

Boolean Operations and Inclusion Test for Attribute-Element Constraints (Extended Abstract)

Haruo Hosoya*

Research Institute for Mathematical Sciences, Kyoto University
hahosoya@kurims.kyoto-u.ac.jp

Makoto Murata

IBM Tokyo Research Laboratory
mmurata@trl.ibm.co.jp

Abstract

The history of schema languages for XML is an increase of expressiveness. While early schema languages mainly focused on the *element* structure, Clark first paid an equal attention to *attributes* by allowing both element and attribute constraints in a single regular expression. In this paper, we investigate an algorithmic aspect of Clark’s mechanism (called “attribute-element constraints”), namely, intersection and difference operations and inclusion test, which have been proved to be important in static typechecking for XML processing programs. The contributions here are (1) proofs of closure under intersection and difference and decidability of inclusion test and (2) algorithm formulations incorporating a “divide-and-conquer” strategy for avoiding an exponential blow-up for typical inputs.

1 Introduction

XML [3] is a standard document format that comes with two notions: *data* and *schemas*. Data are tree structures and basically have two major constituents, *elements*, which are tree nodes forming the “skeleton” of documents, and *attributes*, which are auxiliary name-value pairs associated with each element. Schemas are a mechanism for imposing constraints on these two structures, and what kinds of constraints they can use is defined by a *schema language*.

Numerous schema languages have been proposed and their history has been basically how to increase expressiveness, allowing finer- and finer-grained controls to the structure of documents. At first, the main interest of the designers of schema languages (DTD [3], W3C XML Schema [10], and RELAX [24]) was *elements*, and by now this pursue has mostly converged—most schema languages use regular expressions for describing this part of data. On the other hand, the treatments of *attributes* have been rather simplistic. For example, DTD allows us to specify each individual attribute to be either required or optional. However, there has been a big demand for a higher expressiveness, in particular, involving interdependency among elements and attributes, e.g., “a *person* element must have either a **name** subelement or a **name** attribute, not both.” Such a

*Oiwake-cho, Kitashirakawa, Sakyo-ku, Kyoto 606-8502, Japan. Tel: 075-753-7281. Fax: 075-753-7276.

constraint either cannot be expressed or requires an exponentially large description in early schema languages.

Recently, James Clark, in his schema language TREX [6], proposed a description mechanism called *attribute-element constraints* for overcoming this shortcoming. The kernel of his proposal is to allow mixture of constraints on elements and those on attributes in a single “regular” expression, thus achieving a uniform and symmetric treatment of two kinds of structure. The expressive power yielded by this mechanism is quite substantial—in particular, the above-mentioned attribute-element interdependency can be represented in a simple and straightforward manner.

In this paper, we pay attention to an algorithmic aspect of attribute-element constraints. Specifically, we investigate intersection and difference operations (i.e., compute a new schema representing the intersection of or difference between given two schemas) and inclusion test (i.e., check if any instance of one schema is that of another). Studying these has not only been a tradition in formal language theory, but also is strongly motivated by an important application: static typechecking for XML processing—program analysis technology for detecting all run-time type errors at compile time. Indeed, the recent series of papers have used these operations in crucial parts of their typechecking algorithms [17, 11, 25, 21, 23]. For example, intersection is used in almost all of these languages as the core of type inference mechanisms for their pattern matching facilities; difference is used in XDuce type inference for precisely treating data flow through prioritized pattern clauses [18]; and, perhaps most importantly, inclusion test is used for “subtyping” in many of the above-mentioned languages.

The main contributions that we made in this work are twofold: (1) proofs of closure under intersection and difference and decidability of inclusion and (2) algorithms based on a “divide-and-conquer” technique.

While closure under intersection can be obtained in a straightforward way, that under difference is slightly tricky. We first fail to prove closure under difference due to attributes’ special properties (i.e., ordering among attributes is insignificant and the same label cannot occur in the same set, whereas elements are opposite in both respects). However, we discover that we can regain the closure by imposing a few simple syntactic restrictions. Although these bring a small “bump” in the system, the compromise in expressiveness seems acceptable for practical purposes as we will discuss in Section 3.3.1. These arguments may appear to imply that inclusion test also needs the same restrictions since inclusion test is usually done by computing a difference and then testing the emptiness. On the contrary, they are not needed since our inclusion algorithm computes, rather than an exact difference, an “approximate” difference whose emptiness is exactly the same as the difference. One practical implication from this is that, for applications that use only intersection and inclusion (not difference), the above-mentioned syntactic restrictions can be elided, which may help keeping cleanness of the applications.

We have further made efforts to make the algorithms for intersection, difference, and inclusion more practical. We can think of a naive decision procedure that completely separates constraints on attributes and those on elements from their mixture. However, this easily blows up even for typical inputs as explained in Section 3. Although this explosion seems not avoidable in the worst case, we can make the algorithm efficient in most practical cases by partitioning given constraint formulas into orthogonal subformulas and proceeding to the corresponding subformulas separately. The basic idea is taken from Vouillon’s inclusion algorithm for shuffle expressions [26], but has not been applied to the setting of attribute-element constraints. Our specific contributions here are thus the conditions to apply partitioning and the proofs of correctness.

This work has been carried out in collaboration with committee members of the new schema language RELAX NG [8]. As a result, aiming for closure under booleans, RELAX NG adopted the

syntactic restrictions described in this paper. Boolean algorithms for attribute-element constraints have been presented in an informal workshop paper by the present authors [16]. The addition in this paper is the full treatment of “single-attribute expressions” (defined in Section 2) and the inclusion algorithm.

2 Attribute-element constraints

In our framework, data are XML documents with the restriction that only elements and attributes can appear. That is, we consider trees where each node is given a name and, in addition, associated with a set of name-string pairs. In XML jargon, a node is called element and a name-string pair is called attribute. For example, the following is an element with name `article` that has two attributes `key` and `year` and contains four child elements—two with `authors`, one with `title`, and one with `publisher`.

```
<article key="HosoyaMurata2002" year="2002">
  <author> ... </author>
  <author> ... </author>
  <title> ... </title>
  <publisher> ... </publisher>
</article>
```

The ordering among sibling elements is significant, whereas that among attributes is not. The same name of elements can occur multiple times in the same sequence, whereas this is disallowed for attributes.

Attribute-element constraints describe a pair of an element sequence and an attribute set. Let us illustrate the constraint mechanism. Element expressions describe constraints on elements and are regular expressions on names. For example, we can write

$$\text{author}^+ \text{title publisher}^?$$

to represent that a permitted sequence of child elements are one or more `author` elements followed by a mandatory `title` element and an optional `publisher` element. (Note that the explanation here is informal: for brevity, we show constraints only on names of attributes and elements. The actual constraint mechanism formalized later can also describe contents of attributes and elements.) Attribute expressions are constraints on attributes and have a notation similar to regular expressions. For example,

$$\text{@key @year}^?$$

requires a `key` attribute and optionally allows a `year` attribute. (We prepend an at-sign to each attribute name in order to distinguish attribute names from element names.) A more complex example would be:

$$\text{@key} ((\text{@year @month}^?) | \text{@date})$$

That is, we may optionally append a `month` to a `year`; or we can replace these two attributes with a `date` attribute.

Attribute expressions are different from usual regular expressions in three ways. First, attribute expressions describe (unordered) sets and therefore concatenation is commutative. Second, since names cannot be duplicated in the same set, we require expressions to permit only data that

conform to this restriction (e.g., $(@a|@b)@a?$ is forbidden). Third, for the same reason, repetition (+) is disallowed in attribute expressions. We provide, however, “wild-card” expressions that allow an arbitrary number of arbitrary attributes from a given set of names (discussed later).

Attribute-element expressions or compound expressions allow one expression to mix both attribute expressions and element expressions. For example, we can write

$$@key @year? author^+ title publisher?$$

to require both that the attributes satisfy $@key @year?$ and that the elements satisfy $author^+ title publisher?$. The next example is a compound expression allowing either a `key` attribute or a `key` element, not both.

$$(@key | key) @year? author^+ title publisher?$$

In this way, we can express constraints where some attributes are interdependent with some elements. (Note that we can place attribute expressions anywhere—even after element expressions.) In the extreme, the last example could be made more flexible as follows

$$\begin{aligned} &(@key | key) \\ &(@year | year)? \\ &(@author | author^+) \\ &(@title | title) \\ &(@publisher | publisher)? \end{aligned}$$

where every piece of information can be put in an attribute or an element.

In addition to the above, we provide “multi-attribute expressions,” which allow an arbitrary number of attributes with arbitrary names chosen from a given set of names. Multi-attribute expressions are useful in making a schema “open” so that users can put their own pieces of information in unused attributes. For example, when we want to require `key` and `year` attributes but optionally permit any other attributes, we can write the following expression (where $(*\backslash key \backslash year)$ represents the set of all names except `key` and `year`).

$$@key @year @(*\backslash key \backslash year)^*$$

Although our formulation will not include a direct treatment, multi-attribute expressions can be even more useful if combined with name spaces. (Name spaces are a prefixing mechanism for names in XML documents; see [2] for the details.) For example, when we are designing a schema in the name space `myns`, we can write the following to permit any attributes in difference name spaces (where `myns:*` means “any names in name space `myns`”).

$$@myns:key @myns:year @(*\backslash (myns:*))^*$$

Apart from the kinds of name sets described above, the following can be useful: (1) the set of all names, (2) the set of all names in a specific name space, and (3) the set of all names except those from some specific name spaces. (In fact, these are exactly the ones supported by RELAX NG [8].)

2.1 Data model

We assume a countably infinite set \mathcal{N} of *names*, ranged over by a, b, \dots . We define *values* inductively as follows: a *value* v is a pair $\langle \alpha, \beta \rangle$ where

- α is a set of pairs of a name and a value, and
- β is a sequence of pairs of a name and a value.

A pair in α and a pair in β are called *attribute* and *element*, respectively. In the formalization, attributes associate names with values and therefore may contain elements. This may appear odd since XML allows only strings to be contained in attributes. Our treatment is just for avoiding the need to introduce another syntactic category for attribute contents and thereby simplifying the formalism. We write ϵ for an empty sequence and $\beta_1\beta_2$ for the concatenation of sequences β_1 and β_2 . For convenience, we define several notations for values.

$$\begin{aligned}
a[v] &\equiv \langle \emptyset, \langle a, v \rangle \rangle \\
@a[v] &\equiv \langle \{ \langle a, v \rangle \}, \epsilon \rangle \\
\langle \alpha_1, \beta_1 \rangle \langle \alpha_2, \beta_2 \rangle &\equiv \langle \alpha_1 \cup \alpha_2, \beta_1\beta_2 \rangle \\
\epsilon &\equiv \langle \emptyset, \epsilon \rangle
\end{aligned}$$

For example, $@a[v]@b[w]c[u]$ means $\langle \{ \langle a, v \rangle, \langle b, w \rangle \}, \langle c, u \rangle \rangle$. We write \mathcal{V} for the set of all values.

2.2 Expressions

Let \mathcal{S} be a set of sets of names where \mathcal{S} is closed under boolean operations. In addition, we assume that \mathcal{S} contains at least the set \mathcal{N} of all names and the empty set \emptyset . Each member N of \mathcal{S} is called *name set*.

We next define the syntax of expressions for attribute-element constraints. As already mentioned, our expressions describe not only top-level names of elements and attributes but also their contents. Since, moreover, we want expressions to describe arbitrary depths of trees, we introduce recursive definitions of expressions, that is, grammars.

We assume a countably infinite set of *variables*, ranged over by x, y, z . We use X, Y, Z for sets of variables. A *grammar* G on X is a finite mapping from X to compound expressions. *Compound expressions* c are defined by the following syntax in conjunction with *element expressions* e .

$$\begin{array}{ll}
c ::= @N[x]^+ & e ::= N[x] \\
& @N[x] & \epsilon \\
& c|c & e|e \\
& cc & ee \\
& e & e^+
\end{array}$$

We call the form $@N[x]^+$ *multi-attribute* expression (as mentioned) and $@N[x]$ *single-attribute* expression. (As we discuss in Section 3.3.1, we can encode single-attribute expressions by multi-attribute expressions in the case of finite name sets, and this seems to be sufficient in practical uses.) We define $\mathbf{FV}(c)$ as the set of variables appearing in c and $\mathbf{FV}(G)$ as $\bigcup_{x \in \text{dom}(G)} F(x)$. We require any grammar to be “self-contained,” i.e., $\mathbf{FV}(G) \subseteq \text{dom}(G)$, where $\text{dom}(G)$ is the domain of G . In the sequel, we use the following shorthands.

$$\begin{array}{llll}
@a[x]^+ &\equiv & @\{a\}[x]^+ & @a[x] &\equiv & @\{a\}[x] \\
a[x] &\equiv & \{a\}[x] & c^* &\equiv & c^+ | \epsilon \\
c? &\equiv & c | \epsilon & @N[x]^* &\equiv & @N[x]^+ | \epsilon
\end{array}$$

We forbid concatenation of expressions with overlapping attribute name sets. That is, we first define $\mathbf{att}(c)$ as the union of all the attribute name sets (the N in the form $@N[x]^+$ or $@N[x]$)

appearing in the expression c . Then, any expression must not contain an expression $c_1 c_2$ with $\mathbf{att}(c_1) \cap \mathbf{att}(c_2) \neq \emptyset$. We define $\mathbf{elm}(c)$ as the union of all the element name sets (the N in the form $N[x]$) appearing in the expression c .

2.3 Semantics

The semantics of expressions with respect to a grammar is described by the relation of the form $G \vdash v \in c$, which is read “value v conforms to expression c under G .” This relation is inductively defined by the following rules.

$$\frac{\forall i. (a_i \in N \quad G \vdash v_i \in G(x)) \quad k \geq 1 \quad a_i \neq a_j \text{ for } i \neq j}{G \vdash @_{a_1}[v_1] \dots @_{a_k}[v_k] \in @N[x]^+} \quad (\text{T-ATTREP})$$

$$\frac{a \in N \quad G \vdash v \in G(x)}{G \vdash @a[v] \in @N[x]} \quad (\text{T-ATT})$$

$$\frac{a \in N \quad G \vdash v \in G(x)}{G \vdash a[v] \in N[x]} \quad (\text{T-ELM})$$

$$\frac{G \vdash v \in c_1 \quad \text{or} \quad G \vdash v \in c_2}{G \vdash v \in c_1 | c_2} \quad (\text{T-ALT})$$

$$\frac{G \vdash v_1 \in c_1 \quad G \vdash v_2 \in c_2}{G \vdash v_1 v_2 \in c_1 c_2} \quad (\text{T-CAT})$$

$$G \vdash \epsilon \in \epsilon \quad (\text{T-EPS})$$

$$\frac{\forall i. G \vdash v_i \in c \quad k \geq 1}{G \vdash v_1 \dots v_k \in c^+} \quad (\text{T-PLU})$$

Note that rules T-ALT and T-CAT treat alternation and concatenation both in compound expressions and element expressions.

3 Boolean and Inclusion Algorithms

In this section, we present our algorithms for intersection, difference, and inclusion for attribute-element grammars. For lack of space, the formalizations of the difference and the inclusion algorithms are left out here (but given in Appendix A.2 and A.3). The proofs of correctness appear in the full version of this paper [15].

3.1 Partitioning

The key technique in our algorithms is *partitioning*. Consider first the following intersection of compound expressions.

$$(@a[x] | a[x]) (@b[x] | b[x]) \cap @a[y] (@b[y] | b[y]) \quad (1)$$

How can we calculate this intersection? A naive algorithm would separate constraints on attribute sets and those on element sequences

$$\begin{aligned} & (@a[x]@b[x] \mid @a[x]b[x] \mid a[x]@b[x] \mid a[x]b[x]) \\ & \cap \quad (@a[y]@b[y] \mid @a[y]b[y]) \end{aligned}$$

and compute the intersection of every pair of clauses on both sides. Such use of “distributive laws” makes the algorithm easily blow up. Fortunately, we can avoid it in typical cases. Note that each expression in the formula (1) is the concatenation of two subexpressions, where the left subexpressions on both sides contain the names $@a$ and a and the right subexpressions contain the different names $@b$ and b . In such a case, we can compute intersections of the left subexpressions and of the right subexpressions separately, and concatenate the results:

$$((@a[x] \mid a[x]) \cap @a[y]) \quad ((@b[x] \mid b[x]) \cap (@b[y] \mid b[y]))$$

The intuition behind why this works is that each “partitioned” expression can be regarded as cross products, and therefore the intersection of the whole expressions can be done by intersecting each corresponding pair of subexpressions. Note also that no subexpression is duplicated by this partitioning process. Therefore the algorithm proceeds linearly in the size of the inputs as long as partitioning can be applied. This idea of splitting expressions into orthogonal parts was inspired by Vouillon’s unpublished work on shuffle expressions [26]. We will discuss the difference of our work from his in Section 4.

For treating partitioning in our formalization, it is convenient to view a nested concatenation of expressions as a flat concatenation and ignore empty sequences (e.g., view $(c_1 (c_2 \epsilon)) c_3$ as $c_1 c_2 c_3$). In addition, we would like to treat expressions to be “partially commutative,” that is, concatenated c_1 and c_2 can be exchanged if one of them is element-free. For example, the expression $@a[x] (@b[x] \mid b[x])$ is equal to $(@b[x] \mid b[x]) @a[x]$. On the other hand, $(@a[x] \mid a[x]) (@b[x] \mid b[x])$ is *not* equal to $(@b[x] \mid b[x]) (@a[x] \mid a[x])$ since, this time, $a[x]$ prevents such an exchange.

Formally, we identify expressions up to the relation \equiv defined as follows: \equiv is the smallest congruence relation including the following.

$$\begin{aligned} c_1 c_2 & \equiv c_2 c_1 && \text{if } \mathbf{elm}(c_1) = \emptyset \\ c_1 (c_2 c_3) & \equiv (c_1 c_2) c_3 \\ c\epsilon & \equiv c \end{aligned}$$

Now, $(c'_1, c''_1), \dots, (c'_k, c''_k)$ is a *partition* of c_1, \dots, c_k if

$$\begin{aligned} c_i & = c'_i c''_i && \text{for all } i \\ (\bigcup_i \mathbf{att}(c'_i)) \cap (\bigcup_i \mathbf{att}(c''_i)) & = \emptyset \\ (\bigcup_i \mathbf{elm}(c'_i)) \cap (\bigcup_i \mathbf{elm}(c''_i)) & = \emptyset. \end{aligned}$$

That is, each c_i can be split into two subexpressions such that the names contained in all the first subexpressions are disjoint with those contained in all the second subexpressions. We will use partition of two expressions ($k = 2$) in the intersection algorithm and that of an arbitrary number of expressions in the difference. The partition is said *proper* when $0 < \mathbf{w}(c'_i) < \mathbf{w}(c_i)$ for some i . Here, the function \mathbf{w} counts the number of expressions that are concatenated at the top level (except ϵ). That is, $\mathbf{w}(\epsilon) = 0$, $\mathbf{w}(c_1 c_2) = \mathbf{w}(c_1) + \mathbf{w}(c_2)$, and $\mathbf{w}(c) = 1$ if $c \neq \epsilon$ and $c \neq c_1 c_2$. (Note that \equiv preserves \mathbf{w} .) This properness will be used for ensuring the boolean and inclusion algorithms to make a progress.

- 1) $\mathbf{inter}(e, f) = \mathbf{inter}^{\text{reg}}(e, f)$
- 2) $\mathbf{inter}(@N[x]^+, d) = \emptyset \quad (N \cap \mathbf{att}(d) = \emptyset)$
- 3) $\mathbf{inter}(@N[x], d) = \emptyset \quad (N \cap \mathbf{att}(d) = \emptyset)$
- 4) $\mathbf{inter}(@N[x]^+, @N[y]^+) = @N[\langle x, y \rangle]^+$
- 5) $\mathbf{inter}(@N[x], @N[y]^+) = @N[\langle x, y \rangle]$
- 6) $\mathbf{inter}(@N[x], @N[y]) = @N[\langle x, y \rangle]$
- 7) $\mathbf{inter}(c, d) = \mathbf{inter}(c_1, d_1) \mathbf{inter}(c_2, d_2)$
if $(c_1, c_2), (d_1, d_2)$ is a proper partition of c, d
- 8) $\mathbf{inter}(c_1 (c_2 | c_3) c_4, d) = \mathbf{inter}(c_1 c_2 c_4, d) | \mathbf{inter}(c_1 c_3 c_4, d)$

In addition, rules 2, 3, 5, and 8 each have a symmetric rule.

Figure 1: Intersection algorithm

3.2 Intersection

Let grammars F on X and G on Y be given. We assume that F and G have been *normalized*. That is, the given grammars have already been transformed so that all name sets appearing in them are pair-wise either equal or disjoint. The reason for doing this is to simplify our boolean and inclusion algorithms. For example, in computing the intersection $@N_1[x] @N_2[x] \cap @N_3[y] @N_4[y]$, if N_1 and N_2 are respectively equal to N_3 and N_4 , then this intersection is obvious. However, if these are overlapping in a non-trivial way (e.g., $@\{a, b\}[x] @\{c, d\}[x] \cap @\{a, c\}[y] @\{b, d\}[y]$), it would require more work. An actual algorithm for normalization is presented in Appendix A.1.

Our intersection algorithm is based on product construction. From F and G , we compute a new grammar H on $X \times Y$ that satisfies

$$H(\langle x, y \rangle) = \mathbf{inter}(F(x), G(y))$$

for all $x \in X$ and $y \in Y$. The function **inter** computes an intersection of compound expressions. It works roughly in the following way. We proceed the computation by progressively decomposing the given compound expressions. At some point, they become attribute-free. Then, we convert the expressions to element automata (defined later), compute an intersection by using a variant of the standard automata-based algorithm, and convert back the result to an expression. Formally, **inter** is defined in Figure 1. The base cases are handled by rules 1 through 6, where each of the arguments is either an element expression (as indicated by the metavariables e or f) or a single- or multi-attribute expressions. In rule 1, where both arguments are element expressions, we pass them to another intersection function $\mathbf{inter}^{\text{reg}}$ specialized to element expressions. This function will be explained below. Rules 2 and 3 return \emptyset since the argument expressions obviously denote disjoint sets. Rule 4, 5, and 6 handle the cases where each argument is a single- or multi-attribute expression with the same name set N . When both arguments are multi-attributes, rule 4 yields a multi-attribute where the name set is N and the content is the intersection of their contents x and y . When either argument is a single-attribute, rule 5 or 6 returns a single-attribute. (Note that, in rules 4 to 6, normalization ensures that the name sets in the given expressions are equivalent.) The inductive cases are handled by rules 7 and 8. Rule 7 applies the partitioning technique already explained. Rule 8 simply expands one union form appearing in the argument expressions.

Although it may not be obvious at first sight, the presented rules cover all the cases. To see this, first let us view each given expression as a sequence of the form $c_1 \dots c_k$ where each c_i is

either a multi-attribute $@N[x]^+$, a single-attribute $@N[x]$, a union $c_1 | c_2$, or an element expression e . There are two cases: (1) one of the two given expressions contains at least one union or (2) neither has a union. The first case is handled by rules 7 and 8. The actual algorithm applies rule 7 as often as possible, but rule 8 can always serve as fall backs. (One might think that cases like $\epsilon(c_1 | c_2)\epsilon$ are not handled. However, since, as stated in the previous subsection, we identify expressions up to associativity, partial commutativity, and neutrality of ϵ , such cases are already handled by rule 8.) In the second case, both of the given expressions are sequences consisting of single- or multi-attributes and element expressions. The case that both sequences have width zero or one is handled by rule 1 through 6. The remaining case is therefore that either sequence has width two or more. By recalling that attributes are normalized, we can always find a proper partition for such expressions; therefore this case is handled by rule 7.

The intersection function $\mathbf{inter}^{\text{reg}}$ performs the following: (1) construct element automata M_1 and M_2 from element expressions e_1 and e_2 , (2) compute the “product automaton” M from M_1 and M_2 , and (3) convert M back to an element expression e . Element automata are defined as follows. First, an *automaton* M on an alphabet Σ is a tuple $(Q, q^{\text{init}}, Q^{\text{fin}}, \delta)$ where Q is a finite set of *states*, $q^{\text{init}} \in Q$ is an *initial state*, $Q^{\text{fin}} \subseteq Q$ is a set of *final states*, and $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation* [14]. Then, an *element automaton* is an automaton over $\{N[x] \mid N \in S, x \in X\}$, where S is a set of name sets and X is a set of variables. Since well-known conversion algorithms between automata and regular expressions can directly be used for the case of element automata and element expressions by assuming $N[x]$ as symbols, we use them for (1) and (3) parts of the $\mathbf{inter}^{\text{reg}}$ function.

The product construction for element automata (used for the (2) part of $\mathbf{inter}^{\text{reg}}$) is slightly different from the standard one. Usually, product construction generates, from two transitions with the same label in the input automata, a new transition with that label in the output automaton. In our case, we generate, from a transition with label $N[x]$ and another with label $N[y]$, a new transition with label $N[\langle x, y \rangle]$. Formally, given two element automata $M_i = (Q_i, q_i^{\text{init}}, Q_i^{\text{fin}}, \delta_i)$ on $\{N[x] \mid N \in S, x \in X_i\}$ ($i = 1, 2$), the *product* of M_1 and M_2 is an automaton $(Q_1 \times Q_2, \langle q_1^{\text{init}}, q_2^{\text{init}} \rangle, Q_1^{\text{fin}} \times Q_2^{\text{fin}}, \delta)$ on $\{N[\langle x_1, x_2 \rangle] \mid N \in S, x_1 \in X_1, x_2 \in X_2\}$ where

$$\delta = \{ (\langle q_1, q_2 \rangle, N[\langle x_1, x_2 \rangle], \langle q'_1, q'_2 \rangle) \mid (q_i, N[x_i], q'_i) \in \delta_i \text{ for } i = 1, 2 \}.$$

(Note that we use here the assumption that the name sets of *elements* in the given grammars have been normalized.)

We can prove the following expected property for our intersection algorithm.

Theorem 1 *Let $H(\langle x, y \rangle) = \mathbf{inter}(F(x), G(y))$. Then, $\mathbf{inter}(c, d) = b$ implies that $H \vdash v \in b$ iff $F \vdash v \in c$ and $G \vdash v \in d$.*

The intersection algorithm takes at most a quadratic time in the numbers of variables in the given grammars. However, for each pair of variables, it takes an exponential time in the size of the expressions assigned to the variables in the worst case, where the function needs to fully expand the expressions by using rule 8. There is no other exponential factor in this algorithm. (The function $\mathbf{inter}^{\text{reg}}$ can be computed in a polynomial time since each of the three steps is polynomial.)

3.3 Difference

3.3.1 Restrictions

Our expressions, as they are defined as in Section 2, do not have closure under difference. The kinds of expressions that break the closure are single-attributes $@N[x]$ and multi-attributes $@N[x]^+$ where

N is infinite (in both cases). For single-attributes, consider the difference $@\mathcal{N}[\mathbf{any}]^* \setminus @\mathcal{N}[\mathbf{any}]$ (where \mathcal{N} is the name set containing all names). This would mean zero, two, or more attributes with any name and any content. However, “two or more” is not expressible in our framework. For multi-attributes, consider the difference $@N[\mathbf{any}]^+ \setminus @N[x]^+$ where N is infinite. The resulting expression should satisfy the following. Each value in it has a set of attributes all with names from N . But *at least* one of them has a content *not* satisfying x . The expression $@N[\bar{x}]^+$ is not a right answer because it requires *all* attributes to have contents not satisfying x .

For this reason, we impose two syntactic restrictions. First, the name set of a single-attribute expression must be a singleton. Note that, with this restriction, we can still represent the case where N is finite¹: when $N = \{a_1, \dots, a_k\}$

$$@N[x] \equiv @a_1[x] \mid \dots \mid @a_k[x]$$

On the other hand, the case that N is infinite is not expressible. We consider, however, that this restriction is acceptable from practical point of view since an infinite name set is usually used for representing “arbitrary names from a specific name space” and one naturally wants an arbitrary number of attributes from such a name set. The second restriction is that the content of a multi-attribute expression must be a distinguished variable **any** accepting any values. We assume that any given grammar G has the following mapping.

$$G(\mathbf{any}) = \mathcal{N}[\mathbf{any}]^* @\mathcal{N}[\mathbf{any}]^*$$

We can still represent the case where N is finite and the content is not **any**: when $N = \{a_1, \dots, a_k\}$, we can apply the normalization procedure (Appendix A.1) with the shredded set $\{\{a_1\}, \dots, \{a_k\}\}$. On the other hand, we cannot handle the case where N is infinite and the content is not **any**. However, this restriction seems reasonable since an open schema typically wants both the name and the content to be generic—making only the content specific seems rather unnatural.

3.3.2 Algorithm

The difference algorithm is similar to the intersection algorithm since it uses partitioning, but it is somewhat more complicated because of the need to apply subset construction at the same time. To see this, consider the following difference.

$$@a[x] @b[x] \setminus (@a[y] @b[y] \mid @a[z] @b[z])$$

First of all, note that each of the left expression and the right expressions under the union can be partitioned to the one with $@a$ and the one with $@b$; these components can be regarded as orthogonal. We proceed the difference by first subtracting $@a[y] @b[y]$ from $@a[x] @b[x]$. This yields

$$(@a[x] \setminus @a[y]) @b[x] \mid @a[x] (@b[x] \setminus @b[y]).$$

That is, we obtain the union of two expressions, one resulting from subtracting the first component and the other resulting from subtracting the second. This can be understood by observing that “a value $@a[v] @b[w]$ being not in $@a[y] @b[y]$ ” means “either $@a[v]$ being not in $@a[y]$ or $@b[w]$ being not in $@b[y]$.” Now, back to the original difference calculation, we next subtract the second clause

¹RELAX NG adopts this restriction of finiteness.

$@a[z] @b[z]$ from the above result. Performing a similar subtraction, we obtain:

$$\begin{array}{l}
(@a[x] \setminus (@a[y] | @a[z])) \quad @b[x] \\
| \quad (@a[x] \setminus @a[y]) \quad (@b[x] \setminus @b[z]) \\
| \quad (@a[x] \setminus @a[z]) \quad (@b[x] \setminus @b[y]) \\
| \quad @a[x] \quad (@b[x] \setminus (@b[y] | @b[z]))
\end{array}$$

Thus, the original goal of difference has reduced to the combination of the subgoals of difference. Let us consider only the first difference $(@a[x] \setminus (@a[y] | @a[z]))$ since the other are similar. Since all expressions appearing here are single-attributes with $@a$, the result is obviously a single-attribute with $@a$. What is less obvious is that its content is the difference between the variable x and the “union” of the variables y and z . In general, given two grammars, we have to compute the difference between a variable from one grammar and a *set* of variables from the other grammar. This is why the algorithm involves subset construction.

3.4 Inclusion

One way of deciding inclusion is to compute a difference and then test the emptiness of the result. In this approach, we would need the the restrictions described in the previous section since otherwise difference is not computable. Below, we show a slight variation of this approach that does not require these restrictions. The idea is to compute, in the first step, an expression that denotes “approximately” the difference but whose emptiness is exactly the same as the difference. We call this approximate difference *residual*.

By dropping the restrictions, we need to additionally treat (1) single-attributes with infinite name sets and (2) multi-attributes with infinite name sets and non-**any** content. Accordingly, our residual algorithm must handle the following two tricky cases.

The first case is when we subtract single- or multi-attributes from a single-attribute, where all the name sets are N (which is possibly infinite) and the contents x and y 's are not necessarily **any**.

$$@N[x] \setminus (@N[y_1]^+ | \dots | @N[y_l]^+ | @N[y_{l+1}] | \dots | @N[y_k]) \quad (2)$$

Since the left hand side contains only values with width one, we can restrict the right hand side to values with width one. Thus, we can transform (2) as follows

$$@N[x] \setminus (@N[y_1] | \dots | @N[y_k])$$

which is equivalent to

$$@N[x \setminus (y_1 | \dots | y_k)].$$

The second case is when we subtract single- or multi-attributes from a multi-attribute.

$$@N[x]^+ \setminus (@N[y_1]^+ | \dots | @N[y_l]^+ | @N[y_{l+1}] | \dots | @N[y_k]) \quad (3)$$

We simplify this formula in two steps. First, since the left hand side is a multi-attribute, only multi-attributes on the right hand side can contribute in covering the values on the left hand side. In other words, if the multi-attributes on the right hand side cannot cover the left hand side, then adding single-attributes makes no difference. Therefore we can drop single-attributes from the right hand side:

$$@N[x]^+ \setminus (@N[y_1]^+ | \dots | @N[y_l]^+) \quad (4)$$

The second step transforms this to the following

$$@a_1[x \setminus y_1]^+ \dots @a_l[x \setminus y_l]^+ \quad (5)$$

where a_1, \dots, a_l are arbitrary different names taken from N . We can see that the emptinesses of (4) and (5) are the same. Suppose that (5) is empty. Then, we can say $x \subseteq y_i$ for some i , and therefore $@N[x]^+ \subseteq @N[y_i]^+$, which implies that (4) is empty. The other direction holds since (5) is equal to or smaller than (4). Indeed, take an arbitrary instance from the bottom

$$@a_1[v_1] \dots @a_l[v_l]$$

where

$$v_1 \in (x \setminus y_1) \quad \dots \quad v_l \in (x \setminus y_l).$$

This value is in $@N[x]^+$ since all v 's are taken from x , but is not in either of $@N[y_i]^+$ since v_i is not from y_i .

Note that the last trick works only when N is “big enough”—that is, the cardinality of N is equal to or larger than l . As a counterexample, the following holds

$$@\{a, b\}[x]^+ \setminus (@\{a, b\}[y_1]^+ | @\{a, b\}[y_2]^+ | @\{a, b\}[y_3]^+) = \emptyset$$

where x, y_1, y_2 , and y_3 each denote the following finite sets:

$$\{v_1, v_2, v_3\} \quad \{v_1, v_2\} \quad \{v_2, v_3\} \quad \{v_3, v_1\}$$

Note that, even though x contains three values, the name set $\{a, b\}$ allows the left hand side $@\{a, b\}[x]^+$ to generate only values of width at most two. All such values are covered by the right hand side. However, the transformation from (4) to (5) does not work since we cannot take three different names from the set $\{a, b\}$. In the case that a multi-attribute has a finite name set, we break it into a combination of single-attribute expressions with a singleton name set and apply the rule for single-attributes.

4 Related Work

Our study on attribute constraints has a strong relationship to type theories for record values (i.e., finite mappings from labels to values). Early papers presenting type systems for record types do not consider the union operator and therefore no such complication arises as in our case. (A comprehensive survey of classical records can be found in [13].) Buneman and Pierce have investigated record types with the union operator [5]. Their system does not, however, have any mechanism similar to our multi-attribute expressions or recursion. Frisch, Castagna, and Benzaken [12] have designed a typed XML processing language CDuce that supports attribute types based on records. Although the descriptive power of their types is the same as ours, type expressions to represent interdependency between attributes and elements are exponentially larger than ours since they do not allow mixture of element and attribute constraints. The DSD schema language [20], designed by Klarlund, Møller, and Schwartzbach, can also express the kinds of attribute-element interdependencies discussed here. However, this schema language is not closed under intersection or complementation. Its descendent DSD2 [22] has the expressiveness of concern and is closed under boolean operations.

In his unpublished work, Vouillon has considered an algorithm for checking the inclusion relation between shuffle expressions [26]. His strategy of progressively decomposing given expressions to two

orthogonal parts made much influence on our boolean and inclusion algorithms. The difference is that his algorithm directly answers yes or no without constructing new grammars like our case, and therefore does not incur the complication of switching back and forth between the expression representation and the automata representation.

Another important algorithmic problem related schemas is validation. There have been several validation algorithms proposed for attribute-element constraints. One is designed and implemented by Clark [7] based on derivatives of regular expressions [4, 1]. Another is presented by Hosoya and Murata using so-called attribute-element automata [16].

5 Future work

In this paper, we have presented our intersection, difference, and inclusion algorithms. We have already implemented them in the XDuce language [17]. For the examples that we have tried, the performance seems quite reasonable. We plan to collect and analyze data obtained from the experiment on the algorithms in the near future. We also feel that we need some theoretical characterization of the algorithms. In particular, our algorithms contain potentials of blow up in many places. Although our implementation techniques presented in the full version of this paper [15] have been sufficient for our examples, one would like to have some more confidence.

Acknowledgments

We learned a great deal from discussion with James Clark, Kohsuke Kawaguchi, Akihiko Tozawa, Benjamin Pierce, and members of the Programming Languages Club at University of Pennsylvania. We are also grateful to anonymous PLAN-X referees, who help me improving this paper substantially. Hosoya has been partially supported by Japan Society for the Promotion of Science and Kayamori Foundation of Informational Science Advancement while working on this paper.

References

- [1] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.
- [2] T. Bray, D. Hollander, A. Layman, and J. Clark. Namespaces in XML. <http://www.w3.org/TR/REC-xml-names>, 1999.
- [3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XMLTM). <http://www.w3.org/XML/>, 2000.
- [4] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, Oct. 1964.
- [5] P. Buneman and B. Pierce. Union types for semistructured data. In *Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 1998. Proceedings of the International Database Programming Languages Workshop.
- [6] J. Clark. TREX: Tree Regular Expressions for XML. <http://www.thaiopensource.com/trex/>, 2001.
- [7] J. Clark. <http://www.thaiopensource.com/relaxng/implement.html>, 2002.
- [8] J. Clark and M. Murata. RELAX NG. <http://www.relaxng.org>, 2001.
- [9] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Draft book; available electronically on <http://www.grappa.univ-lille3.fr/tata>, 1999.

- [10] D. C. Fallside. XML Schema Part 0: Primer, W3C Recommendation. <http://www.w3.org/TR/xmlschema-0/>, 2001.
- [11] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 Formal Semantics. <http://www.w3.org/TR/query-semantics/>, 2001.
- [12] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science*, 2002.
- [13] R. Harper and B. Pierce. A recrd calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando FL*, pages 131–142. ACM, Jan. 1991.
- [14] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [15] H. Hosoya. On attribute-element constraints. Full version, 2002.
- [16] H. Hosoya and M. Murata. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*, pages 1–10, 2002.
- [17] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997 of *Lecture Notes in Computer Science*, pages 226–244, May 2000.
- [18] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, Jan. 2001.
- [19] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, Sept. 2000. Full version under submission to TOPLAS.
- [20] N. Klarlund, A. Møller, and M. I. Schwartzbach. DSD: A schema language for XML. <http://www.brics.dk/DSD/>, 2000.
- [21] T. Milo, D. Suciú, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.
- [22] A. Møller. Document Structure Description 2.0, December 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7. Available from <http://www.brics.dk/DSD/>.
- [23] M. Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing '96*, volume 1293 of *Lecture Notes in Computer Science*, pages 153–169. Springer-Verlag, 1997.
- [24] M. Murata. RELAX (REGular LANGUAGE description for XML). <http://www.xml.gr.jp/relax/>, 2001.
- [25] A. Tozawa. Towards static type inference for XSLT. In *Proceedings of ACM Symposium on Document Engineering*, 2001.
- [26] J. Vouillon. Interleaving types for XML. Personal communication, 2001.

A Addendum algorithm formalizations

A.1 Name set normalization

Let $\{N_1, \dots, N_k\}$ be the set of name sets appearing in given grammars. (When we are given two grammars, this set includes all the names both in the grammars.) From this, we generate a set of

disjoint name sets by the following **shred** function. (Here, \uplus is the disjoint union.)

$$\mathbf{shred}(\{N\} \uplus S) = \begin{cases} \{N\} \cup \mathbf{shred}(S) \setminus \{\emptyset\} \\ \qquad \qquad \qquad \text{if } N \cap (\bigcup S) = \emptyset \\ \{N \setminus (\bigcup S)\} \cup \\ \mathbf{shred}(\{N' \cap N \mid N' \in S\}) \cup \\ \mathbf{shred}(\{N' \setminus N \mid N' \in S\}) \setminus \{\emptyset\} \\ \qquad \qquad \qquad \text{otherwise} \end{cases}$$

$$\mathbf{shred}(\emptyset) = \emptyset$$

That is, we pick one member N from the input set. If this member is already disjoint with any other member, we continue shredding for the remaining set S . Otherwise, we first divide each name set N' in S into the disjoint sets $N' \cap N$ and $N' \setminus N$. We separately shred all the name sets of the first form and those of the second form. We then combine the results from these and the name set obtained by subtracting all the members in S from N . As an example, this function shreds the set $\{\{a, b\}, \{b, c\}, \{a, c\}\}$ in the following way.

$$\begin{aligned} & \mathbf{shred}(\{\{a, b\}, \{b, c\}, \{a, c\}\}) \\ &= \{\{a\}\} \cup \mathbf{shred}(\{\{b\}, \{a\}\}) \cup \mathbf{shred}(\{\{c\}\}) \\ &= \{\{a\}\} \cup \{\{b\}\} \cup \{\{a\}\} \cup \{\{c\}\} \\ &= \{\{a\}, \{b\}, \{c\}\} \end{aligned}$$

This function may blow up since the “otherwise” case above uses two recursive calls to **shred** where the size of the arguments do not necessarily decrease by half. (Indeed, we can easily construct an initial set $\{N_1, \dots, N_k\}$ such that every N_i is not disjoint with and not a subset of the union of the other name sets. For this set, the **shred** function takes $O(2^k)$ time and returns a set of size $O(2^k)$.) However, this does not seem to happen in practice since the initial name sets are usually mostly disjoint and therefore the case $N \cap (\bigcup S) = \emptyset$ is taken in most of the time.

Having computed a shredded set $S = \mathbf{shred}(\{N_1, \dots, N_k\})$, we next replace every occurrence of the form $N[x]$, $@N[x]^+$, or $@N[x]$ with an expression that contains only name sets in S . We obtain such an expression by using the following **norm** function.

$$\begin{aligned} \mathbf{norm}_S(\emptyset[x]) &= \emptyset \\ \mathbf{norm}_S((N \uplus N')[x]) &= \\ & N[x] \mid \mathbf{norm}_S(N'[x]) \quad (N \in S) \\ \\ \mathbf{norm}_S(@\emptyset[x]^+) &= \emptyset \\ \mathbf{norm}_S(@N \uplus N'[x]^+) &= \\ @N[x]^+ \mid ((\epsilon \mid @N[x]^+) \mathbf{norm}_S(@N'[x]^+)) \quad (N \in S) \\ \\ \mathbf{norm}_S(\emptyset[x]) &= \emptyset \\ \mathbf{norm}_S(@N \uplus N'[x]) &= \\ @N[x] \mid \mathbf{norm}_S(@N'[x]) \quad (N \in S) \end{aligned}$$

In the special case that a given $@N[x]^+$ is unioned with ϵ , we can transform it to a somewhat simpler form as follows.

$$\begin{aligned} \mathbf{norm}'_S(@\emptyset[x]^+ \mid \epsilon) &= \epsilon \\ \mathbf{norm}'_S(@N \uplus N'[x]^+ \mid \epsilon) &= \\ @N[x]^+ \mid \epsilon \mid \mathbf{norm}'_S(@N'[x]^+ \mid \epsilon) \quad (N \in S) \end{aligned}$$

$$\begin{array}{ll}
1) \quad \mathbf{diff}(e, \{f_1, \dots, f_k\}) & = \mathbf{diff}^{\text{reg}}(e, f_1 \mid \dots \mid f_k) \\
2) \quad \mathbf{diff}(c, \{\text{@}N[y]^+ \uplus D\}) & = \mathbf{diff}(c, D) \quad (\mathbf{att}(c) \cap N = \emptyset) \\
3) \quad \mathbf{diff}(\text{@}N[x]^+, \{d\} \uplus D) & = \mathbf{diff}(\text{@}N[x]^+, D) \quad (N \cap \mathbf{att}(d) = \emptyset) \\
4) \quad \mathbf{diff}(\text{@}a[x], \{\text{@}a[y_1], \dots, \text{@}a[y_k]\}) & = \text{@}a[\langle x, \{y_1, \dots, y_k\} \rangle] \\
5) \quad \mathbf{diff}(\text{@}N[\mathbf{any}]^+, \{\text{@}N[\mathbf{any}]^+\}) & = \emptyset \\
6) \quad \mathbf{diff}(c, \{d_1, \dots, d_k\}) & = \big|_{I \subseteq \{1, \dots, k\}} \mathbf{diff}(c', \{d'_i \mid i \in I\}) \\
& \quad \mathbf{diff}(c'', \{d''_i \mid i \in \{1, \dots, k\} \setminus I\}) \\
& \quad \text{if } (c', c''), (d'_1, d''_1), \dots, (d'_k, d''_k) \text{ is a proper partition of } c, d_1, \dots, d_k \\
7) \quad \mathbf{diff}(c_1 (c_2 \mid c_3) c_4, D) & = \mathbf{diff}(c_1 c_2 c_4, D) \mid \mathbf{diff}(c_1 c_3 c_4, D) \\
8) \quad \mathbf{diff}(c, \{d_1 (d_2 \mid d_3) d_4\} \uplus D) & = \mathbf{diff}(c, \{d_1 d_2 d_4, d_1 d_3 d_4\} \uplus D)
\end{array}$$

Figure 2: Difference algorithm

This specialized rule is important since the straightforward form of concatenations yielded by the rule gives more opportunities to the partitioning technique, compared to the complex form yielded by the general rule, where unions and concatenations are nested each other. In our experience, $\text{@}N[x]^+$ almost always appears in the form $\text{@}N[x]^+ \mid \epsilon$ since the user typically writes zero or more repetitions rather than one or more.

The worst-case complexity of the whole normalization procedure is as follows. Suppose that there are n occurrences of atomic form ($N[x]$, $\text{@}N[x]^+$, or $\text{@}N[x]$) in the given grammars. Since there are at most n different name sets, shredding takes $O(2^n)$ and returns a set of size $O(2^n)$ at most. Since we apply the **norm** or **norm'** function for each occurrence of atomic form and each takes linear time in the size of the shredded set, the whole normalization takes $O(2^n)$ and results in grammars of size $O(2^n)$ in the worst case.

A.2 Difference algorithm

Let grammars F on X and G on Y be given and have been normalized simultaneously, as before. The difference between F and G is to compute a new grammar H on $X \times \mathcal{P}(Y)$ that satisfies

$$H(\langle x, Z \rangle) = \mathbf{diff}(F(x), \{G(y) \mid y \in Z\})$$

for all $x \in X$ and $Z \subseteq Y$. The function **diff** takes a compound expression c and a set of compound expressions d_i and returns a difference between c and the union of d_i 's. The definition of this function is presented in Figure 2. (Here, D ranges over sets of compound expressions and \uplus is the disjoint union.) The base cases are handled by rules 1 through 5. As before, when all the arguments are element expressions, rule 1 passes them to the difference function $\mathbf{diff}^{\text{reg}}$ (explained below). Rules 2 and 3 remove, from the set in the second argument, an expression that is disjoint with the first argument. Rule 4 handles that all expressions in the arguments are single-attributes. Note that, by the above-mentioned restriction, the name set is a singleton. Rule 5 handles that all expressions are multi-attributes. Since the content is **any** as required by the restriction, the rule returns the empty set expression.

The inductive cases are handled by rule 6 through 8. Rules 7 and 8 expand one union form in the argument expressions. Rule 6 is applied when all the arguments can be partitioned altogether. The complex formula involving “for all subsets $I \subseteq \{1, \dots, k\}$ ” on the right hand side is a generalization of the discussion made in the beginning of this section. This “subsetting” technique has repeatedly been used in the literature. Interested readers are referred to [19, 18, 12].

The function $\mathbf{diff}^{\text{reg}}$ is analogous to the function $\mathbf{inter}^{\text{reg}}$ already shown: it constructs element automata M_1 and M_2 from element expressions e_1 and e_2 , then computes the “difference automaton” M from M_1 and M_2 , and finally converts M back to an element expression e . The construction of difference automata uses both product and subset construction. Given two element automata $M_i = (Q_i, q_i^{\text{init}}, Q_i^{\text{fin}}, \delta_i)$ on $\{N[x] \mid N \in S, x \in X_i\}$ ($i = 1, 2$), the *difference* of M_1 and M_2 is an automaton $(Q, q^{\text{init}}, Q^{\text{fin}}, \delta)$ on $\{N[\langle x_1, Y_2 \rangle] \mid N \in S, x_1 \in X_1, Y_2 \subseteq X_2\}$ where:

$$\begin{aligned} Q &= Q_1 \times \mathcal{P}(Q_2) \\ q^{\text{init}} &= \langle I_1, \{I_2\} \rangle \\ Q^{\text{fin}} &= \{ \langle q_1, P \rangle \mid q_1 \in F_1 \text{ and } P \subseteq Q_2 \text{ and } P \cap F_2 \neq \emptyset \} \\ \delta &= \\ &\{ (\langle q_1, \{p_1, \dots, p_k\} \rangle, N[\langle x_1, \{y_1, \dots, y_k\} \rangle], \langle q'_1, \{p'_1, \dots, p'_k\} \rangle) \mid \\ &\quad (q_1, N[x_1], q'_1) \in \delta_1 \text{ and} \\ &\quad \{ (p_1, N[y_1], q'_1), \dots, (p_k, N[y_k], q'_k) \} \subseteq \delta_2 \} \end{aligned}$$

We can prove the following property expected for the difference algorithm.

Theorem 2 *Let $H(\langle x, Z \rangle) = \mathbf{diff}(F(x), \{G(y) \mid y \in Z\})$. Then $\mathbf{diff}(c, D) = b$ implies that $H \vdash v \in b$ iff $F \vdash v \in c$ and $G \vdash v \notin d$ for all $d \in D$.*

The worst-case complexity of the difference algorithm is as follows. It is linear in the size of $\mathbf{dom}(F)$ and exponential in the size of $\mathbf{dom}(G)$. For each pair $\langle x, Z \rangle$, the computation of the \mathbf{diff} function takes an exponential time in the size of the first argument and a double exponential time in the size of the second argument. There are two exponential factors. The first comes from that the given expressions may have to be fully expanded by rules 7 or 8 in the worst case. This factor applies to both arguments. The second exponential factor, which applies only to the second argument, comes from subset construction performed in rules 1 and 6.

A.3 Inclusion algorithm

Let grammars F on X and G on Y be given and have been normalized simultaneously. The “residual” between F and G is a grammar H on $X \times \mathcal{P}(Y)$ that satisfies

$$H(\langle x, Z \rangle) = \mathbf{resid}(F(x), \{G(y) \mid y \in Z\})$$

for all $x \in X$ and $Z \subseteq Y$. The function \mathbf{resid} takes a compound expression c and a set of compound expressions d_i and returns an expression whose emptiness is the same as the difference between c and the union of d_i 's. The definition of the function \mathbf{resid} is given in Figure 3. Here, we assume that N in the form $@N[x]^+$ is an infinite set. We can prove the following expected properties.

Theorem 3 *Let $H(\langle x, Z \rangle) = \mathbf{resid}(F(x), \{G(y) \mid y \in Z\})$. Then $\mathbf{resid}(c, D) = b$ implies that $H \vdash v \in b$ for some v iff $F \vdash w \in c$ and $G \vdash w \notin d$ for all $d \in D$ for some w .*

Emptiness test can be formalized as follows. Given a grammar H , we compute a series of (total) functions ϕ_0, ϕ_1, \dots from the variables in $\mathbf{dom}(H)$ to booleans, as defined below.

$$\begin{aligned} \phi_0(x) &= \mathbf{false} \\ \phi_i(x) &= \mathbf{nemp}(H(x))\phi_{i-1} \end{aligned}$$

- 2') $\mathbf{resid}(c, \{\@N[y]\} \uplus D) = \mathbf{resid}(c, D)$ $(\mathbf{att}(c) \cap N = \emptyset)$
3') $\mathbf{resid}(\@N[x], \{d\} \uplus D) = \mathbf{resid}(\@N[x], D)$ $(N \cap \mathbf{att}(d) = \emptyset)$
4) $\mathbf{resid}(\@N[x], \{\@N[y_1]^+, \dots, \@N[y_l]^+, \@N[y_{l+1}], \dots, \@N[y_k]\}) = \@N[\langle x, \{y_1, \dots, y_k\} \rangle]$
5) $\mathbf{resid}(\@N[x]^+, \{\@N[y_1]^+, \dots, \@N[y_l]^+, \@N[y_{l+1}], \dots, \@N[y_k]\})$
 $= \@a_1[\langle x, \{y_1\} \rangle]^+ \dots \@a_l[\langle x, \{y_l\} \rangle]^+$ $(a_i \in N \text{ and } a_i \neq a_j \text{ for } i \neq j)$

In addition, we transfer rules 1, 2, 3, 6, 7, and 8 of **diff** (with the function symbol **diff** replaced by **resid**) to here. (Rule 1 still uses the function **diff**^{reg} for element expressions.)

Figure 3: Residual algorithm

where the function **nemp** is inductively defined as follows.

$$\begin{aligned}
\mathbf{nemp}(\@N[x]^+) \phi &= (N \neq \emptyset) \wedge \phi(x) \\
\mathbf{nemp}(\@N[x]) \phi &= (N \neq \emptyset) \wedge \phi(x) \\
\mathbf{nemp}(N[x]^+) \phi &= (N \neq \emptyset) \wedge \phi(x) \\
\mathbf{nemp}(\epsilon) \phi &= \mathbf{true} \\
\mathbf{nemp}(c_1 c_2) \phi &= \mathbf{nemp}(c_1) \phi \wedge \mathbf{nemp}(c_2) \phi \\
\mathbf{nemp}(c_1 | c_2) \phi &= \mathbf{nemp}(c_1) \phi \vee \mathbf{nemp}(c_2) \phi \\
\mathbf{nemp}(c^+) \phi &= \mathbf{nemp}(c) \phi
\end{aligned}$$

Intuitively, **nemp** returns whether the given expression is non-empty, assuming the non-emptiness of each variable appearing in the expression to be described by ϕ . We first assume that every variable is empty. Then, we repeatedly “update” the assumption by computing the non-emptiness of each variable under the previous assumption, until we reach the fixpoint. Thus, the answer is ϕ_n such that $\phi_n(x) = \phi_{n-1}(x)$ for all x .

Theorem 4 $\mathbf{nemp}(c) \phi_n = \mathbf{true}$ if and only if $H \vdash v \in c$ for some v .

Since the structure of the inclusion algorithm is similar to the difference, the worst-case complexity is the same. The emptiness test presented above is quadratic (in the size of the number of variables). However, we believe that it can be made linear by choosing an appropriate data structure (similarly to tree automata emptiness [9]).