Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 360 (2006) 327–351
Theoretical Computer Science
www.elsevier.com/locate/tcs

Boolean operations and inclusion test for attribute–element constraints

Haruo Hosoya[a,*], Makoto Murata[b]

[a] *Department of Computer Science, The University of Tokyo, Japan*
[b] *IBM Tokyo Research Laboratory, Japan*

Received 24 August 2004; received in revised form 21 December 2005; accepted 5 May 2006

Communicated by B. Pierce

Abstract

The history of schema languages for XML is (roughly) 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 constraint expression (we call his mechanism "attribute–element constraints"). In this paper, we investigate intersection and difference operations and inclusion test for attribute–element constraints, in view of their importance in static typechecking for XML processing programs. The contributions here are (1) proofs of closure under intersection and difference as well as decidability of inclusion test and (2) algorithm formulations incorporating a "divide-and-conquer" strategy for avoiding an exponential blow-up for typical inputs.
© 2006 Elsevier B.V. All rights reserved.

Keywords: XML; Schema; Formal language theory; Automata

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 [11], and RELAX [25]) was *elements*, and by now this pursue has mostly converged—most of modern schema languages use tree regular expressions for describing this part of data.

On the other hand, much less attention has been paid to attributes, although there has been a big demand for a higher expressiveness, in particular, involving interdependencies among elements and attributes. There are mainly two kinds of such interdependencies that are often wanted. The first one is to allow a choice between an element and an attribute. For example, one might want to allow each `person` element to have *either* a `name` subelement or a `name` attribute,

* Corresponding author. Tel.: +81 3 5841 4116.
E-mail addresses: hahosoya@is.s.u-tokyo.ac.jp (H. Hosoya), mmurata@trl.ibm.com (M. Murata).

0304-3975/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.05.004

but not both. It is one of the most perennial questions for the past 10 years among schema designers whether to put certain information in an attribute or an element (e.g., [27] collects related debates). Thus, it is natural for them to wish to make *either* case possible and such demand has been known among XML engineers as folklore. The second group is to allow attribute values to control permissible subelements. For example, in the Atom Syndication Format [26], the content of a certain element is required to be either a text or an XHTML fragment depending on the value of the attribute called `type`. Neither DTD nor W3C XML Schema allows these interdependencies among elements and attributes since these schema languages only admit attributes to be mandatory or optional. RELAX allows such interdependencies to some extent, but requires a schema description to be exponentially long for the first kind and rather unintuitive for the second kind.

Recently, Clark, in his schema language TREX [7], proposed a description mechanism called *attribute–element constraints* for fulfilling such demands. 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 these two kinds of structure. The expressive power yielded by this mechanism is quite substantial. In particular, both uses of attribute–element interdependencies mentioned above 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 [17,12,28,22,24]. For example, intersection is used in almost all of these XML processing 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 XML processing 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, closure 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 several times 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 cleanliness 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 [29], but has not been applied to the setting of attribute–element constraints. Our specific contributions here are thus the conditions to apply partitioning in our setting and the proofs of correctness.

This work has been carried out in collaboration with committee members of the new schema language RELAX NG [9]. As a result, aiming for closure under boolean operations, RELAX NG adopted the syntactic restrictions described in this paper.

The rest of this paper is organized as follows. The next section gives motivating examples for attribute–element constraints and basic definitions including the data model and the syntax and semantics of constraints. Section 3 describes the intersection, difference, and inclusion algorithms. Section 4 presents some implementation techniques.

Section 5 discusses the relationship with other work and Section 6 concludes the paper. Addendum formalizations and correctness proofs can be found in Appendices.

2. Attribute–element constraints

In this section, we first informally explain attribute–element constraints by example and then formally define the syntax and semantics.

2.1. Informal explanation

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. We first illustrate element-only constraints and then attribute–element constraints. Element expressions describe constraints on elements and are regular expressions on names. For example, we can write

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

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 elements. The actual constraint mechanism formalized later can also describe contents of elements.

Attribute expressions are constraints on attributes and have a notation similar to regular expressions. For example,

$$\texttt{@key}\ \texttt{@year}?$$

requires a `key` attribute and optionally allows a `year` attribute. We prepend an @-sign to each attribute name in order to distinguish attribute names from element names. Again, although informal examples here show constraints only on names of attributes, we later introduce constraints on contents of attributes. A more complex example would be

$$\texttt{@key}\,((\texttt{@year}\ \texttt{@month}?)\mid\texttt{@date})$$

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 (e.g., `@key @year?` is equivalent to `@year? @key`). 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 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

$$\texttt{@key}\ \texttt{@year}?\ \texttt{author}^{+}\ \texttt{title}\ \texttt{publisher}?$$

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 `title` attribute or a `title` element, not both:

```
@key @year? author+ (title | @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

```
(@key | key)
(@year | year)?
author+
(@title | title)
(@publisher | publisher)?
```

where every piece of information except for `author`s 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 number of any other attributes, we can write the following expression (where `(*\key\year)` represents the set of all names except `key` and `year`):

```
@key @year @ (*\key\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 the set of "any names except those in name space `myns`").

```
@myns:key @myns:year @ (*\(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 [9].)

2.2. Data model

We assume a countably infinite set $\mathcal{N}$ of *names*, ranged over by $a, b, \ldots$. 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. We write $\mathcal{V}$ for the set of all values.

For convenience in definitions and proofs in the sequel, we introduce the following slightly unusual notations for constructing values:

$$\begin{array}{rcl@{\qquad}rcl} a[v] & \equiv & \langle \emptyset, \langle a, v \rangle \rangle & @a[v] & \equiv & \langle \{\langle a, v \rangle\}, \varepsilon \rangle \\ \langle \alpha_1, \beta_1 \rangle \, \langle \alpha_2, \beta_2 \rangle & \equiv & \langle \alpha_1 \cup \alpha_2, \beta_1\beta_2 \rangle & \varepsilon & \equiv & \langle \emptyset, \varepsilon \rangle \end{array}$$

For example, $@a[v]\,@b[w]\,c[u]$ means $\langle \{\langle a, v \rangle, \langle b, w \rangle\}, \langle c, u \rangle \rangle$. It is crucial not to mistake $@a[v]\,@b[w]\,c[u]$ as a *sequence* consisting of two attributes and one element—we intend to formalize the unordered and unrepeatable nature of attributes.

2.3. 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 we want expressions to describe arbitrary depths of trees, we introduce recursive definitions of expressions, that is, grammars. (It is also theoretically important to study tree grammars rather than simple word grammars since, as will be discussed in Section 3.3.1, considering contents of attributes introduces a new complication not arising in word grammars, i.e., breakage of closure under difference and syntactic restrictions to recover the closure.)

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. [1]

$$
\begin{array}{rcl@{\qquad}rcl}
c & ::= & @N[x]^+ & e & ::= & N[x] \\
 & & @N[x] & & & \varepsilon \\
 & & c \mid c & & & e \mid e \\
 & & c\,c & & & e\,e \\
 & & e & & & e^+ \\
 & & & & & \emptyset
\end{array}
$$

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 $\bigcup_{x\in\mathbf{dom}(G)} \mathbf{FV}(x)$. We require any grammar to be "self-contained", i.e., $\mathbf{FV}(G) \subseteq \mathbf{dom}(G)$, where $\mathbf{dom}(G)$ is the domain of G. In the sequel, we use the following shorthands:

$$
\begin{array}{llllll}
@a[x]^+ & \equiv @\{a\}[x]^+ & @a[x] & \equiv @\{a\}[x] & a[x] & \equiv \{a\}[x] \\
e^* & \equiv e^+ \mid \varepsilon & c? & \equiv c \mid \varepsilon & @N[x]^* & \equiv @N[x]^+ \mid \varepsilon
\end{array}
$$

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 call this restriction *no-overlapping attribute*. [2] (Note that the no-overlapping restriction disallows the usual rewriting of one-or-more repetitions $@N[x]^+$ by $@N[x]^*\,@N[x]$ and this is why we take the former as primitive. For symmetry, we also take one-or-more repetitions e^+ as primitive in element-only constraints.) 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.

We require that a repetition containing attributes must have the form $@N[x]^+$. This is reflected in the syntax presented above, which contains the atomic form $@N[x]^+$ and stratifies compound expressions c and element expressions e to disallow expressions with attributes to appear under the repetition operator. We call this restriction *stratification*. (We do not formalize in a way that introduces a single syntax and imposes a restriction since explicit stratification is easier to present our boolean and inclusion algorithms.) The reason for introducing it is as follows. First, with the restriction, we may have repetitions of concatenations involving attributes, e.g., $(@N[x]\,@N'[y])^+$. However, this would grant a power to grammars for counting the number of occurrences of attributes. Although an approach directly dealing with this has been pursued by other researchers [30], we choose to disallow such expressions since we have not found

[1] If we consider only element constraints, grammars defined here are essentially *regular hedge grammars* [4] or *regular expression types* [20].

[2] We have chosen this design not only for simplifying our boolean and inclusion test algorithms, but also for enabling our validation algorithm presented elsewhere [16] based on automata with both element and attribute transitions. It is also worth remarking that RELAX NG has adopted this restriction. However, a number of different design choices could be taken. One of the reviewers suggested to drop the restriction on expressions but instead exclude values with repeated attributes. Another proposed to interpret $@N_1[x]^*\,@N_2[y]^*$ as $@N_1[x]^*\,@(N_2 \setminus N_1)[y]^* \mid @(N_1 \setminus N_2)[x]^*@N_2[y]^*$. The latter is only a syntax sugar and therefore we may adopt it without any technical difficulty. However, in general, each design choice requires a fair amount of formal study and we have not undertaken to pursue any.

$$\frac{\forall i.\ (a_i \in N \quad G \vdash v_i \in G(x)) \qquad k \geqslant 1 \qquad a_i \neq a_j \text{ for } i \neq j}{G \vdash @a_1[v_1] \ \ldots \ @a_k[v_k] \in \vdash lu@N[x]}\ \text{T-AttRep} \qquad \frac{a \in N \qquad G \vdash v \in G(x)}{G \vdash @a[v] \in @N[x]}\ \text{T-Att}$$

$$\frac{a \in N \qquad G \vdash v \in G(x)}{G \vdash a[v] \in N[x]}\ \text{T-Elm} \qquad \frac{G \vdash v \in c_1 \qquad \text{or} \qquad G \vdash v \in c_2}{G \vdash v \in c_1 \mid c_2}\ \text{T-Alt} \qquad \frac{}{G \vdash \varepsilon \in \varepsilon}\ \text{T-Eps}$$

$$\frac{G \vdash v_1 \in c_1 \qquad G \vdash v_2 \in c_2}{G \vdash v_1\, v_2 \in c_1\, c_2}\ \text{T-Cat} \qquad \frac{\forall i.\ G \vdash v_i \in e \qquad k \geqslant 1}{G \vdash v_1 \ \ldots \ v_k \in \vdash lue}\ \text{T-Plu}$$

Fig. 1. Semantics.

any useful application; we prefer simplicity instead. Then, what a repetition can now contain is a union, another repetition, or an attribute. A repetition of a union can always be rewritten by using only repetitions without unions, e.g., $(@N[x] \mid @N'[y])^+$ can be rewritten as $@N[x]^+ \mid @N[x]^*\ @N'[y]^+$. (A concrete rewriting procedure is omitted here but would not be so complicated.) So we drop this ability to simplify our formalization. Finally, a repetition of a repetition can be collapsed to a single repetition. [3]

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 rules in Fig. 1. Note that rules T-Alt and T-Cat treat alternation and concatenation both in compound expressions and element expressions.

For example, under the grammar

$$G(x) = \varepsilon,$$

the values conforming to the compound expression $(a[x] \mid @a[x])\ (b[x] \mid @b[x])$ are the following four:

$$a[\varepsilon]\, b[\varepsilon]$$
$$@a[\varepsilon]\, b[\varepsilon]$$
$$a[\varepsilon]\, @b[\varepsilon]$$
$$@a[\varepsilon]\, @b[\varepsilon]$$

Using the notations introduced in the end of Section 2.2, the above four can each be expanded as follows:

$$\langle\ \emptyset \qquad\qquad ,\ \langle a, v\rangle\, \langle b, v\rangle\ \rangle$$
$$\langle\ \{\langle a, v\rangle\} \qquad ,\ \langle b, v\rangle \qquad\ \rangle$$
$$\langle\ \{\langle b, v\rangle\} \qquad ,\ \langle a, v\rangle \qquad\ \rangle$$
$$\langle\ \{\langle a, v\rangle, \langle b, v\rangle\}\ ,\ \varepsilon \qquad\ \rangle$$

where v is the value $\langle \emptyset, \varepsilon \rangle$.

The *language* $\mathcal{L}_G(c)$ of an expression c under a grammar G is defined as $\{v \mid G \vdash v \in c\}$. Given two pairs (c_1, G_1) and (c_2, G_2) of expression and grammar, we define their intersection as a pair (c, G) where $\mathcal{L}_G(c) = \mathcal{L}_{G_1}(c_1) \cap \mathcal{L}_{G_2}(c_2)$, similarly, their difference as (c, G) where $\mathcal{L}_G(c) = \mathcal{L}_{G_1}(c_1) \setminus \mathcal{L}_{G_2}(c_2)$, and their inclusion as the predicate that is true iff $\mathcal{L}_{G_1}(c_1) \subseteq \mathcal{L}_{G_2}(c_2)$. In the next sections, we present algorithms for computing these.

3. Boolean and inclusion algorithms

In this section, we present our algorithms for intersection, difference, and inclusion for attribute–element grammars. The proofs of the theorems shown in this section appear in Appendix B.

3.1. Partitioning

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

[3] RELAX NG adopts essentially the same restriction. Specifically, concatenation and repetition are prohibited in a repetition if it contains an attribute.

$$(@a[x] \mid a[x])\,(@b[x] \mid b[x]) \quad \cap \quad @a[y]\,(@b[y] \mid b[y]) \tag{1}$$

How can we calculate this intersection? In a naive algorithm, we may separate constraints on attribute sets and those on element sequences and thereby apply conventional techniques for each kind of constraint. More specifically, we may repeatedly use distributivity of concatenation over union until we only have a union of sequences each of which consists of attribute-only and element-only expressions. From the formula (1), we obtain the following:

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

Then, we can now easily compute the result by taking each pair of clauses on both sides and computing independently the intersection of the attribute-only parts and that of the element-only parts. (We will describe the last step in more detail since it becomes a part of our algorithm.)

However, as one can immediately see, this naive 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 [29]. We will discuss the difference of our work from his in Section 5.

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 $(c_1\,(c_2\,\varepsilon)\,)\,c_3$ 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. To formally express this, we identify expressions up to the relation $\equiv$ defined as follows: $\equiv$ is the smallest congruence relation including the following [4]:

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

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

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

[4] In implementation, it is desirable to represent equal expressions (in terms of $\equiv$) by an identical data structure. For example, we may represent an expression by a pair of a bag of expressions and a list of expressions (e.g., represent $(a[x]\,((@b[x]\,c[x])\,\varepsilon))\,@d[x]^+$ by the pair of the bag of $@b[x]$ and $@d[x]^+$ and the list of $a[x]$ and $c[x]$).

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 $0 < \mathbf{width}(c_i') < \mathbf{width}(c_i)$ for some i. Here, the function **width** counts the number of expressions that are concatenated at the top level (except ε). That is, $\mathbf{width}(\varepsilon) = 0$, $\mathbf{width}(c_1\, c_2) = \mathbf{width}(c_1) + \mathbf{width}(c_2)$, and $\mathbf{width}(c) = 1$ if $c \neq \varepsilon$ and $c \neq c_1\, c_2$. (Note that $\equiv$ preserves **width**.) This properness will be used for ensuring the boolean and inclusion algorithms to make a progress.

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 pairwise 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.

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 Fig. 2. We apply the rules from top to bottom, that is, earlier rules have higher priority. Also, we assume that there is a deterministic strategy (not specified here) for choosing a proper partition among possibly multiple possibilities in rule 7 and for choosing a union $(c_2 \mid c_3)$ from a sequence in rule 8. [5] The base cases are handled by rules 1–6, where each of the arguments is either an element expression (as indicated by the metavariables e or f) or a single- or multi-attribute expression. In rule 1, where both arguments are element expressions, we pass them to another intersection function $\mathbf{inter}^{\mathrm{reg}}$ specialized to element expressions. This function will be explained below. Rules 2 and 3 return $\emptyset$ since the argument expressions obviously denote disjoint sets. Rules 4–6 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 4–6, normalization ensures that the name sets in the given expressions are equal.) The inductive cases are handled by rules 7 and 8. Rule 7 applies the partitioning technique already explained. [6] Rule 8 simply expands one union form appearing in the argument expressions. Note that rules 7 and 8 overlap, that is, in some cases, both rules could be applied. The first example shown in Section 3.1 indeed illustrated the case where we could apply either distributivity (rule 8) or partitioning (rule 7). However, since rule 7 has higher priority, we always perform partitioning in such a case.

[5] What strategy to use for these could have effects on efficiency. However, our current implementation does not use any clever strategy, yet seems to yield a reasonable performance.

[6] In our current formalism, partitioning is applied only when there is at least one attribute in both arguments. However, there is no fundamental problem in allowing pure element expressions to be partitioned and that would in fact make the algorithm more efficient in some cases.

1) $\mathbf{inter}(e, f) = \mathbf{inter}^{\mathrm{reg}}(e, f)$
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) $\mathbf{inter}(c_1\ (c_2 \mid c_3)\ c_4, d) = \mathbf{inter}(c_1\ c_2\ c_4, d) \mid \mathbf{inter}(c_1\ c_3\ c_4, d)$

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

Fig. 2. Intersection algorithm.

We demonstrate our algorithm using our first example of the intersection operation. The intersection of $(@a[x] \mid a[x])\ (@b[x] \mid b[x])$ and $@a[y]\ (@b[y] \mid b[y])$ is computed as follows:

$$
\begin{aligned}
&\mathbf{inter}((@a[x] \mid a[x])\ (@b[x] \mid b[x]),\ @a[y]\ (@b[y] \mid b[y])) \\
&= \mathbf{inter}((@a[x] \mid a[x]),\ @a[y]), \\
&\qquad \mathbf{inter}((@b[x] \mid b[x]),\ (@b[y] \mid b[y])) && \text{(by rule 7)} \\
&= (\mathbf{inter}(@a[x],\ @a[y]) \mid \mathbf{inter}(a[x],\ @a[y])), && \text{(by rule 8)} \\
&\qquad (\mathbf{inter}(@b[x],\ @b[y]) \mid \mathbf{inter}(@b[x],\ b[y]) \mid \\
&\qquad\quad \mathbf{inter}(b[x],\ @b[y]) \mid \mathbf{inter}(b[x],\ b[y])) \\
&= (@a[\langle x, y\rangle] \mid \mathbf{inter}(a[x],\ @a[y])), && \text{(by rule 5)} \\
&\qquad (@b[\langle x, y\rangle] \mid \mathbf{inter}(@b[x],\ b[y]) \mid \\
&\qquad\quad \mathbf{inter}(b[x],\ @b[y]) \mid \mathbf{inter}(b[x],\ b[y])) \\
&= (@a[\langle x, y\rangle] \mid \emptyset),\ (@b[\langle x, y\rangle] \mid \emptyset \mid \emptyset \mid \mathbf{inter}(b[x],\ b[y])) && \text{(by rule 2)} \\
&= (@a[\langle x, y\rangle] \mid \emptyset),\ (@b[\langle x, y\rangle] \mid \emptyset \mid \emptyset \mid b[\langle x, y\rangle]) && \text{(by rule 1)}
\end{aligned}
$$

(By an obvious optimization, the last expression could further be rewritten as $@a[\langle x, y\rangle],\ (@b[\langle x, y\rangle] \mid b[\langle x, y\rangle])$.)

Termination of this algorithm may seem mysterious at first sight since, while rule 7 decrements the widths of the given expressions, rule 8 may increase them. However, observe that the number of times that rule 8 can be applied from given expressions is finite. More precisely, rule 8 decreases the number of clauses resulting from fully expanding the expression by distributivity and, since rule 7 does not increase this number, we can guarantee the termination. (See Appendix B.2 for a formal treatment.)

Also, the presented rules cover all the cases. To see this, first let us view each given expression as a sequence of the form $c_1 \ldots c_k$ where each c_i is either a multi-attribute $@N[x]^+$, a single-attribute $@N[x]$, a union $c_1 \mid c_2$, 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. We apply rule 7 as often as possible, but rule 8 can always serve as fall backs. (One might think that cases like $\varepsilon\ (c_1 \mid c_2)\ \varepsilon$ 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-attribute and element expressions. The case that both sequences have width zero or one is handled by rules 1–6. The remaining case is therefore that either sequence has width two or more. If the two expressions contain only element expressions, then rule 1 applies. Otherwise, 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 $\mathbf{inter}^{\mathrm{reg}}$ 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 $(Q, q^{\mathrm{init}}, Q^{\mathrm{fin}}, \delta)$ where Q is a finite set of *states*, $q^{\mathrm{init}} \in Q$ is an *initial state*, $Q^{\mathrm{fin}} \subseteq Q$ is a set of *final states*, and $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation* [15]. 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 $\mathbf{inter}^{\mathrm{reg}}$ function.

The product construction for element automata (used for the (2) part of $\mathbf{inter}^{\mathrm{reg}}$) 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 $M_i = (Q_i, q_i^{\mathrm{init}}, Q_i^{\mathrm{fin}}, \delta_i)$ 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 $(Q_1 \times Q_2, \langle q_1^{\mathrm{init}}, q_2^{\mathrm{init}}\rangle, Q_1^{\mathrm{fin}} \times Q_2^{\mathrm{fin}}, \delta)$ 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 $\mathbf{inter}^{\mathrm{reg}}$ can be computed in a polynomial time since each of the three steps is polynomial [15]. [7]) The lower bound of the intersection problem (and also the other problems that we consider later) remains to be an open question.

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 $@N[x]$ where N is infinite, and multi-attributes $@N[x]^+$ where N is infinite. 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, since $(@\mathcal{N}[\mathbf{any}]^+, @\mathcal{N}[\mathbf{any}])$ is disallowed by the no-overlapping-attribute restriction (Section 2.3). 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. An apparently plausible expression $(@N[\mathbf{any}]^*, @N[\overline{x}])$ is forbidden by, again, the no-overlapping-attribute restriction. The expression $@N[\overline{x}]^+$ is also 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 finite [8]: when $N = \{a_1, \ldots, a_k\}$

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

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. We call this restriction *single-name-singleton-content*. 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 (this mapping will be modified when we will normalize the grammar before taking a difference):

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

---

[7] The third step of $\mathbf{inter}^{\mathrm{reg}}$ takes a cubic time in the size of the given automaton *if* the resulting expression is represented as a dag. If we fully unfold the dag to a flat expression, the expression has an exponential size and hence the step takes also an exponential time.

[8] RELAX NG adopts this restriction of finiteness.

We can still represent the case where N is finite and the content is not **any**: when $N=\{a_1,\ldots,a_k\}$, we rewrite $@N[x]^+$ by

$$\Big|_{\emptyset\neq\{i_1,\ldots,i_p\}\subseteq\{1,\ldots,k\}} @a_{i_1}[x]\ldots @a_{i_p}[x].$$

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. We call this restriction *multi-name-universal-content*.

These restrictions do not break closure under intersection. This can easily be shown since rules 5 and 6 preserve the single-name-singleton-content restriction and rule 4 preserves the multi-name-universal-content restriction.

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:

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

First of all, note that each of the left expressions 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 $@a[z]\ @b[z]$ from the above result. Performing a similar subtraction, we obtain

$$\begin{array}{ll}
(@a[x]\setminus(@a[y]\mid @a[z]))\ @b[x] & \\
\mid (@a[x]\setminus @a[y]) & (@b[x]\setminus @b[z]) \\
\mid (@a[x]\setminus @a[z]) & (@b[x]\setminus @b[y]) \\
\mid @a[x] & (@b[x]\setminus(@b[y]\mid @b[z]))
\end{array}$$

Thus, the original goal of difference has reduced to the combination of the subgoals of difference. Then, the result of the first difference $(@a[x]\setminus(@a[y]\mid @a[z]))$ (similarly for the other), should be a single-attribute with $@a$ whose content is the difference between the variable x and the union of the variables y and z. For representing such union symbolically, we use a usual technique of subset construction.

Formally, 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 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 Fig. 3. Similar to the intersection algorithm, we give higher priority to earlier rules. (Here, D ranges over sets of compound expressions and $\uplus$ is the disjoint union.) The base cases are handled by rules 1–5. As before, when all the arguments are element expressions, rule 1 passes them to the difference function $\mathbf{diff}^{\mathrm{reg}}$ (explained below). Rules 2, 2′, 3, 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 rules 6–8. 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 $I\subseteq\{1,\ldots,k\}$" on the right-hand side is a generalization of the discussion made in the beginning of this section.

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

Fig. 3. Difference algorithm.

This "subsetting" technique has repeatedly been used in the literature. Interested readers are referred to [19,18,13]. Note again that the rules overlap each other. However, as mentioned before, we apply earlier rules as often as possible, in particular, we always try to partition expressions before resorting to expanding them. The termination of this algorithm is guaranteed since (1) rules 2, 2′, 3, and 3′ decrement the cardinality of the second argument of **diff**, (2) rule 6 decrements the widths of the given expressions, and (3) the number of times that a given expression can be expanded by rule 7 or 8 is finite (more precise, as in intersection, rule 7 and 8 decrease the number of clauses resulting from fully expanding the expression by distributivity).

The function $\mathbf{diff}^{\mathrm{reg}}$ is analogous to the function $\mathbf{inter}^{\mathrm{reg}}$ 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 $M_i = (Q_i, q_i^{\mathrm{init}}, Q_i^{\mathrm{fin}}, \delta_i)$ 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 $(Q, q^{\mathrm{init}}, Q^{\mathrm{fin}}, \delta)$ on $\{N[\langle x_1, Y_2\rangle] \mid N \in S, x_1 \in X_1, Y_2 \subseteq X_2\}$ where

$$
\begin{array}{lll}
Q & = Q_1 \times \mathcal{P}(Q_2) & \\
q^{\mathrm{init}} & = \langle q_1^{\mathrm{init}}, \{q_2^{\mathrm{init}}\}\rangle & \\
Q^{\mathrm{fin}} & = \{\langle q_1, P\rangle \mid q_1 \in F_1 \text{ and } P \cap F_2 = \emptyset\} & \\
\delta & = \{\ (\langle q_1, P\rangle, N[\langle x_1, Y\rangle], \langle q'_1, P'\rangle) \mid & \\
 & \qquad (q_1, N[x_1], q'_1) \in \delta_1 \text{ and} & \\
 & \qquad Y \subseteq \{y \mid (p, N[y], p') \in \delta_2,\ \ p \in P\} \text{ and} & \\
 & \qquad P' = \{p' \mid (p, N[y], p') \in \delta_2,\ \ p \in P,\ \ y \notin Y\}\} &
\end{array}
$$

We demonstrate this algorithm using the first example of the difference operation shown in the beginning of Section 3.3.2. The difference between $@a[x]\ @b[x]$ and $(@a[y]\ @b[y] \mid @a[z]\ @b[z])$ is computed as follows:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{diff}(@a[x]@b[x], \{(@a[y]@b[y] \mid @a[z]@b[z])\})} \\
= \mathbf{diff}(@a[x]@b[x], \{@a[y]@b[y], @a[z]@b[z]\}) & \text{(by rule 8)} \\
= \mathbf{diff}(@a[x], \{@a[y], @a[z]\})\mathbf{diff}(@b[x], \emptyset) & \\
\quad \mid \mathbf{diff}(@a[x], \{@a[y]\})\mathbf{diff}(@b[x], \{@b[z]\}) & \\
\quad \mid \mathbf{diff}(@a[x], \{@a[z]\})\mathbf{diff}(@b[x], \{@b[y]\}) & \\
\quad \mid \mathbf{diff}(@a[x], \emptyset)\mathbf{diff}(@b[x], \{@b[y], @b[z]\}) & \text{(by rule 6)} \\
= @a[\langle x, \{y, z\}\rangle]@b[\langle x, \emptyset\rangle] & \\
\quad \mid @a[\langle x, \{y\}\rangle]@b[\langle x, \{z\}\rangle] & \\
\quad \mid @a[\langle x, \{z\}\rangle]@b[\langle x, \{y\}\rangle] & \\
\quad \mid @a[\langle x, \emptyset\rangle]@b[\langle x, \{y, z\}\rangle] & \text{(by rule 4)}
\end{array}
$$

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 **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 rule 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.

Traditional formal language theories typically study complementation rather than difference. One might wonder why we do not do the same. The reason is that, even if we consider complementation, its subcomputation needs to calculate differences anyway. For example, suppose that we have an attribute-only expression $c\,d$ where $N = \mathbf{att}(c)$. Then, we would compute the complementation $\overline{c\,d}$ by

$$(@N[\mathbf{any}]^* \setminus c)\,d \quad | \quad c\,(@\overline{N}[\mathbf{any}]^* \setminus d) \quad | \quad @\mathcal{N}[\mathbf{any}]^*\mathcal{N}[\mathbf{any}]^+$$

which uses differences. (Note that it is wrong to answer $\overline{c}\,d \mid c\,\overline{d} \mid @\mathcal{N}[\mathbf{any}]^*\mathcal{N}[\mathbf{any}]^+$ since the names contained in $\overline{c}$ may overlap with those in d.)

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 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 *quasi-difference*.[9]

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 quasi-difference 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**.

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

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:

$$@N[x] \quad \setminus \quad (@N[y_1] \mid \ldots \mid @N[y_k])$$

which is equivalent to

$$@N[x \setminus (y_1 \mid \ldots \mid y_k)].$$

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

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

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:

$$@N[x]^+ \quad \setminus \quad (@N[y_1]^+ \mid \ldots \mid @N[y_l]^+) \tag{4}$$

[9] We could obtain a more efficient algorithm by combining quasi-difference and emptiness check together so that we can skip the rebuilding of a whole grammar. This approach is taken by a "top-down" algorithm for tree automata inclusion [19].

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

Fig. 4. Quasi-difference algorithm: in addition, we transfer rules 1–3, 6–8 of **diff** (with the function symbol **diff** replaced by **qdiff**) to here. (Rule 1 still uses the function $\mathbf{diff}^{\mathrm{reg}}$ for element expressions.)

The second step transforms this to the following:

$$@a_1[x \setminus y_1]^+ \ \ldots \ @a_l[x \setminus y_l]^+ \tag{5}$$

where $a_1, \ldots, a_l$ 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

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

where

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

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:

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

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.

Formally, let grammars F on X and G on Y be given and have been normalized simultaneously. Also, as mentioned in the last paragraph, we assume that each multi-attribute expression has either an infinite name set or a singleton name set. (An arbitrary multi-attribute expression can be converted to a form conforming to this restriction by using the name set normalization described in Appendix A.) The "quasi-difference" between F and G is a grammar H on $X \times \mathcal{P}(Y)$ that satisfies

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

for all $x \in X$ and $Z \subseteq Y$. The function **qdiff** 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 **qdiff** is given in Fig. 4. Here, we assume that N in the form $@N[x]^+$ is an infinite set. We can prove the following expected properties.

Theorem 3. *Let* $H(\langle x, Z\rangle) = \mathbf{qdiff}(F(x), \{G(y) \mid y \in Z\})$. *Then* $\mathbf{qdiff}(c, D) = b$ *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.

To examine the emptiness of a given grammar H, we first check the emptiness of $H(x)$ for each variable x, assuming the emptiness of all variables. We then "update" this assumption if $H(x)$ is non-empty for some variable x. By repeating the emptiness test and assumption update, we can eventually determine whether each $H(x)$ is empty or not. This idea can be formalized as follows. We compute a series of (total) functions $\phi_0, \phi_1, \ldots$ from the variables in $\mathbf{dom}(H)$ to booleans, as defined below, and stop this computation when $\phi_n(x) = \phi_{n-1}(x)$ for all x.

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

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

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

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 (similar to tree automata emptiness [10]).

4. Implementation techniques

As already discussed, the worst-case complexities of the algorithms presented above are quite bad—the intersection algorithm is exponential and the difference and inclusion algorithms are double exponential. Our approach is to cope with these high complexities by optimization techniques that work for typical inputs. One of such techniques is partitioning (already presented), which avoids the need for fully expanding expressions, thus dealing with one of two exponential factors. The other factor is subset construction, for which there are several known techniques. Below, we briefly explain some of the techniques (which we used in our implementation). (More discussions can be found in [19,18].)

4.1. Top-down strategy

The intersection algorithm presented above takes grammars F (on X) and G (on Y) and calculates the product of F and G for the whole domain $X \times Y$. However, the actual algorithm takes "start variables" x_0 and y_0 in addition and the useful part of the output grammar is the one that defines the reachable variables from the output start variable $\langle x_0, y_0\rangle$. Our observation is that such reachable variables are typically much fewer than the whole $X \times Y$. Exploiting this, we construct the output grammar *lazily* from the start variable $\langle x_0, y_0\rangle$ in a top-down manner. That is, we initialize H to be a grammar just containing the mapping $\langle x_0, y_0\rangle \mapsto \mathbf{inter}(F(x_0), G(y_0))$. We then take any pair $\langle x, y\rangle \in \mathbf{FV}(H)$ that is not yet in the domain of H, and add the mapping $\langle x, y\rangle \mapsto \mathbf{inter}(F(x), G(y))$. We repeat this until H becomes self-contained. The same technique can be used in the algorithms for product automata, difference grammars, difference automata, and quasi-difference grammars.

4.2. Sharing

It is quite common that the intersection algorithm encounters, during its computation, a pair of the same expression $c \cap c$. Since the result is trivially c itself whatever c is, we would like to somehow exploit this fact. However, since expressions can be nested and refer to other variables, sameness is a bit complicated. For example, in the following

grammars F and G:

$$F(x_1) = a[x_2] \qquad F(x_2) = \varepsilon$$
$$G(y_1) = a[y_2] \qquad G(y_2) = \varepsilon$$

x_1 and y_1 are the "same", but detecting this requires a bit of work. In addition, even if we can detect it, we need to copy the whole structure reachable from x_1 to the output grammar.

To deal with this issue, we employ one global grammar. All input and output grammars are parts of it, and, furthermore, these grammars can share some variables. We modify our intersection algorithm so that it takes only two start variables, proceeds the computation using the global grammar, and adds a new mapping to it whenever necessary. Now, the intersection of x and x is immediately x. Similarly, difference between x and Y with $x \in Y$ results in a variable $x_\emptyset$ that is assigned to $\emptyset$.

4.3. Other techniques

Since the function **inter** (and similarly for **diff** and **qdiff**) often computes the intersection of the same pair of expressions repeatedly, memoizing the previous computations and reusing the results later are quite helpful. Another technique is to use rules that are specialized to some fixed but frequently used expressions, such as **any** and $\emptyset$. For example, we can use the special rules $x \cap \mathbf{any} = x$ and $x \setminus Y = x_\emptyset$ when $\mathbf{any} \in Y$.

5. 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 [14].) Buneman and Pierce have investigated record types with the union operator [6]. Their system does not, however, have any mechanism similar to our multi-attribute expressions or recursion. Frisch et al. [13] 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 [21] and its descendent DSD2 [23], designed by Klarlund et al., are capable of expressing the kinds of attribute–element interdependencies discussed here by their special constraint mechanisms (separate from regular expressions to describe content models). Closure properties of these schema languages have not yet been clear, though a fragment of DSD2 (eliding pointers and uniqueness facilities) appears to be closed.

In his unpublished work, Vouillon has considered an algorithm for checking the inclusion relation between shuffle expressions [29]. 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 to schemas is validation. There have been several validation algorithms proposed for attribute–element constraints. One is designed and implemented by Clark [8] based on derivatives of regular expressions [5,1]. Another is presented by Hosoya and Murata using so-called attribute–element automata [16].

6. Conclusion

In this paper, we have presented our intersection, difference, and inclusion algorithms. We have used these algorithms in our implementation of the typed XML processing language XDuce [17]. For the examples that we have tried, our algorithms incorporating the implementation techniques presented in Section 4 seem to have a reasonable performance.

For future possibilities, we wish to figure out the lower bounds of the presented algorithmic problems. However, we currently feel that most of operations performed in our algorithms that are expensive in the worst case (e.g., normalization) are unavoidable and therefore the complexities of the algorithms are unlikely to be improved. As a

separate topic, instead of studying closure operations on grammars as in the present paper, we could investigate those on tree automata with "attribute transitions". While it might add a complication arising from commutativity of attribute constraints and their partitioning, it could make a simplification by avoiding the use of conversion back from automata to regular expressions.

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 referees of PLAN-X workshop and Theoretical Computer Science journal, who helped us in 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.

Appendix A. Name set normalization

Let $\{N_1, \ldots, N_k\}$ be the set of name sets appearing in given grammars. (When we are given two grammars, this set includes all the names in both grammars.) From this, we generate a set of disjoint name sets by the following **shred** function. (Here, $\uplus$ is the disjoint union.)

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

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

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 $\{N_1, \ldots, N_k\}$ 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 $N \cap (\cup S) = \emptyset$ is taken in most of the time.

We can easily show the following expected properties.

Lemma 1. *Let* $S = \mathbf{shred}(\{N_1, \ldots, N_k\})$.
(1) *For all* $N, N' \in S$, *either* $N = N'$ *or* $N \cap N' = \emptyset$.
(2) *For all* N_i, *there is uniquely* $S' \subseteq S$ *such that* $N_i = \cup S'$.

Having computed a shredded set $S = \mathbf{shred}(\{N_1, \ldots, N_k\})$, 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:

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

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

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

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 with each other. In our experience, $@N[x]^+$ almost always appears in the form $@N[x]^+ \mid \varepsilon$ since the user typically writes zero or more repetitions rather than one or more. The following lemma can easily be proved.

Lemma 2. *Let G be a grammar on X and G' be the normalization of G. Then, $G \vdash v \in G(x)$ iff $G' \vdash v \in G'(x)$ for all v and $x \in X$.*

The worst-case complexity of the whole normalization procedure is as follows. Suppose that there are n occurrences of atomic form ($N[x]$, $@N[x]^+$, or $@N[x]$) 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.

Appendix B. Correctness of the Boolean and inclusion algorithms

B.1. Element automata

From the standard automata theory, the following is well known.

Proposition 1 (*Hopcroft and Ullman [15]*). (1) *There is an algorithm* compile *that constructs an automaton M from a given regular expression e such that $L(M) = L(e)$.*
(2) *There is an algorithm* decompile *that constructs a regular expression e from a given automaton M such that $L(e) = L(M)$.*

Given a grammar G on X, an *element automaton* is an automaton on $S \times X$, where S is the set of name sets. We present the semantics of an element automaton M w.r.t. G by the relation $G \vdash v \in M$ defined as follows:

$$\frac{a_i \in N_i \qquad G \vdash v_i \in G(x_i) \qquad N_1[x_1] \ldots N_k[x_k] \in L(M)}{G \vdash a_1[v_1] \ldots a_k[v_k] \in M}$$

Then, we can easily show that this semantics satisfies the following properties.

Corollary 1. (1) *$G \vdash v \in e$ iff $G \vdash v \in \mathrm{compile}(e)$.*
(2) *$G \vdash v \in M$ iff $G \vdash v \in \mathrm{decompile}(M)$.*

B.2. Intersection

We inductively define the height $|v|$ of a value v as follows:

$$\begin{array}{ll} |a[v]| & = |v| + 1 \\ |@a[v]| & = |v| + 1 \\ |v_1\, v_2| & = \max(|v_1|, |v_2|) \\ |\varepsilon| & = 1 \end{array}$$

Let H be the "intersection" grammar constructed from two grammars F and G as in Section 3.2. By using the standard proof technique, we can show the following expected property of product element automata.

Lemma 3. *Let M be the product automaton of M_1 and M_2. Let v be any value. Suppose that, for any x, y, w with $|w| < |v|$,*

$H \vdash w \in H(\langle x, y\rangle)$ *iff* $F \vdash w \in F(x)$ *and* $G \vdash w \in G(y)$.

Then, we have

$H \vdash v \in M$ *iff* $F \vdash v \in M_1$ *and* $G \vdash v \in M_2$.

Corollary 2. *Let* $\mathbf{inter}^{\mathrm{reg}}(e_1, e_2) = e$. *With the same assumption as Lemma* 3, *we have*

$H \vdash v \in e$ *iff* $F \vdash v \in e_1$ *and* $G \vdash v \in e_2$.

The following technical lemma shows that, if a value can be partitioned by disjoint sets of attributes and elements, then such a partition is unique.

Lemma 4. *Given a value v, if $v = u\, w = u'\, w'$ where $(\mathbf{elm}(u) \cup \mathbf{elm}(u')) \cap (\mathbf{elm}(w) \cup \mathbf{elm}(w')) = \emptyset$ and $(\mathbf{att}(u) \cup \mathbf{att}(u')) \cap (\mathbf{att}(w) \cup \mathbf{att}(w')) = \emptyset$, then $u = u'$ and $w = w'$.*

Proof. Let $v = \langle \alpha, \beta\rangle$. Also, let $u = \langle \alpha_1, \beta_1\rangle$ and $w = \langle \alpha_2, \beta_2\rangle$; similarly $u' = \langle \alpha'_1, \beta'_1\rangle$ and $w' = \langle \alpha'_2, \beta'_2\rangle$. Suppose $u \neq u'$. Then, either $\alpha_1 \neq \alpha'_1$ or $\beta_1 \neq \beta'_1$.

- When $\alpha_1 \neq \alpha'_1$, either $\alpha_1 \setminus \alpha'_1 \neq \emptyset$ or $\alpha'_1 \setminus \alpha_1 \neq \emptyset$. Therefore, $\alpha_1 \cap \alpha'_2 \neq \emptyset$ or $\alpha_2 \cap \alpha'_1 \neq \emptyset$. Either case contradicts the condition $(\mathbf{att}(u) \cup \mathbf{att}(u')) \cap (\mathbf{att}(w) \cup \mathbf{att}(w')) = \emptyset$.
- When $\beta_1 \neq \beta'_1$, either (1) $\beta_1 = \beta'_1\gamma$ and $\beta'_2 = \gamma\beta_2$ for some non-empty γ or (2) $\beta'_1 = \beta_1\gamma$ and $\beta_2 = \gamma\beta'_2$ for some non-empty γ. Either case contradicts the condition $(\mathbf{elm}(u) \cup \mathbf{elm}(u')) \cap (\mathbf{elm}(w) \cup \mathbf{elm}(w')) = \emptyset$. □

In the subsequent proofs, we use the following function **weight** from compound expressions to integers:

$$\begin{array}{ll} \mathbf{weight}(c_1\, c_2) & = \mathbf{weight}(c_1)\mathbf{weight}(c_2) \\ \mathbf{weight}(c_1 \mid c_2) & = \mathbf{weight}(c_1) + \mathbf{weight}(c_2) \\ \mathbf{weight}(c) & = 1 \quad \text{if } c \neq c_1\, c_2 \text{ and } c \neq c_1 \mid c_2 \end{array}$$

This function computes the number of clauses resulting from fully expanding the given expression by distributivity. For example, $\mathbf{weight}((a[x] \mid @a[y])\ (\varepsilon \mid b[z])) = 4$.

We use both **width** and **weight** functions for the measure of induction in the proof of Theorem 1. When we decompose an expression c into $c_1\, c_2$ by proper partition, each c_i decreases **width** while it either keeps or decreases **weight** (never increases it). When we expand an expression $c\ (c_1 \mid c_2)\ c'$ to $(c\, c_1\, c') \mid (c\, c_2\, c')$, each $c\, c_i\, c'$ decreases **weight** while it may increase **width** (e.g., $\mathbf{width}((a[x]\, b[x]) \mid c[x]) = 1 < \mathbf{width}(a[x]\, b[x]) = 2$). For this reason, the measure gives **weight** higher priority than **width**.

The following theorem (shown in Section 3.2) shows the correctness of the function **inter**.

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 $G \vdash v \in d$.*

Proof. Let $\mathbf{measure}(v, c, d) = (|v|, \mathbf{weight}(c)+\mathbf{weight}(d), \mathbf{width}(c)+\mathbf{width}(d))$. By induction on the lexicographic order of $\mathbf{measure}(v, c, d)$.

Case: $c = e$ (i.e., $\mathbf{att}(c) = \emptyset$) $\quad d = f$
$b = \mathbf{inter}^{\mathrm{reg}}(e, f)$

The induction hypothesis allows us to use Corollary 2, from which the result follows.

Case: $c = @N[x] \quad \mathbf{att}(d) \cap N = \emptyset \quad b = \emptyset$ or
$c = @N[x]^+ \quad \mathbf{att}(d) \cap N = \emptyset \quad b = \emptyset$

Trivial.

Case: $c = @N[x] \quad d = @N[y]^+ \quad b = @N[\langle x, y\rangle]$

Clearly, v has the form $@a[v']$. By definition, $F \vdash v \in c$ and $G \vdash v \in d$ iff $a \in N$ with $F \vdash v' \in F(x)$ and $G \vdash v' \in G(y)$. By induction hypothesis, $F \vdash v' \in F(x)$ and $G \vdash v' \in G(y)$ if and only if $H \vdash v' \in H(\langle x, y\rangle)$. By combining these, the result follows.

Case: $c = @N[x] \quad d = @N[y] \quad b = @N[\langle x, y\rangle]$

Similar to the previous case.

Case: $c = @N[x]^+ \quad d = @N[y]^+ \quad b = @N[\langle x, y\rangle]^+$

Clearly, v has the form $@a_1[v_1] \ldots @a_k[v_k]$. By definition, $F \vdash v \in c$ and $G \vdash v \in d$ iff $k \geqslant 1$ and $a_i \in N$ with $F \vdash v_i \in F(x)$ and $G \vdash v_i \in G(y)$. By induction hypothesis, $F \vdash v_i \in F(x)$ and $G \vdash v_i \in G(y)$ if and only if $H \vdash v_i \in H(\langle x, y\rangle)$. By combining these, the result follows.

Case: $b = \mathbf{inter}(c_1, d_1)\,\mathbf{inter}(c_2, d_2)$
$(c_1, c_2), (d_1, d_2)$ is a proper partition of c, d

By the definition of partition, the result suffices to show

$$\begin{array}{l} F \vdash v \in c_1\, c_2 \text{ and } G \vdash v \in d_1\, d_2 \\ \text{iff } H \vdash v \in \mathbf{inter}(c_1, d_1)\,\mathbf{inter}(c_2, d_2) \end{array} \tag{B.1}$$

with $(\mathbf{att}(c_1) \cup \mathbf{att}(d_1)) \cap (\mathbf{att}(c_2) \cup \mathbf{att}(d_2)) = \emptyset$ and $(\mathbf{elm}(c_1) \cup \mathbf{elm}(d_1)) \cap (\mathbf{elm}(c_2) \cup \mathbf{elm}(d_2)) = \emptyset$.

We first show the "if" direction of the statement (B.1). Let v satisfy the right-hand side of this statement. Then, there are v_1 and v_2 such that $v = v_1\, v_2$ with $H \vdash v_i \in \mathbf{inter}(c_i, d_i)$ for $i = 1, 2$. Since the partition is proper, $\mathbf{measure}(v, c_i, d_i) < \mathbf{measure}(v, c, d)$ for $i = 1, 2$. This allows us to apply the induction hypothesis and obtain $F \vdash v_i \in c_i$ and $G \vdash v_i \in d_i$ for $i = 1, 2$.

To show the other direction, let v satisfy the left-hand side of (B.1). Then, there are v_1 and v_2 such that $v = v_1\, v_2$ with $F \vdash v_1 \in c_1$ and $F \vdash v_2 \in c_2$; also, there are v'_1 and v'_2 such that $v = v'_1\, v'_2$ with $G \vdash v'_1 \in d_1$ and $G \vdash v'_2 \in d_2$. From Lemma 4, $v_1 = v'_1$ and $v_2 = v'_2$. By the same argument as above, we can apply the induction hypothesis shown above, we obtain $H \vdash v_i \in \mathbf{inter}(c_i, d_i)$ for $i = 1, 2$, from which the result follows.

Case: $c = c_1\,(c_2 \mid c_3)\,c_4$
$b = \mathbf{inter}(c_1\, c_2\, c_4, d) \mid \mathbf{inter}(c_1\, c_3\, c_4, d)$

Clearly $F \vdash v \in c$ if and only if either $F \vdash v \in c_1\, c_2\, c_4$ or $F \vdash v \in c_1\, c_3\, c_4$. Note $\mathbf{measure}(v, c_1\, c_2\, c_4, d) < \mathbf{measure}(v, c, d)$ and, similarly, $\mathbf{measure}(v, c_1\, c_3\, c_4, d) < \mathbf{measure}(v, c, d)$. The result follows from the induction hypothesis.

The other cases corresponding to the symmetric rules can be proved similarly. □

B.3. Difference

Let H be the "difference" grammar constructed from two grammars F and G as in Section 3.3. By combining the standard proof technique for product construction and subset construction, we can show the following:

Lemma 5. *Let M be the difference automaton of M_1 and M_2. Let v be any value. Suppose that, for any x, Y, w with $|w| < |v|$,*

$$H \vdash w \in H(\langle x, Y\rangle) \text{ iff } F \vdash w \in F(x) \text{ and } G \vdash w \notin G(y) \text{ for all } y \in Y.$$

Then, we have
$H \vdash v \in M$ *iff* $F \vdash v \in M_1$ *and* $G \vdash v \notin M_2$.

Corollary 3. *Let* $\mathbf{diff}^{\mathrm{reg}}(e_1, e_2) = e$. *With the same assumption as Lemma* 5, *we have*
$H \vdash v \in e$ *iff* $F \vdash v \in e_1$ *and* $G \vdash v \notin e_2$.

Now, we prove Theorem 2 (given in Section 3.3.1). In the proof, we need a care in considering the measure of induction since we pass a set of expressions (instead of a single expression) as the second argument to the **diff** function. The rules making recursive calls (where let c and D be the two arguments) are (a) ones removing an expression from D (rules 2, 2′, 3, and 3′), (b) one splitting c and the expressions in D by proper partitioning (rule 6), (c) one expanding c (rule 7), and (d) one expanding D (rule 8). What does each kind of rules decrease? It is easy to see that (a) decreases the cardinality of D, (b) decreases the sum of the widths of c and the expressions in D, and (c) decreases the weight of c, that is, the number of expressions resulted from fully expanding c by distributivity. Note that (d) does *not* decrease but retains the sum of the weights of the expressions in D. What this rule decreases is actually *how close D is to the full expansion*. This can formally be expressed by

$$\sum_{d \in D} \mathbf{weight}(d) - |D|.$$

Since (c) and (d) decrease their measures independently, these measures can be combined as

$$\mathbf{weight}(c) + \sum_{d \in D} \mathbf{weight}(d) - |D|.$$

Let us define $m_1(c, D)$ be this formula, $m_2(c, D)$ be $|D|$, and $m_3(c, D)$ be $\mathbf{width}(c) + \sum_{d \in D} \mathbf{width}(d)$.

Now, we need to combine these measures for forming a lexicographic order. The question is which measure is more robust than the others. At first, m_1 may look fragile since it contains the clause $-|D|$. However, we can see that (a) does not decrease m_1 since, when $D' \subseteq D$,

$$\begin{aligned}
&\left(\sum_{d \in D} \mathbf{weight}(d) - |D|\right) - \left(\sum_{d \in D'} \mathbf{weight}(d) - |D'|\right) \\
&\quad = \sum_{d \in D \setminus D'} \mathbf{weight}(d) - (|D \setminus D'|) \\
&\quad \geqslant 0.
\end{aligned}$$

(The last inequation follows from $\mathbf{weight}(d) \geqslant 1$ for any d.) Also, (b) does not since it decreases the cardinality of D to D' *and* each expression d_i' (or d_i'') in D' is resulted from a proper partition of the corresponding expression d_i in D. From this discussion, the measure m_1 is actually the most robust among the three measures. The next most robust is m_2 since it may increase only when m_1 decreases (by rule 8).

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* $d \in D$.

Proof. Let $\mathbf{measure}(v, c, D)$ be

$$\left(|v|,\ \mathbf{weight}(c) + \sum_{d \in D} \mathbf{weight}(d) - |D|,\ |D|,\ \mathbf{width}(c) + \sum_{d \in D} \mathbf{width}(d)\right).$$

By induction on the lexicographic order of $\mathbf{measure}(v, c, D)$:

Case: $c = e$ $\quad D = \{f_1, \ldots, f_k\}$
$b = \mathbf{diff}^{\mathrm{reg}}(e, f_1 \mid \ldots \mid f_k)$
The induction hypothesis allows us to use Corollary 3, from which the result follows.

Case: $D = \{@N[y]^+\} \uplus D'$ or $D = \{@N[y]\} \uplus D'$ $\quad \mathbf{att}(c) \cap N = \emptyset$
$b = \mathbf{diff}(c, D')$

Since c and $@N[y]^+$ are obviously disjoint, the result follows by the induction hypothesis.

Case: $c = @N[x]^+$ or $c = @N[x]$ $\quad N \cap \mathbf{att}(d) = \emptyset$
$b = \mathbf{diff}(@N[y]^+, D')$

Similar to the previous case.

Case: $c = @N[\mathbf{any}]^+ \quad D = \{@N[\mathbf{any}]^+\} \quad b = \emptyset$

Trivial.

Case: $c = @a[x] \quad D = \{@a[y_1], \ldots, @a[y_k]\}$
$b = @a[\langle x, \{y_1, \ldots, y_k\}\rangle]$

By definition, $F \vdash v \in c$ and $G \vdash v \notin @a[y_i]$ for all $i = 1, \ldots, k$ if and only if $v = @a[v']$ with $F \vdash v' \in F(x)$ and $G \vdash v' \notin G(y_i)$ for all $i = 1, \ldots, k$. By the induction hypothesis, the latter is equivalent to saying $H \vdash v \in b$.

Case: $D = \{d_1, \ldots, d_k\}$
$b = \Big|_{I \subseteq \{1,\ldots,k\}} \mathbf{diff}(c', \{d'_i \mid i \in I\}) \; \mathbf{diff}(c'', \{d''_i \mid i \in \{1, \ldots, k\} \setminus I\})$
$(c', c''), (d'_1, d''_1), \ldots, (d'_k, d''_k)$
is a proper partition of $c, d_1, \ldots, d_k$

By the definition of partition, we have $c = c'\, c''$ and $d_i = d'_i\, d''_i$ with $(\mathbf{att}(c') \cup \bigcup_i \mathbf{att}(d'_i)) \cap (\mathbf{att}(c'') \cup \bigcup_i \mathbf{att}(d''_i)) = \emptyset$ and $(\mathbf{elm}(c') \cup \bigcup_i \mathbf{elm}(d'_i)) \cap (\mathbf{elm}(c'') \cup \bigcup_i \mathbf{elm}(d''_i)) = \emptyset$. We first show the "only if" direction. Let $H \vdash v \in b$. Then, there are v' and v'' such that $H \vdash v' \in \mathbf{diff}(c', \{d'_i \mid i \in I\})$ and $H \vdash v'' \in \mathbf{diff}(c'', \{d''_i \mid i \in \{1, \ldots, k\} \setminus I\})$. By the induction hypothesis, we have

$$F \vdash v' \in c' \text{ and } G \vdash v' \notin d'_i \text{ for all } i \in I$$

and

$$F \vdash v'' \in c'' \text{ and } G \vdash v'' \notin d''_i \text{ for all } i \in \{1, \ldots, k\} \setminus I.$$

These imply that $F \vdash v \in c$ and $G \vdash v \notin d_i$ for all $i \in \{1, \ldots, k\}$.

We next show the "if" direction. Let $F \vdash v \in c$ and $G \vdash v \notin d_i$ for all $i \in \{1, \ldots, k\}$. Then, there are v' and v'' such that $v = v'\, v''$ with $F \vdash v' \in c'$ and $F \vdash v'' \in c''$. From Lemma 4, v' and v'' are uniquely determined. Therefore, either $G \vdash v' \notin d'_i$ or $G \vdash v'' \notin d''_i$. Since this holds for all i, there is $I \subseteq \{1, \ldots, k\}$ such that $G \vdash v' \notin d'_i$ for all $i \in I$ and $G \vdash v'' \notin d''_i$ for all $i \in \{1, \ldots, k\} \setminus I$. By the induction hypothesis,

$$H \vdash v' \in \mathbf{diff}(c', \{d'_i \mid i \in I\})$$

and

$$H \vdash v'' \in \mathbf{diff}(c'', \{d''_i \mid i \in \{1, \ldots, k\} \setminus I\}).$$

The desired result $H \vdash v \in b$ follows from these.

Case: $c = c_1\, (c_2 \mid c_3)\, c_4$
$b = \mathbf{diff}(c_1\, c_2\, c_4, D) \mid \mathbf{diff}(c_1\, c_3\, c_4, D)$

The result can be shown straightforwardly by using that

$$F \vdash v \in c_1\, (c_2 \mid c_3)\, c_4 \text{ iff } F \vdash v \in c_1\, c_2\, c_4 \text{ or } F \vdash v \in c_1\, c_3\, c_4$$

plus the induction hypothesis.

Case: $D = \{d_1\, (d_2 \mid d_3)\, d_4\} \uplus D'$
$b = \mathbf{diff}(c, \{d_1\, d_2\, d_4, d_1\, d_3\, d_4\} \uplus D')$

Similar to the previous case. □

B.4. Inclusion

Let H be the "quasi-difference" grammar constructed from two grammars F and G as in Section 3.4. Then, we can prove the following statement. Note that, since H is now the quasi-difference grammar rather than the difference grammar, what we can state is somewhat weaker than Lemma 5. That is, we want to have that, if there is some value accepted by M_1 but not by M_2, then there is some (possibly different) value of the same height accepted by M; and vice versa. We can ensure this if a similar statement holds for the quasi-difference grammar for any values with *smaller* heights.

Lemma 6. *Let M be the difference automaton of M_1 and M_2. Take an arbitrary $h \geqslant 0$. Suppose that, for any $h' < h$, x, and Y,*

$H \vdash v' \in H(\langle x, Y\rangle)$ for some v' with $|v'| = h'$ iff $F \vdash w' \in F(x)$ and $G \vdash w' \notin G(y)$ for all $y \in Y$ for some w' with $|w'| = h'$.

Then, we have

$H \vdash v \in M$ for some v with $|v| = h$ iff $F \vdash w \in M_1$ and $G \vdash w \notin M_2$ for some w with $|w| = h$.

Corollary 4. *Let* $\mathbf{diff}^{\mathrm{reg}}(e_1, e_2) = e$. *With the same assumption as Lemma* 6, *we have*

$H \vdash v \in e$ for some v with $|v| = h$ iff $F \vdash v \in e_1$ and $G \vdash v \notin e_2$ for some w with $|w| = h$.

Theorem 3. *Let* $H(\langle x, Z\rangle) = \mathbf{qdiff}(F(x), \{G(y) \mid y \in Z\})$. *Then* $\mathbf{qdiff}(c, D) = b$ *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.*

Proof. To prove the result, we show the following slightly more general statement:

$\mathbf{qdiff}(c, D) = b$ implies, for any $h \geqslant 0$, that $H \vdash v \in b$ for some v with $|v| = h$ iff $F \vdash w \in c$ and $G \vdash w \notin d$ for all $d \in D$ for some w with $|w| = h$.

Let $\mathbf{measure}(h, c, D)$ be

$$\left(h,\ \mathbf{weight}(c) + \sum_{d\in D} \mathbf{weight}(d) - |D|,\ |D|,\ \mathbf{width}(c) + \sum_{d\in D} \mathbf{width}(d)\right).$$

The proof proceeds by induction on the lexicographic order of $\mathbf{measure}(h, c, D)$. Most of the cases are analogous to the proof of Theorem 2. The exceptions are the following:

Case: $c = e \qquad D = \{f_1, \ldots, f_k\}$
$b = \mathbf{diff}^{\mathrm{reg}}(e, f_1 \mid \ldots \mid f_k)$

The induction hypothesis allows us to use Corollary 4, from which the result follows.

Case: $c = @N[x]$
$D = \{@N[y_1]^+, \ldots, @N[y_l]^+, @N[y_{l+1}], \ldots, @N[y_k]\}$
$b = @N[\langle x, \{y_1, \ldots, y_k\}\rangle]$

We first show the "only if" direction. Let $H \vdash v \in b$. Then, $v = @a[v']$ for some v' with $a \in N$ and $H \vdash v' \in H(\langle x, \{y_1, \ldots, y_k\}\rangle)$. By the induction hypothesis, $F \vdash w' \in F(x)$ and $G \vdash w' \notin G(y_i)$ $(i = 1, \ldots, k)$ for some w' with $|w'| = h - 1$. Let $w = @a[w']$. Then, $F \vdash w \in c$ and $G \vdash w \notin d$ for $d \in D$.

We next show the "if" direction. Let $F \vdash w \in c$ and $G \vdash w \notin d$ for $d \in D$. Then, $w = @a[w']$ where $a \in N$ with $F \vdash w' \in F(x)$ and $G \vdash w' \notin G(y_i)$ $(i = 1, \ldots, k)$ for some w'. By the induction hypothesis, $H \vdash v' \in H(\langle x, \{y_1, \ldots, y_k\}\rangle)$ for some v' with $|v'| = h - 1$. Let $v = @a[v']$. Then, $H \vdash v \in b$. The result follows.

Case: $c = @N[x]^+$
$D = \{@N[y_1]^+, \ldots, @N[y_l]^+, @N[y_{l+1}], \ldots, @N[y_k]\}$
N is infinite
$b = @a_1[\langle x, \{y_1\}\rangle]^+ \ \ldots \ @a_l[\langle x, \{y_l\}\rangle]^+$
$a_i \in N \qquad a_i \neq a_j \quad (i \neq j)$

We first show the "only if" direction. Let $H \vdash v \in b$. Then, $v = @a_1[v_1] \ldots @a_l[v_l]$ for some $v_1, \ldots, v_l$ with $H \vdash v_i \in H(\langle x, \{y_i\}\rangle)$ $(i = 1, \ldots, l)$. By the induction hypothesis, $F \vdash w_i \in F(x)$ and $G \vdash w_i \notin G(y_i)$ for some w_i with $|w_i| = |v_i|$ for $i = 1, \ldots, l$. Take two arbitrary different names b_1 and b_2 from N that are also different from any a_i (which is always feasible because N is infinite). Let $w = @a_1[w_i] \ldots @a_l[w_l]@b_1[w_1]@b_2[w_2]$. Clearly, $|w| = h$ and $F \vdash w \in c$. For each $i = 1, \ldots, l$, we have $G \vdash w \notin @N[y_i]^+$ since $G \vdash w_i \notin G(y_i)$. Finally, for each $i = l + 1, \ldots, k$, we have $G \vdash w \notin @N[y_i]$ since w has at least width two. The result follows.

We next show the "if" direction. Let $F \vdash w \in c$ and $G \vdash w \notin d$ for $d \in D$. Then, w must have the form $@b_1[w_1] \ldots @b_l[w_l]$ where, for $i = 1, \ldots, l$, we have $b_i \in N$ with $F \vdash w_{j_i} \in F(x)$ and $G \vdash w_{j_i} \notin G(y_i)$ for some j_i. By applying the induction hypothesis for $i = 1, \ldots, l$, we obtain that $H \vdash v_i \in H(\langle x, \{y_i\}\rangle)$ for some v_i with $|v_i| = |w_{j_i}|$. Let $v = @a_1[v_1] \ldots @a_l[v_l]$. Then, the result $H \vdash v \in b$ follows. □

Lemma 7. $\mathbf{nemp}(c)\phi_i = \mathbf{true}$ *if and only if* $H \vdash v \in c$ *for some* v *with* $|v| \leqslant i + 1$.

Proof. The proof proceeds by induction on the lexicographic order of $(i, |c|)$.

Case: $c = \varepsilon$

$\mathbf{nemp}(c)\phi_i = \mathbf{true}$ always holds. The result follows from $H \vdash \varepsilon \in \varepsilon$ and $|\varepsilon| = 1 \leqslant i + 1$.

Case: $c = N[x]$

By definition, $\mathbf{nemp}(c)\phi_i = (N \neq \emptyset) \wedge \phi_i(x) = (N \neq \emptyset) \wedge \mathbf{nemp}(H(x))\phi_{i-1}$. By the induction hypothesis, $\mathbf{nemp}(H(x))\phi_{i-1} = \mathbf{true}$ if and only if $H \vdash v' \in H(x)$ for some v' with $|v'| \leqslant i$. By T-ELM, $N \neq \emptyset$ and $H \vdash v' \in H(x)$ if and only if $H \vdash a[v'] \in N[x]$ where $a \in N$. Since $|a[v']| \leqslant i + 1$, the result follows.

Case: $c = @N[x]$ or $c = @N[x]^+$

Similar to the previous case.

Case: $c = c_1 \,|\, c_2$

By definition, $\mathbf{nemp}(c)\phi_i = \mathbf{true}$ if and only if $\mathbf{nemp}(c_1)\phi_i = \mathbf{true}$ or $\mathbf{nemp}(c_2)\phi_i = \mathbf{true}$. By the induction hypothesis, the latter is equivalent to $H \vdash v_1 \in c$ with $|v_1| \leqslant i + 1$ or $H \vdash v_2 \in c$ with $|v_2| \leqslant i + 1$. The result follows by T-OR.

Case: $c = c_1\, c_2$

Similar to the previous case. □

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

Proof. The "only if" direction immediately follows from Lemma 7. For the "if" direction, suppose that $H \vdash v \in c$ for some v and let $|v| = h + 1$. If $h \leqslant n$, then $|v| \leqslant n + 1$ and therefore $\mathbf{nemp}(c)\phi_n = \mathbf{true}$ by Lemma 7. Otherwise, $\mathbf{nemp}(c)\phi_h = \mathbf{true}$ by Lemma 7. The result follows since $\phi_h = \phi_n$. □

References

[1] G. Berry, R. Sethi, From regular expressions to deterministic automata, Theoret. Comput. Sci. 48 (1) (1986) 117–126.

[2] T. Bray, D. Hollander, A. Layman, J. Clark, Namespaces in XML, 1999, ⟨http://www.w3.org/TR/REC-xml-names⟩.

[3] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, Extensible markup language (XML™), 2000, ⟨http://www.w3.org/XML/⟩.

[4] A. Brüggemann-Klein, M. Murata, D. Wood, Regular tree and regular hedge languages over unranked alphabets, Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[5] J.A. Brzozowski, Derivatives of regular expressions, J. ACM 11 (4) (1964) 481–494.

[6] P. Buneman, B. Pierce, Union types for semistructured data, in: Internet Programming Languages, Lecture Notes in Computer Science, Vol. 1686, Springer, Berlin, 1998, Proc. Internat. Database Programming Languages Workshop.

[7] J. Clark, TREX: tree regular expressions for XML, 2001, ⟨http://www.thaiopensource.com/trex/⟩.

[8] J. Clark, 2002, ⟨http://www.thaiopensource.com/relaxng/implement.html⟩.

[9] J. Clark, M. Murata, RELAX NG, 2001, ⟨http://www.relaxng.org⟩.

[10] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree automata techniques and applications, draft book, 1999, Available electronically on: ⟨http://www.grappa.univ-lille3.fr/tata⟩.

[11] D.C. Fallside, XML schema part 0: primer, W3C recommendation, 2001, ⟨http://www.w3.org/TR/xmlschema-0/⟩.

[12] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, P. Wadler, XQuery 1.0 formal semantics, 2001, ⟨http://www.w3.org/TR/query-semantics/⟩.

[13] A. Frisch, G. Castagna, V. Benzaken, Semantic subtyping, in: 17th Annu. IEEE Symp. on Logic in Computer Science, 2002, pp. 137–146.

[14] R. Harper, B. Pierce, A record calculus based on symmetric concatenation, in: Proc. 18th Annu. ACM Symp. on Principles of Programming Languages, Orlando, FL, ACM, New York, 1991, pp. 131–142.

[15] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.

[16] H. Hosoya, M. Murata, Validation and boolean operations for attribute–element constraints, in: Programming Languages Technologies for XML (PLAN-X), 2002, pp. 1–10.

[17] H. Hosoya, B.C. Pierce, XDuce: a typed XML processing language (preliminary report), in: Proc. Third Internat. Workshop on the Web and Databases (WebDB2000), Lecture Notes in Computer Science, Vol. 1997, Springer, Berlin, 2000, pp. 226–244.

[18] H. Hosoya, B.C. Pierce, Regular expression pattern matching for XML, in: 25th Annu. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, 2001, pp. 67–80.

[19] H. Hosoya, J. Vouillon, B.C. Pierce, Regular expression types for XML, in: Proc. Internat. Conf. on Functional Programming (ICFP), 2000, pp. 11–22.

[20] H. Hosoya, J. Vouillon, B.C. Pierce, Regular expression types for XML, ACM Transactions on Programming Languages and Systems 27 (1) (2004) 46–90.
[21] N. Klarlund, A. Møller, M.I. Schwartzbach, DSD: a schema language for XML, 2000, ⟨http://www.brics.dk/DSD/⟩.
[22] T. Milo, D. Suciu, V. Vianu, Typechecking for XML transformers, in: Proc. 19th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems, ACM, 2000, pp. 11–22.
[23] A. Møller, Document structure description 2.0, BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7, December 2002, Available from: ⟨http://www.brics.dk/DSD/⟩.
[24] M. Murata, Transformation of documents and schemas by patterns and contextual conditions, in: Principles of Document Processing '96, Lecture Notes in Computer Science, Vol. 1293, Springer, Berlin, 1997, pp. 153–169.
[25] M. Murata, RELAX (REgular LAnguage description for XML), 2001, ⟨http://www.xml.gr.jp/relax/⟩.
[26] M. Nottingham, R. Sayre, The atom syndication format, December 2005, ⟨ftp://ftp.rfc-editor.org/in-notes/rfc4287.txt⟩, RFC 4287.
[27] OASIS, SGML/XML elements versus attributes, 2002, ⟨http://xml.coverpages.org/elementsAndAttrs.html⟩.
[28] A. Tozawa, Towards static type checking for XSLT, in: Proc. ACM Symp. on Document Engineering, 2001.
[29] J. Vouillon, Interleaving types for XML, Personal communication, 2001.
[30] S.D. Zilio, D. Lugiez, XML schema, tree logic and sheaves automata, in: Internat. Conf. on Rewriting Techniques and Applications, Vol. 2706, Springer, Berlin, 2003, pp. 246–263.