# The Essence of XML

Jérôme Siméon
Bell Laboratories
simeon@research.bell-labs.com

Philip Wadler
Avaya Labs
wadler@avaya.com

**Abstract**

The World-Wide Web Consortium (W3C) promotes XML and related standards, including XML Schema, XQuery, and XPath. This paper describes a formalization XML Schema. A formal semantics based on these ideas is part of the official XQuery and XPath specification, one of the first uses of formal methods by a standards body. XML Schema features both named and structural types, with structure based on tree grammars. While structural types and matching have been studied in other work (notably XDuce, Relax NG, and a previous formalization of XML Schema), this is the first work to study the relation between named types and structural types, and the relation between matching and validation.

## 1 Introduction

What's in a name? A Montague and a Capulet possess the same structure, and some would argue that is all that matters.

Traditionally, there are two approaches to type systems, named and structural. The named approach is prevalent in most widely-used programming languages, including Fortran, Cobol, Algol, Pascal, C, Modula, Java, and others. The structural approach is prevalent in most theories of types, including theories of record and object types devised by Reynolds, Wand, Abadi and Cardelli, Kim, and others [9, 1, 15].

As a simple example of the distinction between named and structural typing, consider the following type declarations.

type Feet = Integer
type Miles = Integer

In a language with named typing, this creates two new types, and one cannot pass a parameter of type Feet where a parameter of type Miles is expected. They are different types because they have different names. In a language with structural typing, both Feet and Miles are synonyms for the type Integer. They are the same type because they have the same structure.

(Astronauts may prefer named typing. In a 1985 test for the Strategic Defense Initiative, a laser aimed from an observatory in Hawaii was to be bounced off a mirror on the space shuttle. An astronaut entered the height of the laser and the shuttle rolled over. The problem is that the astronaut entered a height of 10,023 feet (below the shuttle), which the software interpreted as 10,023 miles (above the shuttle), hence the roll. [13].)

The dichotomy between names and structures is not quite so stark as at first it might appear. Many languages use combinations of named and structural typing. For instance, in ML record types are purely structural, but two types declared with "datatype" are distinct, even if they have the same structure. Further, relations between names always imply corresponding relations between structures. For instance, in Java if one class is declared to extend another then the first class always has a structure that extends the second. Conversely, structural relations depend upon names. For instance, names are used to identify the fields of a record.

**Types and languages for XML.** As types spread to new areas of computing, so too does the feud between names and structures. A case in point is XML, a standard for describing documents promulgated by the World-Wide Web Consortium (W3C) [2].

There are a number of type systems for XML, including: DTDs, part of the original W3C recommendation defining XML [2]; XML Schema, a W3C recommendation which supersedes DTDs [18]; Relax NG, an Oasis standard [6]; Relax [12] and TREX [5], two ancestors of Relax NG; and the type systems of XDuce [10] and YATL [7]. All of these take a structural approach to typing, with the exception of XML Schema, which takes a named approach, and the possible exception of DTDs, which are so restricted that the named and structural approaches might be considered to coincide.

A type system without a programming language is like Juliet without Romeo — unable to survive alone. The W3C is responsible for three programming languages connected with XML: XSLT, a language for stylesheets [4, 11]; XQuery, an analogue of SQL for XML data [20]; and XPath, the common core of XSLT and XQuery, which is jointly managed by the working groups responsible for the other two languages [19]. All three of these are functional languages. XSLT 1.0 and XPath 1.0 became recommendations in November 1999 — they are untyped. XML Schema 1.0 became a recommendation in May 2001. XSLT 2.0, XQuery 1.0, and XPath 2.0 are currently being designed — they have type systems based on XML Schema.

This paper presents a formalization of XML Schema, developed in conjunction with the XQuery and XPath working groups. The paper presents a simplified version, treating the essential constructs. The full version is being developed as part of the XQuery and XPath Formal Semantics [21], one of the first industrial specifications to exploit formal methods. The full version treats not just XML Schema, but also the dynamic and static semantics of the XQuery and XPath.

(As of this writing, the draft of the XQuery Formal Semantics that incorporates named typing has not yet been approved for publication by the XQuery working group. Approval should be granted within the next month or two.)

XQuery has both a specification in prose [20] and a formal semantics [21], each with parallel structure. Formal methods are particularly helpful for typing — the only complete description of the static type system of XQuery and XPath is in the formal specification. However, keeping two specifications in sync has not always been easy.

The act of preparing the formal semantics has uncovered a number of errors or omissions in the prose specification. In particular, developing the material on the formal semantics of named typing led to the formulation of ten issues for consideration by the XQuery working group, each dealing with a point that was omitted in the prose specification of XQuery [22]

An earlier formal specification of XML Schema [3] was influenced by XDuce [10]; it ignored the named aspects of Schema and took a purely structural approach. The specification of Relax NG [6] also uses formal methods, also is purely structural, and was influenced by the earlier work on XML Schema [3].

**Matching and validation.** Types in XML differ in some ways from types as used elsewhere in computing. Traditionally, a value *matches* a type — given a value and a type, either the value belongs to the type or it does not. In XML, a value *validates* against a type — given an (external) value and a type, validation produces an (internal) value or it fails.

For instance, consider the following XML Schema.

```
<xs:simpleType name="feet">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>
<xs:element name="height" type="feet"/>
```

In our type system, this is written as follows.

```
define type feet restricts xs:integer
define element height of type feet
```

Now consider the following XML document.

```
<height>10023</height>
```

In our model, before validation this is represented as follows.

```
  <height>10023</height>
⇒
  element height { "10023" }
```

And after validation it is represent as follows.

```
  validate as element height { <height>10023</height> }
⇒
  element height of type feet { 10023 }
```

Validation has annotated the element with its type, and converted the text "10023" into the corresponding integer number 10023.

Our model provides both validation and matching. Validation attaches types to XML data. Unvalidated data may not match against a type. The following *does not* hold.

```
  element height { "10023" }
matches
  element height
```

After validation, matching succeeds. The following *does* hold.

```
  element height of type feet { 10023 }
matches
  element height
```

The inverse of validation is type erasure.

```
  element height of type feet { 10023 }
erases to
  element height { "10023" }
```

The following theorem characterizes validation in terms of matching and erasure.

**Theorem 1** *We have that*

$$\textsf{\textbf{validate as}}\ \mathit{Type}\ \{\ \mathit{UntypedValue}\ \}\ \Rightarrow\ \mathit{Value}$$

*if and only if*

$$\mathit{Value}\ \textsf{\textbf{matches}}\ \mathit{Type}$$

*and*

$$\mathit{Value}\ \textsf{\textbf{erases to}}\ \mathit{UntypedValue}.$$

Perhaps this theorem looks obvious, but if so let us assure you that it was not obvious to us when we began. It took some time to come to this formulation, and some tricky adjustments were required to ensure that it holds.

One trick is that we model validation and erasure by relations, not functions. Naively, one might expect validation to be a partial function and erasure to be a function. That is, for a given type each untyped value validates to yield at most one typed value, and each typed values erases to one untyped value. One subtlety of the system presented here is that validation and erasure are modeled by relations. For example, the strings "7" and "007" both validate to yield the integer 7, and hence we also have that the integer erases to yield either string.

**Relation of our model to Schema.** Schema is a large and complex standard. In this paper, we attempt to model only the most essential features. These include: simple types and complex types; named and anonymous types; global and local elements; atomic, list, and union simple types; and derivation by restriction and by extension. We model only two primitive datatypes, xs:integer and xs:string, while Schema has nineteen primitive datatypes.

Many features of Schema that are omitted here are dealt with in the formal semantics for XQuery [21]. These include: namespaces; attributes; all groups (interleaving); text nodes; mixed content; substitution groups; xsi:nil attributes; and xsi:type attributes. There are other features of Schema that are not yet dealt with in the full formal semantics, but which we hope to model in future. These include: abstract types; default and fixed values; skip, lax, and strict wildcards; and facets of simple types.

Schema is normally written in an XML notation, but here we use a notation that is more readable and compact. The mapping of XML notation into our notation is described in the XQuery formal semantics.

There are a few aspects in which our treatment diverges from Schema. First, we permit ambiguous content models, while Schema does not. We do this because it makes our model simpler, and because ambiguity is important to support type checking, as discussed in Section 8. Second, we permit one type to be a restriction of another whenever any value that matches against the first type also matches against the second, while Schema imposes ad hoc syntactic constraints. Again, we do this because it makes our model simpler, and because our more general model better supports type checking. Third, we only support the occurrence operators ?, +, and *, while Schema supports arbitrary counts for minimum and maximum occurrences. This is because arbitrary counts may lead to a blow-up in the size of the finite-state automata we use to check when one type is included in another.

**Shortcomings of XML and Schema.** Our aim is to model XML and Schema as they exist — we do not claim that these are the best possible designs. Indeed, we would argue that XML and Schema have several shortcomings.

First, we would argue that a data representation should explicitly distinguish, say, integers from strings, rather than to infer which is which by validation against a Schema. (This is one of the many ways in which Lisp S-expressions are superior to XML.)

Second, while derivation by extension in Schema superficially resembles subclassing in object-oriented programming, in fact there are profound differences. In languages such as Java, one can typecheck code for a class without knowing all subclasses of that class (this supports separate compilation). But in XML Schema, one cannot validate against a type without knowing all types that derive by extension from that type (and hence separate compilation is problematic).

Nonetheless, XML and Schema are widely used standards, and there is value in modeling these standards. In particular, such models may: (i) improve our understanding of exactly what is mandated by the standard, (ii) help implementors create conforming implementations, and (iii) suggest how to improve the standards.

**Related publications.** A preliminary version of this paper was delivered as an invited talk at FLOPS 2002 [17]. This version has revised material in Sections 1, 4, 6, and 7, entirely new material in Sections 9 and 10, and numerous improvements throughout.

**Organization.** The remainder of this paper is organized as follows. Section 2 introduces XML Schema by example. Section 3 describes values and types. Section 4 describes four ancillary judgements. Section 5 describes matching. Section 6 describes erasure. Section 7 describes validation. Section 8 discusses ambiguity and the validation theorem. Section 9 describes subtyping and constraints on sensible types. Section 10 returns to the relation between names and structures, and describes an optimization theorem for matching.

## 2 XML Schema by example

XML Schema supports a wide range of features. These include simple types and complex types, anonymous types, global and local declarations, derivation by restriction, and derivation by extension.

**Simple and complex types.** Here are declarations for two elements of simple type, one element with a complex type, and one complex type.

```
define element title of type xs:string
define element author of type xs:string
define element paper of type paperType
define type paperType {
  element title ,
  element author +
}
```

Schema specifies nineteen primitive simple type types, including xs:string and xs:integer.

A type declaration associates a name and a structure. The structure of a complex type is a regular expression over elements. As usual, , denotes sequence, | denotes alternation, ? denotes an optional occurrence, + denotes one or more occurrences, and * denotes zero or more occurrences.

Validating annotates each element with its type.

```
validate as paper {
  <paper>
    <title>The Essence of Algol</title>
    <author>John Reynolds</author>
  </paper>
}
⇒
element paper of type paperType {
  element title of type string { "The Essence of Algol" } ,
  element author of type string { "John Reynolds" }
}
```

**Anonymous types.** Instead of naming a type, it can defined in place without a name. Here is the paper element with its type expanded in place.

```
define element paper {
  element title , element author +
}
```

Validating now yields the following result.

```
validate as paper {
  <paper>
    <title>The Essence of ML</title>
    <author>Robert Harper</author>
    <author>John Mitchell</author>
  </paper>
}
⇒
element paper {
  element title of type xs:string { "The Essence of ML" } ,
  element author of type xs:string { "Robert Harper" } ,
  element author of type xs:string { "John Mitchell" }
}
```

Now the paper element has no type annotation, because there is no type name to annotate it with. The other elements still have type annotations.

**Global and local declarations.** Similarly, one may include an element declaration in place. Here is the paper element with the nested elements expanded in place.

```
define element paper {
  element title of type xs:string ,
  element author of type xs:string +
}
```

Here the paper is declared globally, while title and author are declared locally. In this case, validation proceeds exactly as before.

Allowing local declarations increases expressiveness, because now it is possible for elements with the same name to be assigned different types in different places; see [14, 8]. An example of such a definition appears later.

**Atomic, list, and union types.** Every simple type is an atomic type, a list type, or a union type. The atomic types are the nineteen primitive types of Schema, such as xs:string and xs:integer, and the types derived from them. List types are formed using the occurrence operators ?, +, and *, taken from regular expressions. Union types are formed using the alternation operator |, also taken from regular expressions.

Here is an example of a list type.

```
element ints { xs:integer + }
```

In XML notation, lists are written space-separated.

```
 validate as ints { <ints>1 2 3</ints> }
⇒
 element ints { 1, 2, 3 }
```

Some types may be ambiguous. XML Schema specifies how to resolve this ambiguity: every space is taken as a list separator, and in case of a union the first alternative that works is chosen.

```
define element trouble { ( xs:integer | xs:string ) * }
```

```
 validate as trouble { <trouble>this is not 1 string</trouble> }
⇒
 element trouble "this", "is", "not", 1, "string"
```

Ambiguous types can be problematic; this will be further discussed in Section 8.

**Derivation by restriction on simple types.** New simple types may be derived by restriction.

```
define type miles restricts xs:integer
define type feet restricts xs:integer
```

Here is an example where two height elements have different types.

```
define element configuration {
  element shuttle { element height of type miles } ,
  element laser { element height of type feet }
}
```

Validation annotates the different height elements with different types.

```
 validate as element configuration {
   <configuration>
     <shuttle><height>120</height></shuttle>
     <laser><height>10023</height></laser>
   </configuration>
 }
⇒
 element configuration {
   element shuttle { element height of type miles { 120 } } ,
   element laser { element height of type feet { 10023 } }
 }
```

Both miles and feet are subtypes of xs:integer, but neither is a subtype of the other. The following function definition is legal.

```
define function laser˙height (
  element configuration $c
) returns element height of type feet {
  $c/laser/height
}
```

It would still be legal if feet was replaced by xs:integer, but not if feet was replaced by miles. In this example, element configuration is the type of the formal parameter $c, and the XPath expression $c/laser/height extracts the height child of the laser child of the configuration element.

**Derivation by restriction on complex types.** New complex types may also be derived by restriction. The following example is a simplified form of the information that may occur in a bibliographic database, such as that used by BibTeX.

```
define element bibliography {
  element of type publicationType *
}
define type publicationType {
  element author * ,
  element title ? ,
  element journal ? ,
  element year ?
}
define type articleType restricts publicationType {
  element author + ,
  element title ,
  element journal ,
  element year
}
define type bookType restricts publicationType {
  element author + ,
  element title ,
  element year
}
define element book of type bookType
define element article of type articleType
```

Here a publication may have any number of authors, a mandatory title, and a optional journal and year. An article must have at least one author, and a mandatory title, journal, and year. A book must have at least one author, a mandatory title and year, and no journal.

Derivation by restriction declares a relationship between two types. This relation depends on both names and structures, in the sense that one name may be derived by restriction from another name only if every value that matches the structure of the first also matches the structure of the second.

When one type is derived from another by restriction, it is fine to pass the restricted type where the base type is expected. For example, consider the following function.

```
define function getTitle (
  element of type publicationType $p
) returns element title {
  $p/title
}
```

Here it is acceptable to pass either an article or book element to the function getTitle().

There is a type xs:anyType from which all other types are derived. If a type definition does not specify otherwise, it is considered a restriction of xs:anyType. There is also a type xs:anySimpleType that is derived from xs:anyType and from which all other simple types are derived.

**Derivation by extension.** New complex types may also be derived by extension.

```
define type color restricts xs:string
define type pointType {
  element x of type xs:integer ,
  element y of type xs:integer
}
define type colorPointType extends pointType {
  element c of type color
}
define element point of type pointType
define element colorPoint of type colorPointType
```

When one type restricts another, one must check that the proper relation holds between the types. When one type extends another, the relation holds automatically, since values of the new type are defined to consist of the concatenation of values of the base type with values of the extension.

Again, when one type is derived from another by extension, it is fine to pass the extended type where the base type is expected. Unlike with restriction, this can lead to surprising consequences. Consider the following.

```
define function countChildren (
  element of type pointType $p
) returns xs:integer {
  count($p/*)
}
```

This function counts the number of children of the element $p, which will be 2 or 3, depending on whether $p is an element of type point or colorPoint.

In XQuery, type checking requires that one knows all the types that can be derived from a given type — the type is then treated as the union of all types that can be derived from it. Types derived by restriction add nothing new to this union, but types derived by extension do. This "closed world" approach — that type checking requires knowing all the types derived from a type — is quite different from the "open world" approach used in many object-oriented languages — where one can type-check a class without knowing all its subclasses.

In an object-oriented language, one might expect that if an element of type colorPoint is passed to this function, then the x and y elements would be visible but the c element would not be visible. Could the XQuery design adhere better to the object-oriented expectation? It is not obvious how to do so. For instance, consider the above function when pointType is replaced by xs:anyType.

```
define function countChildren (
  element of type xs:anyType $x
) returns xs:integer {
  count($x/*)
}
```

Here it seems natural to count all the children, while an object-oriented interpretation might suggest counting none of the children, since xs:anyType is the root of the type hierarchy.

## 3 Values and types

### 3.1 Values

We now give a formal defintion of values. We take names, string, and integers as primitive. We do not formalize the mapping between XML notation and our notation for values, as it is straightforward.

A value is a sequence of zero or more items. An item is either an element or an atomic value. An element has an element name, an optional type annotation, and a value. An element with no type annotation is the same as an element with the type annotation xs:anyType. An atomic value is a string or an integer. We write $Value_1$ , $Value_2$ for the concatenation of two values.

$$\begin{array}{lcl} \mathit{Value} & ::= & () \\ & \mid & \mathit{Item}(,\mathit{Item})* \\ \mathit{Item} & ::= & \mathit{Element} \\ & \mid & \mathit{Atom} \\ \mathit{Element} & ::= & \textsf{element}\ \mathit{ElementName}\ \mathit{OfType}?\ \{\ \mathit{Value}\ \} \\ \mathit{OfType} & ::= & \textsf{of type}\ \mathit{TypeName} \\ \mathit{Atom} & ::= & \mathit{String} \mid \mathit{Integer} \end{array}$$

An untyped value is a sequence of zero or more untyped items. An untyped item is either an element without type annotation or a string. Untyped values are used to described XML documents before validation. Every untyped value is a value.

$$\begin{array}{lcl} \mathit{UntypedValue} & ::= & () \\ & \mid & \mathit{UntypedItem}(,\mathit{UntypedItem})* \\ \mathit{UntypedItem} & ::= & \textsf{element}\ \mathit{ElementName}\ \{\ \mathit{UntypedValue}\ \} \\ & \mid & \mathit{String} \end{array}$$

A simple value consists of a sequence of zero or more atomic values. Every simple value is a value.

$$\begin{array}{lcl} \mathit{SimpleValue} & ::= & () \\ & \mid & \mathit{Atom}(,\mathit{Atom})* \end{array}$$

Here is an example of a value.

```
element paper of type paperType {
  element title of type xs:string { "The Essence of Algol" } ,
  element author of type xs:string { "John Reynolds" }
}
```

Here is an example of an untyped value.

```
element paper {
  element title { "The Essence of Algol" },
  element author { "John Reynolds" }
}
```

Here are examples of simple values.

```
"John Reynolds"
10023
1, 2, 3
```

## 3.2 Types

Types are modeled on regular tree grammars [16, 8]. A type is either the empty sequence (()), an item type, or composed by sequence (,), choice (|), or multiple occurrence — either optional (?), one or more (+), or zero or more (*).

$$
\begin{array}{lcl}
\mathit{Type} & ::= & () \\
 & | & \mathit{ItemType} \\
 & | & \mathit{Type}\ ,\ \mathit{Type} \\
 & | & \mathit{Type}\ \mathsf{|}\ \mathit{Type} \\
 & | & \mathit{Type}\ \mathit{Occurrence} \\
\mathit{Occurrence} & ::= & \mathsf{?} \mid \mathsf{+} \mid \mathsf{*}
\end{array}
$$

An item type is an element type or an atomic type. Atomic types are specified by name; these names include xs:string and xs:integer. Every *AtomicTypeName* is also a *TypeName*.

$$
\begin{array}{lcl}
\mathit{ItemType} & ::= & \mathit{ElementType} \\
 & | & \mathit{AtomicTypeName}
\end{array}
$$

An element type gives an optional name and an optional type specifier. A name alone refers to a global declaration (element author). A name with a type specifier is a local declaration (element author of type xs:string). A type specifier alone is a local declaration that matches any name (element of type xs:string). The word element alone refers to any element.

$$
\mathit{ElementType} \ ::= \ \mathsf{element}\ \mathit{ElementName}?\ \mathit{TypeSpecifier}?
$$

A type specifier either references a global type, or defines a type by derivation. A type derivation either restricts an atomic type, or restricts a named type to a given type, or extends a named type by a given type.

$$
\begin{array}{lcl}
\mathit{TypeSpecifier} & ::= & \mathit{TypeReference} \\
 & | & \mathit{TypeDerivation} \\
\mathit{TypeReference} & ::= & \mathsf{of\ type}\ \mathit{TypeName} \\
\mathit{TypeDerivation} & ::= & \mathsf{restricts}\ \mathit{AtomicTypeName} \\
 & | & \mathsf{restricts}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \} \\
 & | & \mathsf{extends}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}
\end{array}
$$

A simple type is composed from atomic types by choice or occurrence. Every simple type is a type.

$$
\begin{array}{lcl}
\mathit{SimpleType} & ::= & \mathit{AtomicTypeName} \\
 & | & \mathit{SimpleType}\ \mathsf{|}\ \mathit{SimpleType} \\
 & | & \mathit{SimpleType}\ \mathit{Occurrence}
\end{array}
$$

We saw many examples of types and simple types in Section 2.

## 3.3 Top level definitions

At the top level one can define elements and types. A global element declaration, like a local element declaration, consists of an element name and a type specifier. A global type declaration consists of a type name and a type derivation.

$$
\begin{array}{lcl}
\mathit{Definition} & ::= & \mathsf{define\ element}\ \mathit{ElementName}\ \mathit{TypeSpecifier} \\
 & | & \mathsf{define\ type}\ \mathit{TypeName}\ \mathit{TypeDerivation}
\end{array}
$$

We saw many examples of element and type declarations in Section 2.

## 3.4 Built-in type declarations

The two XML Schema built-in types xs:anyType and xs:anySimpleType are defined as follows.

```
define type xs:anyType restricts xs:anyType {
  xs:anySimpleType | element *
}
define type xs:anySimpleType restricts xs:anyType {
  ( xs:integer | xs:string ) *
}
```

## 4 Ancillary judgments

We now define four ancillary judgments that are used in matching and validation. Here is the rule from matching that uses these judgments.

$$
\frac{
\begin{array}{c}
\mathit{ElementType}\ \textbf{yields}\ \mathit{BaseElementName}\ \mathit{TypeSpecifier} \\
\mathit{TypeSpecifier}\ \textbf{resolves to}\ \mathit{BaseTypeName}\ \{\ \mathit{Type}\ \} \\
\mathit{ElementName}\ \textbf{substitutes for}\ \mathit{BaseElementName} \\
\mathit{TypeName}\ \textbf{derives from}\ \mathit{BaseTypeName} \\
\mathit{Value}\ \textbf{matches}\ \mathit{Type}
\end{array}
}{
\begin{array}{c}
\mathsf{element}\ \mathit{ElementName}\ \mathsf{of\ type}\ \mathit{TypeName}\ \{\ \mathit{Value}\ \} \\
\textbf{matches}\ \mathit{ElementType}
\end{array}
}
$$

The element type yields an element name set and a type specifier, and the type specifier resolves to a base type name and a type. Then the given element matches the element type if three things hold: the element name must be within the element name set, the type name must derive from the base type name, and the value must match the type.

The next four sections define the first four judgments in the hypothesis of the above rule.

### 4.1 Yields

The judgment

$$
\mathit{ElementType}\ \textbf{yields}\ \mathit{ElementName}\ \mathit{TypeSpecifier}
$$

takes an element type and yields an element name and a type specifier. If the element type does not specify an element name, then the distinguished element name * is returned. For example,

```
element author yields author of type xs:string
element height of type feet yields height of type feet
element of type feet yields * of type feet
element yields * of type xs:anyType
```

If the element type is a reference to a global element, then it yields the the name of the element and the type specifier from the element declaration. (Note the use of a top-level definition as a hypothesis.)

$$\frac{\textsf{define element}\ \mathit{ElementName}\ \mathit{TypeSpecifier}}{\textsf{element}\ \mathit{ElementName}\ \textbf{yields}\ \mathit{ElementName}\ \mathit{TypeSpecifier}}$$

If the element type contains an element name and a type specifier, then it yields the given element name and type specifier.

$$\frac{}{\begin{array}{c}\textsf{element}\ \mathit{ElementName}\ \{\ \mathit{TypeSpecifier}\ \}\\ \textbf{yields}\ \mathit{ElementName}\ \mathit{TypeSpecifier}\end{array}}$$

If the element type contains only a type specifier, then it yields the wildcard name and the type specifier.

$$\frac{}{\textsf{element}\ \{\ \mathit{TypeSpecifier}\ \}\ \textbf{yields}\ *\ \mathit{TypeSpecifier}}$$

If the element type has no element name and no type specifier, then it yields the wildcard name and the type xs:anyType.

$$\frac{}{\textsf{element}\ \textbf{yields}\ *\ \textsf{of type xs:anyType}}$$

## 4.2 Resolution

The judgment

$$\mathit{TypeSpecifier}\ \textbf{resolves to}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}$$

resolves a type specifier to a type name and a type. For example,

```
  of type colorPoint
resolves to
  colorPoint {
    element x of type xs:integer ,
    element y of type xs:integer ,
    element c of type color
  }
```

and

```
  restricts xs:integer
resolves to
  xs:integer { xs:integer }
```

and

```
  restricts publicationType {
      element author + ,
      element title ,
      element year
  }
resolves to
  publicationType {
      element author + ,
      element title ,
      element year
  }
```

and

```
  extends pointType {
    element c of type color
  }
resolves to
  colorPoint {
    element x of type xs:integer ,
    element y of type xs:integer ,
    element c of type color
  }
```

If the type specifier references a global type, then resolve the type derivation in its definition, yielding a base type name and a type. Resolution returns the type name and the type. (The base type name is discarded.)

$$\frac{\begin{array}{c}\textsf{define type}\ \mathit{TypeName}\ \mathit{TypeDerivation}\\ \mathit{TypeDerivation}\ \textbf{resolves to}\ \mathit{BaseTypeName}\ \{\ \mathit{Type}\ \}\end{array}}{\textsf{of type}\ \mathit{TypeName}\ \textbf{resolves to}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}}$$

If the type specifier restricts an atomic type, then return the atomic type as both the type name and the type.

$$\frac{}{\begin{array}{c}\textsf{restricts}\ \mathit{AtomicTypeName}\\ \textbf{resolves to}\ \mathit{AtomicTypeName}\ \{\ \mathit{AtomicTypeName}\ \}\end{array}}$$

If the type specifier is a restriction of a non-atomic type, then return the given type name and the given type.

$$\frac{}{\begin{array}{c}\textsf{restricts}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}\\ \textbf{resolves to}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}\end{array}}$$

If the type specifier is an extension, then resolve the name to get the base type, and return the given type name, and the result of concatenating the base type and the given type.

$$\frac{\textsf{of type}\ \mathit{TypeName}\ \textbf{resolves to}\ \mathit{TypeName}\ \{\ \mathit{BaseType}\ \}}{\begin{array}{c}\textsf{extends}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}\\ \textbf{resolves to}\ \mathit{TypeName}\ \{\ \mathit{BaseType}\ ,\ \mathit{Type}\ \}\end{array}}$$

## 4.3 Element name sets

The judgment

$$\mathit{ElementName}_1\ \textbf{substitutes for}\ \mathit{ElementName}_2$$

holds when the first element name may substitute for the second element name. This happens when the two names are equal, or when the second name is the distinguished element name *. For example:

```
paper substitutes for paper
paper substitutes for *
```

(We do not discuss element substitution groups here, but the judgment generalizes neatly to handle these.)

An element name may substitute for itself.

$$\frac{}{\mathit{ElementName}\ \textbf{substitutes for}\ \mathit{ElementName}}$$

An element name may substitute for the distinguished element name *.

$$\frac{}{\mathit{ElementName}\ \textbf{substitutes for}\ *}$$

### 4.4 Derives

The judgment

$$\mathit{TypeName}_1 \ \textbf{derives from} \ \mathit{TypeName}_2$$

holds when the first type name derives from the second type name. For example,

bookType **derives from** bookType
bookType **derives from** publicationType
bookType **derives from** xs:anyType

This relation is a partial order: it is reflexive and transitive by the rules below, and it is asymmetric because no cycles are allowed in derivation by restriction or extension.

Derivation is reflexive and transitive.

$$\frac{}{\mathit{TypeName} \ \textbf{derives from} \ \mathit{TypeName}}$$

$$\frac{\begin{array}{c}\mathit{TypeName}_1 \ \textbf{derives from} \ \mathit{TypeName}_2 \\ \mathit{TypeName}_2 \ \textbf{derives from} \ \mathit{TypeName}_3\end{array}}{\mathit{TypeName}_1 \ \textbf{derives from} \ \mathit{TypeName}_3}$$

Every type name derives from the type it is declared to derive from by restriction or extension.

$$\frac{\text{define type } \mathit{TypeName} \text{ restricts } \mathit{BaseTypeName}}{\mathit{TypeName} \ \textbf{derives from} \ \mathit{BaseTypeName}}$$

$$\frac{\text{define type } \mathit{TypeName} \text{ restricts } \mathit{BaseTypeName} \ \{ \ \mathit{Type} \ \}}{\mathit{TypeName} \ \textbf{derives from} \ \mathit{BaseTypeName}}$$

$$\frac{\text{define type } \mathit{TypeName} \text{ extends } \mathit{BaseTypeName} \ \{ \ \mathit{Type} \ \}}{\mathit{TypeName} \ \textbf{derives from} \ \mathit{BaseTypeName}}$$

## 5 Matches

The judgment

$$\mathit{Value} \ \textbf{matches} \ \mathit{Type}$$

holds when the given value matches the given type. For example,

```
  element author of type xs:string { "Robert Harper" } ,
  element author of type xs:string { "John Mitchell" }
matches
  element author of type xs:string +
```

and

```
10023 matches feet
```

and

```
  element colorPoint of type colorPointType {
    element x of type xs:integer { 1 }
    element y of type xs:integer { 2 }
    element c of type color { "blue" }
  }
matches
  element colorPoint
```

The empty sequence matches the empty sequence type.

$$\frac{}{() \ \textbf{matches} \ ()}$$

If two values match two types, then their sequence matches the corresponding sequence type.

$$\frac{\begin{array}{c}\mathit{Value}_1 \ \textbf{matches} \ \mathit{Type}_1 \\ \mathit{Value}_2 \ \textbf{matches} \ \mathit{Type}_2\end{array}}{\mathit{Value}_1 \ , \ \mathit{Value}_2 \ \textbf{matches} \ \mathit{Type}_1 \ , \ \mathit{Type}_2}$$

If a value matches a type, then it also matches a choice type where that type is one of the choices.

$$\frac{\mathit{Value} \ \textbf{matches} \ \mathit{Type}_1}{\mathit{Value} \ \textbf{matches} \ \mathit{Type}_1 \ | \ \mathit{Type}_2}$$

$$\frac{\mathit{Value} \ \textbf{matches} \ \mathit{Type}_2}{\mathit{Value} \ \textbf{matches} \ \mathit{Type}_1 \ | \ \mathit{Type}_2}$$

A value matches an optional occurrence of a type if it matches either the empty sequence or the type.

$$\frac{\mathit{Value} \ \textbf{matches} \ () \ | \ \mathit{Type}}{\mathit{Value} \ \textbf{matches} \ \mathit{Type}?}$$

A value matches one or more occurrences of a type if it matches a sequence of the type followed by zero or more occurrences of the type.

$$\frac{\mathit{Value} \ \textbf{matches} \ \mathit{Type} \ , \ \mathit{Type}*}{\mathit{Value} \ \textbf{matches} \ \mathit{Type}+}$$

A value matches zero or more occurrences of a type if it matches an optional one or more occurrences of the type.

$$\frac{\mathit{Value} \ \textbf{matches} \ \mathit{Type}+?}{\mathit{Value} \ \textbf{matches} \ \mathit{Type}*}$$

A string matches an atomic type name if the atomic type name derives from xs:string. Similarly for integers.

$$\frac{\mathit{AtomicTypeName} \ \textbf{derives from} \ \text{xs:string}}{\mathit{String} \ \textbf{matches} \ \mathit{AtomicTypeName}}$$

$$\frac{\mathit{AtomicTypeName} \ \textbf{derives from} \ \text{xs:integer}}{\mathit{Integer} \ \textbf{matches} \ \mathit{AtomicTypeName}}$$

The rule for matching elements was explained at the beginning of Section 4.

$$\frac{\begin{array}{c}\mathit{ElementType} \ \textbf{yields} \ \mathit{BaseElementName} \ \mathit{TypeSpecifier} \\ \mathit{TypeSpecifier} \ \textbf{resolves to} \ \mathit{BaseTypeName} \ \{ \ \mathit{Type} \ \} \\ \mathit{ElementName} \ \textbf{substitutes for} \ \mathit{BaseElementName} \\ \mathit{TypeName} \ \textbf{derives from} \ \mathit{BaseTypeName} \\ \mathit{Value} \ \textbf{matches} \ \mathit{Type}\end{array}}{\begin{array}{c}\text{element } \mathit{ElementName} \text{ of type } \mathit{TypeName} \ \{ \ \mathit{Value} \ \} \\ \textbf{matches} \ \mathit{ElementType}\end{array}}$$

## 6 Erasure

The judgment

$$\mathit{Value}\ \textbf{erases to}\ \mathit{UntypedValue}$$

holds when the given value erases to the untyped value. For example,

```
  element author of type xs:string { "John Reynolds" }
erases to
  element author { "John Reynolds" }
```

and

```
  element configuration {
   element shuttle { element height of type miles { 120 } } ,
    element laser { element height of type feet { 10023 } }
  }
erases to
  element configuration {
    element shuttle { element height { "120" } } ,
    element laser { element height { "10023" } }
  }
```

Erasure turns all atomic values into strings, and concatenates any adjacent strings in the result with a separating space. No space is added when an atomic value is adjacent to an element. For example,

```
  1, 2, 3 erases to "1 2 3"
```

and

```
  element fact { "I saw", 8, "cats" }
erases to
  element fact { "I saw 8 cats" }
```

and

```
  element fact { "I saw", 8, " ", element em { } "cats"  }
erases to
  element fact { "I saw 8 ", element em { } "cats"  }
```

Erasure is defined as a relation. Since an integer has more than one string representation, it may have more than one erasure. For example,

```
  7 erases to "7"
  7 erases to "007"
```

The empty sequence erases to itself.

$$\frac{}{()\ \textbf{erases to}\ ()}$$

The erasure of the concatenation of two values yields the concatenation of their erasures. If the first erasure ends in a string and the second erasure begins with a string, concatenate the strings with an intervening space.

$$\frac{\begin{array}{c}\mathit{Value}_1\ \textbf{erases to}\ \mathit{UntypedValue}_1\ ,\ \mathit{String}_1\\ \mathit{Value}_2\ \textbf{erases to}\ \mathit{String}_2\ ,\ \mathit{UntypedValue}_2\end{array}}{\begin{array}{c}\mathit{Value}_1\ ,\ \mathit{Value}_2\\ \textbf{erases to}\\ \mathit{UntypedValue}_1\ ,\ \textbf{concat}(\mathit{String}_1, \text{" "}, \mathit{String}_2)\ ,\ \mathit{UntypedValue}_2\end{array}}$$

$$\frac{\begin{array}{c}\mathit{Value}_1\ \textbf{erases to}\ \mathit{UntypedValue}_1\\ \mathit{Value}_2\ \textbf{erases to}\ \mathit{UntypedValue}_2\\ \mathit{UntypedValue}_1 \text{ does not end in a string or}\\ \mathit{UntypedValue}_2 \text{ does not begin with a string}\end{array}}{\mathit{Value}_1\ ,\ \mathit{Value}_2\ \textbf{erases to}\ \mathit{UntypedValue}_1\ ,\ \mathit{UntypedValue}_2}$$

The erasure of an element is an element that has the same name and the erasure of the given content.

$$\frac{\mathit{Value}\ \textbf{erases to}\ \mathit{UntypedValue}}{\begin{array}{c}\text{element } \mathit{ElementName} \text{ of type } \mathit{TypeName}\ \{\ \mathit{Value}\ \}\\ \textbf{erases to}\ \text{element } \mathit{ElementName}\ \{\ \mathit{UntypedValue}\ \}\end{array}}$$

A string erases to itself.

$$\frac{}{\mathit{String}\ \textbf{erases to}\ \mathit{String}}$$

An integer erases to any string that represents it.

$$\frac{}{\textbf{integer-of-string}(\mathit{String})\ \textbf{erases to}\ \mathit{String}}$$

## 7 Validation

The judgment

$$\textbf{validate as}\ \mathit{Type}\ \{\ \mathit{UntypedValue}\ \}\ \Rightarrow\ \mathit{Value}$$

holds if validating the string against the simple type succeeds and returns the validated value. For example,

```
validate as element of type xs:string {
  element author { "John Reynolds" }
}
⇒
  element author of type xs:string { "John Reynolds" }
```

and

```
validate as element configuration {
  element configuration {
    element shuttle { element height { "120" } } ,
    element laser { element height { "10023" } }
  }
} ⇒
  element configuration {
   element shuttle { element height of type miles { 120 } } ,
    element laser { element height of type feet { 10023 } }
  }
```

and

```
validate as xs:integer * { "1 2 3" } ⇒ 1, 2, 3
```

Validating the empty sequence as the empty type yields the empty sequence.

$$\frac{}{\textbf{validate as}\ ()\ \{\ ()\ \}\ \Rightarrow\ ()}$$

Validating a concatenation of untyped values against a concatenation of types yields the concatenation of the validated values. Decomposing a concatenation of untyped values removes a space between adjacent strings, inverting the corresponding rule for erasure.

$$\frac{\begin{array}{c}\textsf{validate as } \mathit{Type}_1 \texttt{ \{ } \mathit{UntypedValue}_1 \texttt{ , } \mathit{String}_1 \texttt{ \} } \Rightarrow \mathit{Value}_1 \\ \textsf{validate as } \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue}_2 \texttt{ , } \mathit{String}_2 \texttt{ \} } \Rightarrow \mathit{Value}_2\end{array}}{\begin{array}{c}\textsf{validate as } \mathit{Type}_1 \texttt{ , } \mathit{Type}_2 \texttt{ \{ } \\ \mathit{UntypedValue}_1 \texttt{ , } \textsf{concat}(\mathit{String}_1, \text{" "}, \mathit{String}_2) \texttt{ , } \mathit{UntypedValue}_2 \\ \texttt{\} } \Rightarrow \mathit{Value}_1 \texttt{ , } \mathit{Value}_2\end{array}}$$

$$\frac{\begin{array}{c}\textsf{validate as } \mathit{Type}_1 \texttt{ \{ } \mathit{UntypedValue}_1 \texttt{ \} } \Rightarrow \mathit{Value}_1 \\ \textsf{validate as } \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue}_2 \texttt{ \} } \Rightarrow \mathit{Value}_2 \\ \mathit{UntypedValue}_1 \text{ does not end in a string or} \\ \mathit{UntypedValue}_2 \text{ does not begin with a string}\end{array}}{\begin{array}{c}\textsf{validate as } \mathit{Type}_1 \texttt{ , } \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue}_1 \texttt{ , } \mathit{UntypedValue}_2 \texttt{ \} } \\ \Rightarrow \mathit{Value}_1 \texttt{ , } \mathit{Value}_2\end{array}}$$

Validating a value against a choice type yields the result of validating the value as either the first or second type in the choice.

$$\frac{\textsf{validate as } \mathit{Type}_1 \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}{\textsf{validate as } \mathit{Type}_1 \mid \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}$$

$$\frac{\textsf{validate as } \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}{\textsf{validate as } \mathit{Type}_1 \mid \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}$$

The validation rules for occurrences are similar to the rules for occurrences in matching.

$$\frac{\textsf{validate as } (() \mid \mathit{Type}) \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}{\textsf{validate as } \mathit{Type}\texttt{?} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}$$

$$\frac{\textsf{validate as } (\mathit{Type} \texttt{ , } \mathit{Type}\texttt{*}) \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}{\textsf{validate as } \mathit{Type}\texttt{+} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}$$

$$\frac{\textsf{validate as } \mathit{Type}\texttt{+?} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}{\textsf{validate as } \mathit{Type}\texttt{*} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}}$$

Validating a string against an atomic type derived from xs:string yields the string itself.

$$\frac{\mathit{AtomicTypeName} \textsf{ derives from } \textsf{xs:string}}{\textsf{validate as } \mathit{AtomicTypeName} \texttt{ \{ } \mathit{String} \texttt{ \} } \Rightarrow \mathit{String}}$$

Validating a string against an atomic type derived from xs:integer yields the result of converting the string to an integer.

$$\frac{\mathit{AtomicTypeName} \textsf{ derives from } \textsf{xs:integer}}{\begin{array}{c}\textsf{validate as } \mathit{AtomicTypeName} \texttt{ \{ } \mathit{String} \texttt{ \} } \\ \Rightarrow \textsf{integer-of-string}(\mathit{String})\end{array}}$$

Validating an element against an element type is described by the following rule.

$$\frac{\begin{array}{c}\mathit{ElementType} \textsf{ yields } \mathit{BaseElementName}\ \mathit{TypeSpecifier} \\ \mathit{TypeSpecifier} \textsf{ resolves to } \mathit{TypeName} \texttt{ \{ } \mathit{Type} \texttt{ \} } \\ \mathit{ElementName} \textsf{ substitutes for } \mathit{BaseElementName} \\ \textsf{validate as } \mathit{Type} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}\end{array}}{\begin{array}{c}\textsf{validate as } \mathit{ElementType} \texttt{ \{ } \\ \textsf{element } \mathit{ElementName} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \\ \texttt{\} } \Rightarrow \textsf{element } \mathit{ElementName} \textsf{ of type } \mathit{TypeName} \texttt{ \{ } \mathit{Value} \texttt{ \} }\end{array}}$$

The element type yields an element name set and a type specifier, and the type specifier resolves to a type name and a type. Then the given element matches the element type if two things hold: the element name must be within the element name set, and validating the untyped value against the type must yield a value. The resulting element has the element name, the type name, and the validated value.

## 8 Ambiguity and the validation theorem

For a given type, validation takes an external representation (an untyped value) into an internal representation (a value annotated with types). For a given type, we would like each external representation to correspond to just one internal representation, and conversely. We show that this is the case if the type is unambiguous, using a characterization of validation in terms of erasure and matching.

### 8.1 Ambiguity

Validation is a judgment that relates a type and an untyped value to a value.

$$\textsf{validate as } \mathit{Type} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} } \Rightarrow \mathit{Value}$$

In most of the examples we have seen, validation behaves as a partial function. That is, for a given type, for every untyped value, there is at most one value such that the above judgment holds. In this case, we say the type is *unambiguous for validation*. But just as there is more than one way to skin a cat, sometimes there is more than one way to validate a value.

Here is an example of an ambiguous complex type:

```
define element amb {
  element elt of type xs:integer |
  element elt of type xs:string
}

validate as amb { <amb><elt>1</elt></amb> }
⇒
  element amb { element elt of type xs:integer { 1 } }

validate as amb { <amb><elt>1</elt></amb> }
⇒
  element amb { element elt of type xs:string { "1" } }
```

Here is an example of an ambiguous simple type:

```
validate as xs:string * { "a b c" } ⇒ "a b c"
validate as xs:string * { "a b c" } ⇒ "a b", "c"
validate as xs:string * { "a b c" } ⇒ "a", "b c"
validate as xs:string * { "a b c" } ⇒ "a", "b", "c"
```

There are well-known algorithms for determining when regular expressions are ambiguous, and there are similar algorithms for regular tree grammars [16, 8]. These are easily adapted to give an algorithm for determining when a given type is ambiguous.

In Schema, the issue of ambiguity is resolved differently than here. Complex types are required to be unambiguous. Simple types have rules that resolve the ambiguity: every space is taken as a list separator, and in a union the first alternative that matches is chosen. Thus, for the first example above Schema deems the type illegal, while for the second example above Schema validation yields the last of the four possibilities.

Our formal model differs from Schema for two reasons. First, while Schema is concerned solely with validation against types written by a user, XQuery must also support type inference. And while it may be reasonable to require that a user write types that are unambiguous, it is not reasonable to place this restriction on a type inference system. For example, if expression $e_0$ has type xs:boolean and $e_1$ has type $t_1$ and $e_2$ has type $t_2$, the expression if $(e_0)$ then $e_1$ else $e_2$ has type $t_1 \,|\, t_2$, and it is not reasonable to require that $t_1$ and $t_2$ be disjoint.

Second, defining validation as a relation rather than a function permits a simple characterization of validation in terms of matching and erasure, as given in the next section.

Erasure is a judgement that relates a value to an untyped value.

$$\textit{Value}\ \textsf{\textbf{erases to}}\ \textit{UntypedValue}$$

Again, in most of the examples we have seen, erasure behaves as a function. That is, for a given value, there is exactly one untyped value such that the above judgement holds. Indeed, the only ambiguity arises when the value is an integer or contains an integer. This ambiguity occurs because there is more than one string represents the same integer, and hence there is more than one way to erase it. For example, the integer 7 is represented by both "7" and "007", and so has both of these as erasures. If a type does not contain any integers, then we say it is *unambiguous for erasure*.

### 8.2 The validation theorem

We can characterize validation in terms of erasure and matching.

**Theorem 1** *We have that*

$$\textsf{\textbf{validate as}}\ \textit{Type}\ \texttt{\{}\ \textit{UntypedValue}\ \texttt{\}} \Rightarrow \textit{Value}$$

*if and only if*

$$\textit{Value}\ \textsf{\textbf{matches}}\ \textit{Type}$$

*and*

$$\textit{Value}\ \textsf{\textbf{erases to}}\ \textit{UntypedValue}.$$

The proof is by induction over derivations.

We would like to know that if we convert an external value to an internal value (using validation) and then convert the internal value back to an external value (using erasure) that we end up back where we started, so long as the type is unambiguous for erasure. This follows immediately from the validation theorem.

**Corollary 1** *If*

$$\textsf{\textbf{validate as}}\ \textit{Type}\ \texttt{\{}\ \textit{UntypedValue}\ \texttt{\}} \Rightarrow \textit{Value}$$

*and*

$$\textit{Value}\ \textsf{\textbf{erases to}}\ \textit{UntypedValue}'$$

*and*

$$\textit{Type}\ \text{is unambiguous for erasure}$$

*then*

$$\textit{UntypedValue} = \textit{UntypedValue}'.$$

**Proof** From the first hypothesis and the validation theorem we have that

$$\textit{Value}\ \textsf{\textbf{erases to}}\ \textit{UntypedValue}$$

Taking this together with the second hypothesis and the fact that erasure is a function, the conclusion follows immediately. □

Similarly, we would like to know that if we convert an internal value of a given type to an external value (using erasure) and then convert the internal value back to an external value (using validation against that type) that we again end up back where we started, so long as the type is unambiguous. Again, this follows immediately from the validation theorem.

**Corollary 2** *If*

$$\textit{Value}\ \textsf{\textbf{matches}}\ \textit{Type}$$

*and*

$$\textit{Value}\ \textsf{\textbf{erases to}}\ \textit{UntypedValue}$$

*and*

$$\textsf{\textbf{validate as}}\ \textit{Type}\ \texttt{\{}\ \textit{UntypedValue}\ \texttt{\}} \Rightarrow \textit{Value}'$$

*and*

$$\textit{Type}\ \text{is unambiguous for validation}$$

then

$$\textit{Value} = \textit{Value}'.$$

**Proof** By the validation theorem, we have that the first two hypotheses are equivalent to

$$\textsf{\textbf{validate as}}\ \textit{Type}\ \texttt{\{}\ \textit{UntypedValue}\ \texttt{\}} \Rightarrow \textit{Value}$$

Taking this together with the third hypothesis and the fact that validate is a partial function when the type is unambiguous, the conclusion follows immediately. □

## 9 Sensible types

We can easily define a notion of structural subtyping, similar to that used in XDuce [10]. We say that one type is a subtype of another if every value that matches the first type also matches the second. When one type is declared to derive from another type by restriction, we expect that the first type should be a subtype of the second. This section defines subtyping and the rules to ensure that types are sensible.

A value is sensible if whenever it contains an element with a type annotation, then the value of the element matches the type in the annotation. This section also defines sensible values, and observes that the value returned by validation is always sensible.

### 9.1 Subtype

The judgment

$$\textit{Type}_1\ \textsf{\textbf{subtype}}\ \textit{Type}_2$$

holds if every value that matches the first type also matches the second. For example, element of type feet **subtype** element of type xs:integer

element author + **subtype** element author *

element of type bookType
**subtype**
element of type publicationType

Subtyping is the only judgement that is not defined by structural inference rules. Instead, it is defined by a logical equivalence. We have

$$\mathit{Type}_1 \ \textsf{\textbf{subtype}}\ \mathit{Type}_2$$

if and only if

$$\forall \mathit{Value}.\ \ \mathit{Value}\ \textsf{\textbf{matches}}\ \mathit{Type}_1 \Longrightarrow \mathit{Value}\ \textsf{\textbf{matches}}\ \mathit{Type}_2.$$

There are well-known algorithms for determining when one type is a subtype of another, based on tree automata [16, 8, 10].

## 9.2 Sensible type

The judgment

$$\mathit{Type}\ \textsf{\textbf{ok}}$$

holds if a type is sensible.

An element type is sensible if its type specifier is sensible.

$$\frac{\mathit{TypeSpecifier}\ \textsf{\textbf{ok}}}{\textsf{element}\ \mathit{ElementName}?\ \mathit{TypeSpecifier}\ \textsf{\textbf{ok}}}$$

An element type with no type specifier is sensible.

$$\frac{}{\textsf{element}\ \mathit{ElementName}?\ \ \textsf{\textbf{ok}}}$$

A sequence type is sensible if the types it contains are sensible.

$$\frac{\begin{array}{c}\mathit{Type}_1\ \textsf{\textbf{ok}}\\ \mathit{Type}_2\ \textsf{\textbf{ok}}\end{array}}{\mathit{Type}_1\ ,\ \mathit{Type}_2\ \textsf{\textbf{ok}}}$$

The sensibility rules for `()`, `?`, `+`, `*`, and `|` are similar.

## 9.3 Sensible type specifier

The judgment

$$\mathit{TypeSpecifier}\ \textsf{\textbf{ok}}$$

holds if a type specifier is sensible.

A reference to a global type is always sensible.

$$\frac{}{\textsf{of type}\ \mathit{TypeName}\ \textsf{\textbf{ok}}}$$

A restriction of an atomic type is always sensible.

$$\frac{}{\textsf{restricts}\ \mathit{AtomicTypeName}\ \textsf{\textbf{ok}}}$$

A restriction to a given type is sensible if the type is sensible and is a subtype of the base type.

$$\frac{\begin{array}{c}\textsf{define type}\ \mathit{TypeName}\ \mathit{TypeDerivation}\\ \mathit{TypeDerivation}\ \textsf{\textbf{resolves to}}\ \mathit{TypeName}'\ \{\ \mathit{Type}'\ \}\\ \mathit{Type}\ \textsf{\textbf{subtype}}\ \mathit{Type}'\\ \mathit{Type}\ \textsf{\textbf{ok}}\end{array}}{\textsf{restricts}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}\ \textsf{\textbf{ok}}}$$

An extension is sensible if the type is sensible.

$$\frac{\mathit{Type}\ \textsf{\textbf{ok}}}{\textsf{extends}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}\ \textsf{\textbf{ok}}}$$

## 9.4 Sensible definition

The judgment

$$\mathit{Definition}\ \textsf{\textbf{ok}}$$

holds if a type or element definition is sensible.

An element definition is sensible if its type specifier is sensible.

$$\frac{\mathit{TypeSpecifier}\ \textsf{\textbf{ok}}}{\textsf{define element}\ \mathit{ElementName}\ \mathit{TypeSpecifier}\ \textsf{\textbf{ok}}}$$

A type definition is sensible if its type derivation is sensible.

$$\frac{\mathit{TypeDerivation}\ \textsf{\textbf{ok}}}{\textsf{define type}\ \mathit{TypeName}\ \mathit{TypeDerivation}\ \textsf{\textbf{ok}}}$$

## 9.5 Sensible value

The judgment

$$\mathit{Value}\ \textsf{\textbf{ok}}$$

holds if a value is sensible.

The empty sequence is sensible.

$$\frac{}{()\ \textsf{\textbf{ok}}}$$

If two values are sensible, then their sequence is sensible.

$$\frac{\begin{array}{c}\mathit{Value}_1\ \textsf{\textbf{ok}}\\ \mathit{Value}_2\ \textsf{\textbf{ok}}\end{array}}{\mathit{Value}_1\ ,\ \mathit{Value}_2\ \textsf{\textbf{ok}}}$$

An element is sensible if the value is sensible, and if the value matches the annotated type.

$$\frac{\begin{array}{c}\textsf{define type}\ \mathit{TypeName}\ \mathit{TypeDerivation}\\ \mathit{TypeDerivation}\ \textsf{\textbf{resolves to}}\ \mathit{BaseTypeName}\ \{\ \mathit{Type}\ \}\\ \mathit{Value}\ \textsf{\textbf{ok}}\\ \mathit{Value}\ \textsf{\textbf{matches}}\ \mathit{Type}\end{array}}{\textsf{element}\ \mathit{ElementName}\ \textsf{of type}\ \mathit{TypeName}\ \{\ \mathit{Value}\ \}\ \textsf{\textbf{ok}}}$$

An atomic value is sensible.

$$\frac{}{\mathit{Atom}\ \textsf{\textbf{ok}}}$$

### 9.6 Sensibility theorem

Validation always yields a sensible value.

**Theorem 2** *(Sensibility theorem) If*

$$\textsf{\textbf{validate as}}\ \mathit{Type}\ \texttt{\{}\ \mathit{UntypedValue}\ \texttt{\}} \Rightarrow \mathit{Value}$$

*then*

$$\mathit{Value}\ \textsf{\textbf{ok}}$$

The proof is by induction on derivations.

## 10 Name vs. structure: the optimization theorem

Finally, we return to the relation between names and structures. The notion of matching defined in Section 5 is structural: it examines whether the structure of a value matches the structure of the corresponding type. There is a second notion of matching that is named: just check whether the type name that annotates an element is derived from a given type name. In this section, we consider when named matching can safely replace structural matching.

Recall the rule for matching an element against an element type.

$$\frac{\begin{array}{c}\mathit{ElementType}\ \textsf{\textbf{yields}}\ \mathit{BaseElementName}\ \mathit{TypeSpecifier}\\ \mathit{TypeSpecifier}\ \textsf{\textbf{resolves to}}\ \mathit{BaseTypeName}\ \texttt{\{}\ \mathit{Type}\ \texttt{\}}\\ \mathit{ElementName}\ \textsf{\textbf{substitutes for}}\ \mathit{BaseElementName}\\ \mathit{TypeName}\ \textsf{\textbf{derives from}}\ \mathit{BaseTypeName}\\ \mathit{Value}\ \textsf{\textbf{matches}}\ \mathit{Type}\end{array}}{\begin{array}{c}\textsf{element}\ \mathit{ElementName}\ \textsf{of type}\ \mathit{TypeName}\ \texttt{\{}\ \mathit{Value}\ \texttt{\}}\\ \textsf{\textbf{matches}}\ \mathit{ElementType}\end{array}}$$

(STRUCTURAL MATCH)

This rule checks not only that the named relationship holds (*TypeName* **derives from** *BaseTypeName*), but also that the structural relationship holds (*Value* **matches** *Type*). However, in common cases it is possible to check only the named relation, and skip checking the structural relation entirely.

This is reflected by the fact that the above rule simplifies considerably in the case that the type specifier is a reference to a global type. In such cases, we may replace it with the following rule.

$$\frac{\begin{array}{c}\mathit{ElementType}\ \textsf{\textbf{yields}}\ \mathit{BaseElementName}\ \textsf{of type}\ \mathit{BaseTypeName}\\ \mathit{ElementName}\ \textsf{\textbf{substitutes for}}\ \mathit{BaseElementName}\\ \mathit{TypeName}\ \textsf{\textbf{derives from}}\ \mathit{BaseTypeName}\end{array}}{\begin{array}{c}\textsf{element}\ \mathit{ElementName}\ \textsf{of type}\ \mathit{TypeName}\ \texttt{\{}\ \mathit{Value}\ \texttt{\}}\\ \textsf{\textbf{matches}}\ \mathit{ElementType}\end{array}}$$

(NAMED MATCH)

Here it is not necessary to check the structural relationship, because it is guaranteed by the way in which validation annotates elements with types.

The result that allows one to replace the first rule by the second is called the optimization theorem, because checking the structural relationship can be expensive (it requires a tree automaton or a recursive traversal), while checking the named relationship can be efficient (it just requires walking up the type hierarchy). The increased efficiency of named typing is one of the major motivations for pursuing its use in XQuery.

Recall that a value is sensible if whenever an element is annotated with a type then the value of the element matches that type. The optimization theorem applies only for sensible value.

**Theorem 3** *(Optimization theorem)* Consider computing *Value* **matches** *Type* when *Value* **ok**. Then whenever rule (NAMED MATCH) applies it may be used in place of (STRUCTURAL MATCH), without changing the result.

The proof is by induction over derivations.

The optimization theorem applies only when the element type in question contains a type reference (of type) rather than a type derivation (restricts, extends). In other words, named matching works only when there is a top-level type definition; if the type is anonymous, one must fall back on structural matching. For this reason, one design question currently faced by XQuery is whether to introduce some way of naming anonymous types, so that named matching may always be used in preference to structural matching.

### References

[1] Martin Abadi and Luca Cardelli *A Theory of Objects* Springer-Verlag, 1996.

[2] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation, February 1998.

[3] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL - a model for W3C XML Schema. In *Proceedings of International World Wide Web Conference*, pages 191–200, Hong Kong, China, 2001.

[4] James Clarke. XSL Transformations (XSLT) version 1.0. W3C Proposed Recommendation, October 1999.

[5] James Clarke. TREX — Tree Regular Expressions for XML. Thai Open Source Software Center, February 2001.

[6] James Clarke and Murata Makoto. RELAX NG specification. Oasis, December 2001.

[7] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 177–188, Seattle, Washington, June 1998.

[8] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1997.

[9] Carl Gunter and John Mitchell. *Theoretical Aspects of Object-Oriented Programming.* MIT Press, 1994.

[10] Haruo Hosoya and Benjamin C. Pierce. XDuce: an XML processing language. In *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 2000.

[11] Michael Kay. XSL Transformations (XSLT) version 2.0. W3C Working Draft, April 2002.

[12] Murata Makoto. Document description and processing languages – regular language description for XML (relax), October 2000.

[13] Peter Neumann. Risks to the public from the use of computers. *ACM Software Engineering Notes* 10(3), July 1985.

[14] Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, Dallas, Texas, May 2000.

[15] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[16] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages*. Springer-Verlag, 1997.

[17] Jérôme Siméon and Philip Wadler. The essence of XML (preliminary version) [invited talk]. In *International Symposium on Functional and Logic Programming (FLOPS)*, Aizu, Japan, September 2002. Springer-Verlag, 2002.

[18] Henri S. Thompson, David Beech, Murray Maloney, and N. Mendelsohn. XML Schema part 1: Structures. W3C Recommendation, May 2001.

[19] XPath 2.0. W3C Working Draft, April 2002.

[20] XQuery 1.0: An XML Query Language. W3C Working Draft, April 2002.

[21] XQuery 1.0 Formal Semantics. W3C Working Draft, March 2002.

[22] XQuery Formal Semantics FS Issue-0141 – FS Issue-0151.