Subsumption for XML Types

Gabriel M. Kuper and Jérôme Siméon

Bell Laboratories, 600 Mountain Avenue, 07974, NJ, USA
{kuper,simeon}@research.bell-labs.com
http://www-db.research.bell-labs.com/user/{kuper,simeon}

Abstract. XML data is often used (validated, stored, queried, etc) with respect to different types. Understanding the relationship between these types can provide important information for manipulating this data. We propose a notion of subsumption for XML to capture such relationships. Subsumption relies on a syntactic mapping between types, and can be used for facilitating validation and query processing. We study the properties of subsumption, in particular the notion of the greatest lower bound of two schemas, and show how this can be used as a guide for selecting a storage structure. While less powerful than inclusion, subsumption generalizes several other mechanisms for reusing types, notably extension and refinement from XML Schema, and subtyping.

1 Introduction

XML [5] is a data format for Web applications. As opposed to e.g., relational databases, XML documents do not have to be created and used with respect to a fixed, existing schema. This is particularly useful in Web applications, for simplifying exchange of documents and for dealing with semistructured data. But the lack of typing has many drawbacks, inspiring many proposals [2–4, 10, 12, 23, 24, 33] of type systems for XML. The main challenge in this context is to design a typing scheme that retains the portability and flexibility of untyped XML. To achieve this goal, the above proposals depart from traditional typing frameworks in a number of ways. First, in order to deal with both structured and semistructured data, they support very powerful primitives, such as regular expressions [2, 10, 26, 33, 28] and predicate languages to describe atomic values [2, 6, 10]. Secondly, documents remain independent from their type, which allows the same document to be typed in multiple ways according to various application needs. These features result in additional complexity: the fact that data is often used with respect to different types, means that it is difficult to recover the traditional advantages (such as safety and performance enhancements) that one expects from type systems. To get these advantages back, one need to understand how types of the same document relates to each other.

In this paper, we propose a notion of subsumption to capture the relationship between XML types. Intuitively, subsumption captures not just the fact than one type is contained in another, but also captures some of the structural relationships between the two schemas. We show that subsumption can be used to facilitate commonly used type-related operations on XML data, such as type assignment, or for query processing.

We compare subsumption with several other mechanisms aimed at reusing types. Subsumption is less powerful than inclusion, but it captures *refinement* and *extension*, recently introduced by XML Schema [33], subtyping, as in traditional type systems, as well as the instantiation mechanism of [10, 32]. As a consequence, subsumption provides some formal foundations to these notions, and techniques to take advantage of them.

We study the lattice theoretic properties of subsumption. These provide techniques to rewrite inclusion into subsumption. Notably we show the existence of a *greatest lower bound*. Greatest lower bound captures the information from several schemas, while preserving the relationship with them, and can be used as the basis for storage design.

Practical scenario. To further motivate the need for a subsumption mechanism for XML, consider the following application scenario. In order to run an integrated shopping site for some useful product, such as mobile phone jammers, company "A" accesses catalogs from various sources. The first catalog, on the left below, is taken from company "SESP" [22], while the second, on the right, is extracted from miscellaneous pages.

```
<products>
 <jammer>
  <company>SESP</company>
  <name>VHP Jammer</name>
  <price><onrequest/></price>
  <case><type>Mobile Attache Case</type>
  </case></jammers>

 <jammer>
  <company>SESP</company>
  <name>Full Milspec. Portable
        High  Power (HP) Jammer</name>
  <price><onrequest/></price>
  <case><type>Rugged military
              type case</type></case>
  <booster><range>1km</range></booster>
  <supplement>39</supplement></jammer>
  ...
```

```
<products>
 <jammer>
  <name>Static HP Jammer</name>
  <price><onrequest/></price>
  <case><type>metal</type>
        <size>180x180x80mm
         </size></case></jammer>
 <jammer>
  <company>JamLogic</company>
  <name>Personal Jammer</name>
  <price><onrequest/></price>
  <input>Digital/Analog</input>
  <warranty>2 years</warranty>
 </jammer>
 <jammer>
  <name>Cell-Phone Jammer</name>
  <price>749</price></jammer>
  ...
```

Company "SESP" only sells high power jammers, and provides precise information about their products as the SESP schema, given on the left hand side below[1]. This schema indicates that the `SESPcatalog` (we write types in upper case and element names in lower case), is composed of an element with name `products`, which has 0 or more children of type `HPJammer` ('`*`' stands for the Kleene star). `HPJammer`s have a `company` sub-element which is always `"SESP"`, a `name`, etc., and may have a `booster` option with a `supplement` cost. On the right-hand side is the schema used by company "A". Because it accesses jammer information from many places, it supports a more general description where

[1] Note that we will write some of the examples using the concrete schema syntax developed for the YAT System [10]

Jammers might not have a company information, and may have any kind of Option, with or without a supplement.

```
SESPCatalog := products *HPJammer;

HPJammer :=
 jammer [ company [ "SESP" ],
          name [ String ],
          price [ Int | onrequest ],
          case [ type [ String ],
                 ?size [ String ] ],
          ?(booster [ range [ Int ] ],
          supplement [ Int ] ) ];
```

```
IntegratedCatalog := products
                       * Jammer;

Jammer :=
 jammer [ ?company [ String ],
          name [ String ],
          price [Int|onrequest],
          *(Option,
            ?supplement [ Int ]
                            ) ] ];
Option := Symbol *Any;
```

Because it knows precisely the type of its data, company SESP can support more efficient storage (using, for instance, techniques in [14, 18, 31]), with fast access to the name, price and case information. But the fact that company "A" assumes a different type for the same data results in a mismatch. Verifying that type SESPCatalog is included in type IntegratedCatalog allows company "A" to make sure the information provided by SESP will conform to the structure expected by the application. However, this will not help in performing further operations, such as: actually assigning types of the integrated schema to elements of the SESP document, or understanding that the name and price elements can be efficiently accessed using the storage used by company SESP. Doing so requires to understand that the name and price in the Jammer type *are related to* the name and the price elements in the HPJammer type. We shall see that subsumption allows one to understand this relationship and to take advantage of it.

Another important use of typing is to support better query processing. To find all jammers that have a two years warranty, one can write the following YAT$_L$ [10, 16, 11] query:

```
define q($x) = make $n
               match $x with products/jammer/{ name/$n,
                                               warranty/$w }
               where contains($w,"2 years");
```

whose input type is:

```
q_type := products * jammer [ *(Name | Warranty | Other) ];
Name := name * Any1;
Warranty := warranty * Any2;
Other := !name!warranty * Any;
Any1 := true [ Any* ]; Any2 := true [ Any* ]
```

where ! stands for tag negation, i.e., any tag other than name and warranty.

Company "A" might wish to support queries on all Jammers, but more efficient access for this query, i.e. for products with a warranty. The relational approach [30] would be to use a specific access structure for the warranty field, but the integrated schema does not mention it. We will see that the greatest lower bound of the query type and the integrated schema is a new schema (with an explicit `warranty` field) that can be used for storage design, while the relationships with the original schemas are preserved through subsumption.

Organization of the paper. Section 2 introduces the type system we will use in the rest of the paper (essentially that of [2]) and the notion of type assignment. Section 3 defines subsumption, investigates its properties and its use for validation. Section 4 compares it to other relations on types, such as inclusion, refinement and extension in XML Schema, etc. Section 5 studies the greatest lower bound, the corresponding lattice, and how this can be used to bridge the gap between inclusion and subsumption. Section 6 discusses how one can take advantage of subsumption for storage and query processing. Section 7 summarizes related works and indicates directions for future work.

2 Data model and type system

Data model. The data model, based on ordered labeled trees with references, is similar to other previously proposed models [10, 15, 25, 28]. $\mathcal{O}$ denotes a fixed (infinite) set of *object ids* and $\mathcal{L}$ a fixed set of *labels*. References are modeled as a special type of node, that is labeled with a distinguished symbol "&" in $\mathcal{L}$ and has exactly one child. The root of the database is treated specially: A database is a tree with a root "$\triangle$", which has *no* label, and cannot be referenced by any node. (The reason for the special treatment of the root is explained later.)

Definition 1. *A* database *is a structure* $D = \langle O_D, label_D, children_D\rangle$, *where*

1. $O_D \subset \mathcal{O}$;
2. $label_D$ *is a mapping from* O_D *to* $\mathcal{L}$;
3. $children_D$ *is a mapping from* $O_D \cup \{\triangle\}$ *to* $\cup_{i\geq 0} O_D^i$; *If* $label_D(o) = \&$, *then* $children(o) \in O_D^1$;
4. *The structure that we obtain by considering only children of non-reference nodes (nodes with a label other than "&") is a tree.*

Example 1. The upper part of Figure 1 is a (partial) representation for the Jammers document from Section 1 and would correspond to the following structure. $D = \langle O_D, label_D, children_D\rangle$, where $O_D = \{o_1, o_2, \ldots\}$, $children(\triangle) = [\ldots, o_1, \ldots]$, and

$$\begin{array}{ll} label(o_1) = \text{jammer} & children(o_1) = [o_{11}; o_{12}; o_{13}; o_{14}] \\ label(o_{11}) = \text{company} & children(o_{11}) = [o_{111}] \\ label(o_{111}) = \text{“SESP”} & children(o_{111}) = [\ ] \\ \vdots & \vdots \end{array}$$

Type system. We adopt the type system of [2, 25], where predicates are used to describe labels and regular expressions are used to describe children. Note though that we do not handle unordered trees, and that we model references in a slightly different way. Also, we choose not to use XML Schema [33], which

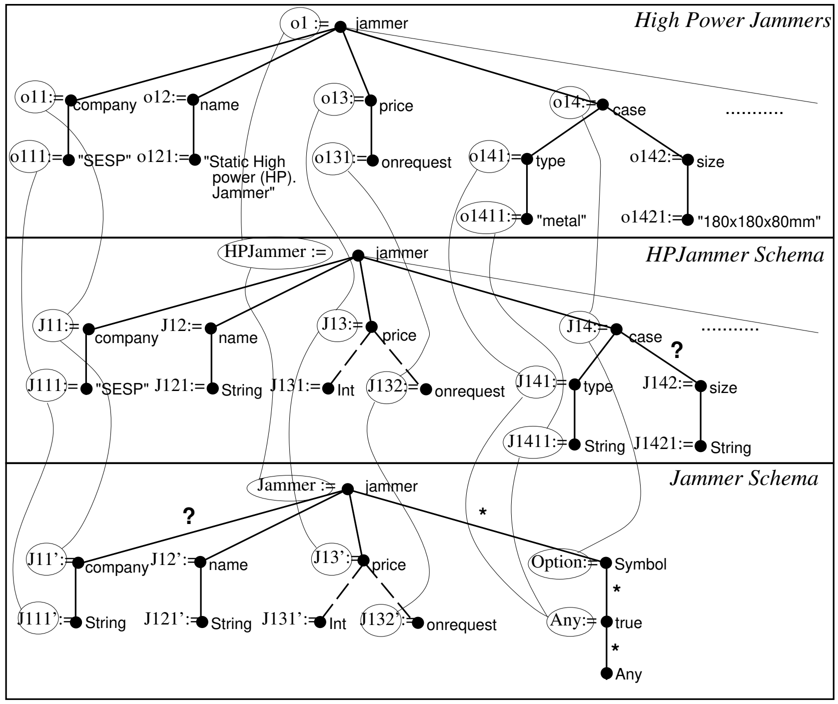

Fig. 1. Type assignment and subsumption mapping

is more a user syntax for types than a model, but we will explain later on how subsumption can be used in the context of XML Schema.

Let $\mathcal{T}$ be a fixed, infinite, set of *type names*, and $\mathcal{P}$ a fixed set of *label predicates*, which is closed under disjunction, conjunction, and complementation. We use τ, τ' etc., to denote elements of $\mathcal{T}$. Regular expressions over $\mathcal{T}$ are of the form ϵ, τ, $\&\tau$, (R_1, R_2), $(R_1 \mid R_2)$, or R_1^*, where R_1 and R_2 are regular expressions, and $\tau \in \mathcal{T}$. $L(R)$ denotes the language defined by the regular expression R, in which $\&\tau$ is treated as a single symbol.

Definition 2. *A* type schema *is a structure* $S = \langle T_S, predicate_S, regexp_S \rangle$, *in which*

1. T_S *is a finite subset of* $\mathcal{T}$;
2. $predicate_S$ *is a mapping from* T_S *to* $\mathcal{P}$ *with the property that for each* τ, *either* $predicate_S(\tau) = \{\&\}$, *or* $\& \notin predicate_S(\tau)$; *and*
3. $regexp_S$ *is a mapping from* $T_S \cup \{\triangle\}$ *to regular expressions over* T_S. *Whenever* $predicate_S(\tau) = \{\&\}$, $regexp_S(\tau)$ *must be of the form* $\tau_1 \mid \cdots \mid \tau_n$.

For convenience, we will sometimes describe schemas as $\tau \mapsto p; r$, where p and r are the predicate and regular expression corresponding to τ. We write $predicate(\tau) = \textbf{true}$ to mean that it is satisfied by all tags *except* "&" – the restrictions on the interaction between reference and non-reference types guarantee that this will never cause any confusion.

Example 2. The middle part of Figure 1 is a (partial) representation of the schema for HP jammers and would correspond to the structure $\langle T_{\text{cat}}, p_{\text{cat}}, r_{\text{cat}} \rangle$, where T_{cat} is the set $\{\mathbf{catalog}, \mathbf{HPjammer}, \mathbf{J_{11}}, \mathbf{J_{12}}, \mathbf{J_{13}}, \mathbf{J_{14}}, \mathbf{J_{111}}, \ldots, \mathbf{J_{1421}}\}$, $\mathit{regexp}(\triangle)$ is **catalog**, and

$$
\begin{array}{ll}
\mathit{predicate}_{\text{cat}}(\mathbf{HPjammer}) = \{\text{jammer}\} & \mathit{regexp}_{\text{cat}}(\mathbf{HPjammer}) = \mathbf{J_{11}}, \mathbf{J_{12}}, \mathbf{J_{13}}, \mathbf{J_{14}} \\
\vdots & \vdots \\
\mathit{predicate}_{\text{cat}}(\mathbf{J_{13}}) = \{\text{price}\} & \mathit{regexp}_{\text{cat}}(\mathbf{J_{13}}) = \mathbf{J_{131}} | \mathbf{J_{132}} \\
\mathit{predicate}_{\text{cat}}(\mathbf{J_{131}}) = \{0, 1, \ldots\} & \mathit{regexp}_{\text{cat}}(\mathbf{J_{131}}) = \epsilon \\
\mathit{predicate}_{\text{cat}}(\mathbf{J_{131}}) = \{\text{onrequest}\} & \mathit{regexp}_{\text{cat}}(\mathbf{J_{132}}) = \epsilon \\
\vdots & \vdots
\end{array}
$$

Typing and type assignment.

Definition 3. *Let D be a database and S a schema. We say D is of type S under the type assignment θ, and write $D :_\theta S$ iff θ is a function from $O_D \cup \{\triangle\}$ to $T_S \cup \{\triangle\}$ such that:*

1. $\theta(\triangle) = \triangle$,
2. *for each $o \in O_D$, $\mathit{predicate}_S(\theta(o)) \models \mathit{label}(o)$, and*
3. *for each $o \in O_D \cup \{\triangle\}$ with $\mathit{children}(o) = [o_1, \ldots, o_n]$, $\theta(o_1) \ldots \theta(o_n) \in L(\mathit{regexp}_S(\theta(o)))$.*

We say that D is of type S, and write $D : S$, iff $D :_\theta S$ for some θ. Models(S) is the set of databases of type S, i.e., $\{D \mid D : S\}$. It is immediate that $D :_\theta S$ and $D' \subseteq D$ (i.e., $O_{D'} \subseteq O_D$ and the corresponding labels and children are the same) imply $D' :_{\theta|O_{D'}} S$.

Example 3. Figure 1 illustrates the type assignment between the `Jammer` document and the `HPJammer` schema, corresponding to the following θ:

$$
\begin{array}{rr}
\theta(o_1) = \mathbf{HPjammer} & \theta(o_{11}) = \mathbf{J_{11}} \\
\theta(o_{13}) = \mathbf{J_{13}} & \theta(o_{131}) = \mathbf{J_{132}} \\
\theta(o_{14}) = \mathbf{J_{14}} & \theta(o_{141}) = \mathbf{J_{141}} \\
\vdots & \vdots
\end{array}
$$

Type assignment is the most important information coming out of the typing process (also called *validation* in the XML world). Once computed, it allows the system to efficiently obtain the type of a given data whenever needed, e.g., in order to chose the storage or take query processing decisions at run time. Note that type assignment information is logically provided in the XML Query data model [15] by the `Def_T` reference[2].

However simple, our type system is powerful enough to capture most of the other proposals, including XML Schema. It can be used to represent existing

[2] `http://www.w3.org/TR/query-datamodel/#def_t`

type information from heterogeneous sources [10, 32, 2] or to describe mixes of structured and semistructured data. The two following remarks will also play an important role in the rest of the paper.

Remark 1. `Any` is the schema that such that D : `Any` holds for any database D:

$$\begin{aligned} \triangle &\mapsto (\tau_{\text{anytype}} \mid \tau_{\text{anyref}})^* \\ \tau_{\text{anyref}} &\mapsto \{\&\}; (\tau_{\text{anyref}} \mid \tau_{\text{anytype}}) \\ \tau_{\text{anytype}} &\mapsto \textbf{true}; (\tau_{\text{anytype}} \mid \tau_{\text{anyref}})^* \end{aligned}$$

Remark 2. For each database D, one can define a schema S that types this database only, by taking T_S such that it contains exactly a type name τ for each object o in O_D, with $\theta(o) = \tau$, $predicate_S(\tau) = \{label_D(o)\}$ and $regexp_S(\tau) = children_D(o)$. Then, $D :_\theta S$ and $\text{Models}(S) = \{D\}$.

We will write $S_{[D]}$ the schema that types the database D only. We will call `None` the schema that types the empty database only. `None` has $T_{\texttt{None}} = \emptyset$ and $regexp_{\texttt{None}}(\triangle) = \epsilon$.

3 Subsumption

Intuitively, subsumption relies on a mapping between types (playing a role similar to type assignment for typing) and on inclusion between regular expressions over these types.

Definition 4. *Let S and S' be two schemas. We say that schema S* subsumes *S' under the* subsumption mapping *θ, and write $S \preceq_\theta S'$, iff θ is a function from $T_S \cup \{\triangle\}$ to $T_{S'} \cup \{\triangle\}$ such that:*

1. *$\theta(\tau) = \triangle$ iff $\tau = \triangle$.*
2. *For all $\tau \in T_S$, $predicate_S(\tau) \subseteq predicate_{S'}(\theta(\tau))$.*
3. *For all $\tau \in T_S \cup \{\triangle\}$, $\theta(L(regexp_S(\tau))) \subseteq L(regexp_{S'}(\theta(\tau)))$ (where θ is extended to words in the language in the natural way)*

We write $S \preceq S'$ if there exists a θ such that $S \preceq_\theta S$, and $S \approx S'$ for $(S \preceq S') \wedge (S' \preceq S)$: this is clearly an equivalence relation.

Example 4. Figure 1 illustrates the subsumption mapping between the `Jammer` and `HPJammer` types, corresponding to the following θ':

$$\begin{array}{rr} \theta'(\textbf{HPJammer})\!=\!\textbf{Jammer} & \theta'(\mathbf{J_{11}})\!=\!\mathbf{J'_{11}} \\ \theta'(\mathbf{J_{111}})\!=\!\mathbf{J_{111}} & \theta'(\mathbf{J_{13}})\!=\!\mathbf{J'_{13}} \\ \theta'(\mathbf{J_{14}})\!=\!\textbf{Option} & \theta'(\mathbf{J_{141}})\!=\!\textbf{Any}\ldots \end{array}$$

The following propositions cover the elementary properties of subsumption. The first states that type checking is a special case of subsumption, and is a direct consequence of Remark 2. The second and third propositions state the transitivity of subsumption, and more importantly of their underlying subsumption mapping, giving the means to propagate relationships between types.

Proposition 1. *Let S, S', S'' be three schemas, and D be a database.*

1. $D :_\theta S$ *iff* $S_{[D]} \preceq_\theta S$.
2. $S \preceq_{\theta_1} S'$ *and* $S' \preceq_{\theta_2} S''$ *imply* $S \preceq_{\theta_1 \circ \theta_2} S''$.
3. *If* $D :_{\theta_1} S$ *and* $S \preceq_{\theta_2} S'$, *then* $D :_{\theta_1 \circ \theta_2} S'$.

Using subsumption for validation. An important consequence of Prop. 1 is the ability to take advantage of subsumption for computing type assignments. Intuitively, if one has a type assignment for a given database, and a subsumption mapping from the original type to the new type, the new type assignment can be obtained by composing the mappings rather than by evaluating the type assignment from scratch.

This is especially useful as in most practical scenarios, including the one we sketched in Section 1, XML data is generated from a legacy source, along with its original schema (`SESPCatalog`). If instead of checking inclusion, company "A" computes subsumption between the two schemas, it obtains the new type-assignment at the same time. This approach has a number of advantages. First, the size of schema is orders of magnitude smaller than the data. Secondly, this can be done at compile time, without requiring to access the whole data.

Example 5. For instance, assume Company "A" runs a query to the SESP store that returns the jammer o_1. We know from θ in Example 2 that o_1 has type **HPJammer** and from θ' in Example 4, that **HPJammer**s correspond to **Jammer**s in the integrated schema. This gives us directly that the type of o_1 with respect to the integrated schema is **Jammer** (see also Figure 1).

4 Comparison with inclusion, extension, et al

To get a better understanding of the scope of subsumption, we now compare it to other relations over types, notably, inclusion, XML schema's mechanisms of refinement and extension, subtyping, and the instantiation mechanism of [10].

Inclusion. Type inclusion is defined in terms of containment of models.

Definition 5. $S \subseteq S'$ *iff* $\text{Models}(S) \subseteq \text{Models}(S')$.

Of course, subsumption provides additional information compared to inclusion because of the subsumption mapping. A natural question is: can one always find a subsumption mapping between two types for which inclusion holds.

Proposition 2. *Let S and S' be two schemas. Then (1) $S \preceq S' \rightarrow S \subseteq S'$, but not conversely; and (2) $S \preceq S' \rightarrow S \subseteq S'$, and this implication is proper.*

Proof. (2) is trivial. (1) and (3) are direct consequences of Remark 2. To see why the implications are proper, consider the following type schemas:

$$\begin{array}{rl} S,\, S' & \tau_1 \mapsto \{a\}; \epsilon \\ & \tau_2 \mapsto \{a\}; \epsilon \\ S & \triangle \mapsto \tau_1^*, \tau_2 \\ S' & \triangle \mapsto \tau_1, \tau_2^* \end{array}$$

Then both S and S' type precisely those databases for which *children*$(\triangle)$ are all leaves with tag "a", but neither $S \preceq S'$ or $S' \preceq S$.

As shown in [20, 21], type inclusion can be used to type-check XML languages. Proposition 2 implies that some queries might type-check even though a subsumption mapping does not exist. In such a case one might not be able to take advantage of subsumption. Fortunately, we will see that there are many practical cases for which a subsumption mapping between types exists, including: when they are defined through XML Schema's refinement or extension mechanisms or when they are exported from a traditional type system with subtyping. Moreover, we will show (Proposition 1) that if $S'' \subseteq S$, then one can construct a schema S' equivalent to S for which $S'' \preceq S'$.

Extension and refinement in XML Schema. XML Schema: Part 1 [33] defines two subtyping-like mechanisms, called *extension* and *refinement*, aimed at reusing types. For obvious space limitations, we cannot explain all the complex features of XML Schema, so our presentation will rely on a simple modeling of these two mechanisms. In a nutshell, *extension* allows to add new fields at the end of a given type, while *refinement* provides syntactic means to restrict the domain of a given type.

Example 6. The following XML Schema declaration defines a `Stated-Address` by *refining* an `Address` to always have a unique `state` element and `US-Address` by extending `Stated-Address` with a new `zip` element.

```
<complexType name="Address">
  <element name="street" type="string"/>
  <element name="city"   type="string"/>
  <element name="state"  type="string" minOccurs="0" maxOccurs="1"/>
</complexType>

<complexType name="Stated-Address" base="Address"
                                          derivedBy="refinement">
  <element name="street" type="string"/>
  <element name="city"   type="string"/>
  <element name="state"  type="string" minOccurs="1" maxOccurs="1">
</complexType>

<complexType name="US-Address" base="Stated-Address"
                                           derivedBy="extension">
 <element name="zip"   type="positiveInteger"/>
</complexType>
```

In our model, these three types would be defined as follows:

$$\begin{aligned}&\mathit{regexp}(\textbf{Address}) = \textbf{Street}, \textbf{City}, \textbf{State}?, \tau_{\text{anytype}}*\\&\mathit{regexp}(\textbf{Stated-Address}) = \textbf{Street}, \textbf{City}, \textbf{State}, \tau_{\text{anytype}}*\\&\mathit{regexp}(\textbf{US-Address}) = \textbf{Street}, \textbf{City}, \textbf{State}, \textbf{Zip}, \tau_{\text{anytype}}*\end{aligned}$$

The type τ_{anytype}, as defined in Remark 1, indicates the ability to have additional fields. Note that the subsumption relationship holds **US-Address** $\preceq$ **Stated-Address** $\preceq$ **Address**.

Proposition 3. *A type τ' derived by extension or refinement from a type τ is such that $\tau' \preceq \tau$.*

Proof Sketch: Refinement corresponds to adding a field at the end of a given type. This corresponds to regular expressions of the form: $regexp = \tau_1, \ldots, \tau_n, \tau_{\text{anytype}}*$, and $regexp' = \tau_1, \ldots, \tau_n, \tau_{n+1}, \tau_{\text{anytype}}*$ for which subsumption holds with $\theta(\tau_{n+1}) = \tau_{\text{anytype}}$.

Extension can be obtained by restricting a datatype, which yields inclusion between predicates. `minOccur` and `maxOccur` restrictions corresponds to regular expressions of the form:

$$regexp = (\underbrace{\tau, \tau, \ldots, \tau}_{n}), \underbrace{\tau?, \ldots, \tau?}_{m} \quad \text{and} \quad regexp' = (\underbrace{\tau, \tau, \ldots, \tau}_{n'}), \underbrace{\tau?, \ldots, \tau?}_{m'}$$

Subsumption holds when $n \leq n'$ and $(n+m) \geq (n'+m')$. Union type restrictions correspond to regular expressions of the form $regexp = \tau_1 | \ldots | \tau_n | \ldots | \tau_{n+m}$, and $regexp' = \tau_1 | \ldots | \tau_n$ for which subsumption holds. The result follows by induction.

Subtyping. The literature proposes a large number of different mechanisms called or related to subtyping [8, 27, 29]. Basic subtyping usually relies on two mechanisms: additions of new attributes in tuples (e.g., `{ name: String; age: Int } <: { name: String }`) and restrictions on atomic types (e.g., `Int <: Float`). The last mechanism is captured by predicate restrictions in our context, while the first is captured by adding `Any`* types when modeling tuples[3].

Instantiation. [10] proposes a notion of *instantiation* that corresponds to certain restrictions over types. This mechanism allows: restrictions on the label predicates, restrictions on the arity of collections (similar to the `minOccur` and `maxOccur` restrictions in XML schema), and restrictions on the unions. As for XML Schema, these restrictions yields only types for which subsumption holds.

5 Greatest Lower and Least Upper Bound

Let S and S' be two schemas. We consider equivalence classes of schemas with respect to subsumption $[S]_{\approx}$, ordered by $\preceq$, and show that this is a lattice. We first define the greatest lower bound, which intuitively is a schema describing the type information that is common to the given schemas.

We shall assume that whenever τ and τ' are in $\mathcal{T}$, so is the symbol $\tau \sqcap \tau'$. We need to define appropriately intersection of regular expressions: our regular expressions are over type names, but the intersection should be over the *semantics* of the types, not the names. For example, if the regular expressions are τ_1^* and (τ_2, τ_3), the intersection will be $((\tau_1 \sqcap \tau_2), (\tau_1 \sqcap \tau_3))$.

[3] Note however that our type system does not capture the unordered semantics of tuples.

Definition 6. *Let S and S' be two type schemas.*[4] *The* greatest lower bound $S \sqcap S'$ *and* least upper bound $S \sqcup S'$ *are the schemas with* $T_{S \sqcap S'} = \{\tau \sqcap \tau' \mid \tau \in T_S, \tau' \in T_{S'}\}$, $T_{S \sqcup S'} = T_S \cup T_{S'}$, *and*

$$\begin{array}{lll} S \sqcap S': & \triangle \mapsto \mathit{regexp}_S(\triangle) \cap \mathit{regexp}_{S'}(\triangle) & \\ & \tau \mapsto \mathit{predicate}_S(\tau); \mathit{regexp}_S(\tau) \cap \mathit{regexp}_{S'}(\tau') & \\ & & \\ S \sqcup S': & \triangle \mapsto \mathit{regexp}_{S'}(\triangle) | \mathit{regexp}_{S'}(\triangle) & \\ & \tau \mapsto \mathit{predicate}_S(\tau); \mathit{regexp}_S(\tau) & \tau \in T_S \\ & \tau \mapsto \mathit{predicate}_{S'}(\tau); \mathit{regexp}_{S'}(\tau) & \tau \in T_{S'} \end{array}$$

Example 7. Consider the following two schemas (where τ_{anytype} is as in the definition of the schema `Any`).

$$\begin{array}{llll} S: & \triangle \mapsto (\tau_1, \tau_{\text{anytype}}*) & S': & \triangle \mapsto (\tau_{\text{anytype}}*, \tau_2) \\ & \tau_1 \mapsto \{a\}; \epsilon & & \tau_2 \mapsto \{b\}; \epsilon \end{array}$$

$$\begin{array}{ll} S \sqcap S': & \triangle \mapsto ((\tau_1 \sqcap \tau_{\text{anytype}}), (\tau_{\text{anytype}} \sqcap \tau_{\text{anytype}})*, (\tau_{\text{anytype}} \sqcap \tau_2)) \\ & \tau_1 \sqcap \tau_{\text{anytype}} \mapsto \{a\}; \epsilon \\ & \tau_{\text{anytype}} \sqcap \tau_2 \mapsto \{b\}; \epsilon \end{array}$$

where $\tau_{\text{anytype}} \sqcap \tau_{\text{anytype}}$ is the same as τ_{anytype} up to renaming.

The greatest lower bound of schemas requires intersection of regular expressions, that can lead to a blowup in the size of the schema but this is unlikely to happen in practice.

The greatest lower bound is the best description, with respect to subsumption, of all of the type information that we have about both schemas. In particular, if a database is typed by both S and S', it is also typed by $S \sqcap S'$. More generally:

Proposition 4. *1.* $S \sqcap S' \preceq S$ *and* $S \sqcap S' \preceq S'$; $S \preceq S \sqcup S'$ *and* $S' \preceq S \sqcup S'$.
2. If $S'' \preceq S$ *and* $S'' \preceq S'$, *then* $S'' \preceq S \sqcap S'$; *similarly If* $S \preceq S''$ *and* $S' \preceq S''$, *then* $S \sqcup S' \preceq S''$.
3. If $D : S$ *and* $D : S'$, *then* $D : S \sqcap S'$ *and* $D : S \sqcup S'$.

Theorem 1. $\mathcal{L} = \left\langle [\mathcal{S}]_{\approx}, \sqcap_{\approx}, \sqcup_{\approx}, [\texttt{None}]_{\approx}, [\texttt{Any}]_{\approx} \right\rangle$ *is an incomplete distributive lattice without complement.*

The next theorem is essential as it gives a relationship between the syntactic definitions of $S \sqcap S'$ and $S \sqcup S'$ and the semantics of the respective schemas. The proof of this theorem relies on Remark 2, that connects typing, on which Models are defined, and subsumption.

Theorem 2. *For any schemas S and S', (1)* $\text{Models}(S \sqcap S') = \text{Models}(S) \cap \text{Models}(S')$ *and (2)* $\text{Models}(S \sqcup S') = \text{Models}(S) \cup \text{Models}(S')$.

[4] We assume for simplicity that T_S and $T_{S'}$ are disjoint. This can always be achieved by appropriate renaming.

The use of untagged roots was introduced in [2]. Our results give another, technical, reason why such special treatment of the root is needed. Specifically, if the database root were allowed to be tagged, then $\mathcal{L}$ would not be distributive. On the other hand, a data model based on forests rather than trees would not work either, as then $\mathrm{Models}(S \sqcup S') = \mathrm{Models}(S) \cup \mathrm{Models}(S')$ would not hold.

Subsumption is weaker than inclusion, as there are schemas that are contained in other schemas without subsuming them. For this reason, the following Corollary is very important: it shows that whenever a schema S is contained in a schema S', S can be rewritten in an equivalent way such that S subsumes S'.

Corollary 1. *Let S and S' be two schemas such that $\mathrm{Models}(S) \subseteq \mathrm{Models}(S')$. Then there exists a schema S'' such that (1) $\mathrm{Models}(S'') = \mathrm{Models}(S)$ and (2) $S'' \preceq S'$.*

6 Practical use of subsumption

We now come back to our example from the introduction and illustrate how subsumption can be helpful for storage and query processing.

Standard relational techniques are used to design storage structures that take into account which queries are likely to be asked. If we take query `q` from the introduction, one might wish to find a schema S that would allow to store data in such a way this query is answered in an efficient way. However, if one only considers the integrated schema, one can only use the available information about `Jammer`s. Existing techniques [14, 18, 31] would provide the following relational schema:

```
jammers(jid, company, name, price);
options(jid,att,treeid);
tree(treeid,...);
```

where the `tree` table is used to store any tree, playing a similar role to the overflow graph in [14].

The greatest lower bound can be used to derive a schema that includes the `warranty` attribute. After appropriate renaming of types, this is:

```
Warranty_Jammer :=
      jammer [ ?Company', Name', Price',
             *(  WarrantyOption' | (OtherOption', ?Supplement') ) ];
Company'        := company [ String ];
Name'           := name[ String ];
Price'          := price [ Int | onrequest ];
WarrantyOption' := warranty * Any;
OtherOption'    := !warranty * Any;
Supplement'     := supplement [ Int ];
```

We can then use this information to store the data with a faster access to the `warranty` attribute, using the following relational schema:

```
jammers(jid, company, name, price);
jammers(jid, warranty);
options(jid,att,supplement,treeid);
tree(treeid,...);
```

We then need to evaluate query q on top of this storage. The key remark is that YAT_L [10, 11] uses pattern matching with type expressions. This captures the navigation performed in other languages [1, 13].

Following [9], the `match` clause of a YAT_L query is represented by a pattern-matching operation called *Bind*. *Bind* matches a regular expression with the data, and returns a binding between variables in the query and values in the document. In the case of query q, *Bind* $_{\texttt{p[\$n,\$w]}}$ where

```
p[$n,$w] := products * Jammer;
Jammer   := jammer [ *(Name | Warranty | Other) ];
Name     := name * ($n:Any1);
Warranty := warranty * ($w:Any2);
Other    := !name!warranty * Any;
Any1     := true [ Any* ]; Any2 := true [ Any* ]
```

Most XML processors evaluate similar operations by loading the document in memory and parsing it according to the given filter. This can be expensive and does not make use of the knowledge of how the document is stored (here with using the relational schema above).

Let θ be the subsumption mapping from the type of `p[$n,$w]` to the greatest lower bound:

$$\theta'(\texttt{Warranty\_Jammer}){=}\texttt{Jammer} \qquad \theta'(\texttt{Company}'){=}\texttt{Other}$$
$$\theta'(\texttt{Name}'){=}\texttt{Name} \qquad \theta'(\texttt{Price}'){=}\texttt{Other}$$
$$\theta'(\texttt{WarrantyOption}'){=}\texttt{Warranty} \qquad \theta'(\texttt{OtherOption}'){=}\texttt{Other}\ldots$$

Through θ, we know that the values of `$n`
are the values of the elements of type `Name'` in the the stored schema, hence how to access them using the relational engine.

7 Related work and conclusion

Typing for XML is a heavily studied problem. Existing work covers the type systems themselves [2, 10, 12, 33], type checking [20, 26] and type inference [25, 28]. XML types have been used for query formulation [19], query optimization [17, 9], storage [14, 31], and compile-time error detection [20]. A notion of subsumption for unordered semistructured data was proposed in [6] based on a graph bisimulation. Our work extends this approach to types that involve order and regular expressions. Typing in XDuce [20] relies on full type inclusion. [7] describes a notion of containment between XML DTDs, which are less expressive than our type system and is based on full inclusion with tag renaming.

There are many directions in which this work can be continued. First of all, while our work (and most other work in this area), uses a list model for data, for database applications a set semantics may be more appropriate, and therefore extending the results to sets (and bags) would be of interest. For applying the results to inheritance, as indicated above, one may want to be able to type an object in multiple ways – formally this may be captured by the greatest lower bound, but this does not provide the intuitive semantics desired here.

We have not discussed complexity in this paper. Typing a database is a special case of subsumption (where the database is itself the schema), and the complexity of typing is known [2] to be hard. Note, however, that complexity of checking subsumption is in the size of the schema rather than in the size the database. Furthermore, many of the problems that relate to typing become tractable in the case of *unambiguous* schemas: in our framework there are many possible definitions of ambiguity, such as the existence of a single typing, unambiguity up to reference nodes, unambiguous regular expressions, etc. Efficient evaluation of queries is one of the main motivations for this work. Many complex parameters must be taken into account in this context, such as the impact of storage structures, memory management issues, etc. To evaluate the real impact of subsumption, we consider an implementation of the techniques presented here in the context of the YAT System [10, 9].

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, Apr. 1997.
2. C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of International Conference on Database Theory (ICDT)*, Lecture Notes in Computer Science, Jerusalem, Israel, Jan. 1999.
3. R. Bourret, J. Cowan, I. Macherius, and S. St. Laurent. Document definition markup language (ddml) specification, version 1.0, Jan. 1999. W3C Note.
4. T. Bray, C. Frankston, and A. Malhotra. Document content description for XML. Submission to the World Wide Web Consortium, July 1998.
5. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation, Feb. 1998. `http://www.w3.org/TR/REC-xml/`.
6. P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of International Conference on Database Theory (ICDT)*, volume 1186 of *LNCS*, pages 336–350, Delphi, Greece, Jan. 1997.
7. D. Calvanese, G. D. Giacomo, and M. Lenzerini. Representing and reasoning on xml documents: A description logic approach. *Journal of Logic and Computation*, 9(3):205–318, 1999.
8. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.
9. V. Christophides, S. Cluet, and J. Siméon. On wrapping query languages and efficient XML integration. In SIGMOD'2000, Dallas, Texas, May 2000.
10. S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In SIGMOD'1998, pages 177–188, Seattle, Washington, June 1998.
11. S. Cluet and J. Siméon. YAT_L: a functional and declarative language for XML. Draft manuscript, Mar. 2000.
12. A. Davidson, M. Fuchs, M. Hedin, M. Jain, J. Koistinen, C. Lloyd, M. Maloney, and K. Schwarzhof. Schema for object-oriented XML 2.0, July 1999. W3C Note.
13. A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. A query language for XML. In *Proceedings of International World Wide Web Conference*, Toronto, May 1999.

14. A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In SIGMOD'1999, pages 431–442, Philadelphia, Pennsylvania, June 1999.
15. M. F. Fernandez and J. Robie. XML Query data model. W3C Working Draft, May 2000. `http://www.w3.org/TR/query-datamodel/`.
16. M. F. Fernandez, J. Siméon, and P. Wadler (editors). XML query languages: Experiences and exemplars. draft manuscript, communication to the W3C, Sept. 1999.
17. M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In ICDE'1998, Orlando, Florida, Feb. 1998.
18. M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In SIGMOD'2000, pages 165–176, Dallas, Texas, May 2000.
19. R. Goldman and J. Widom. Data guides: Enabling query formulation and optimization in semistructured databases. In VLDB'1997, pages 436–445, Athens, Greece, Aug. 1997.
20. H. Hosoya and B. C. Pierce. XDuce: an XML processing language. In *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 2000.
21. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. Submitted for publication, Mar. 2000.
22. `http://sesp.co.uk/4.htm`.
23. N. Klarlund, A. Moller, and M. I. Schwartzbach. DSD: A schema language for XML. In *Workshop on Formal Methods in Software Practice*, Portland, Oregon, Aug. 2000.
24. M. Murata. Tutorial: How to relax. `http://www.xml.gr.jp/relax/`.
25. T. Milo and D. Suciu. Type inference for queries on semistructured data. In PODS'1999, pages 215–226, Philadephia, Pennsylvania, May 1999.
26. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In PODS'2000, Dallas, Texas, May 2000.
27. J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
28. Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In PODS'2000, Dallas, Texas, May 2000.
29. F. Pottier. *Synthèse de types en présence de sous-typage: de la théorie à la pratique*. Thèse de doctorat, Université Paris VII, July 1998. `http://pauillac.inria.fr/~fpottier/publis/these-fpottier.ps.gz`.
30. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.
31. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of International Conference on Very Large Databases (VLDB)*, Edinburgh, Scotland, Sept. 1999.
32. J. Siméon and S. Cluet. Using YAT to build a web server. In *International Workshop on the Web and Databases (WebDB'98)*, volume 1590 of *LNCS*, pages 118–135, Valencia, Spain, Mar. 1998.
33. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. W3C Working Draft, Feb. 2000.