volume 3153 of *Lecture Notes in Computer Science*, pages 889–900. Springer, 2004.

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