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 of 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.

Categories and Subject Descriptors

F.3.2 [**Theory of Computation**]: Logics and meanings of programs—*Semantics of Programming Languages*

General Terms

Languages, Standardization, Theory

Keywords

XML, XML Schema, XPath, XQuery, validation

1. INTRODUCTION

XML is touted as an external format for representing data. This is not a hard problem. All we require are two properties:

- *Self-describing* From the external representation one should be able to derive the corresponding internal representation.
- *Round-tripping* If one converts from an internal representation to the external representation and back again, the new internal representation should equal the old.

Lisp S-expressions, for example, possess these properties.

XML has neither property. It is not always self-describing, since the internal format corresponding to an external XML description depends crucially on the *XML Schema* that is used for validation (for instance, to tell whether data is an integer or a string). And it is not always round-tripping, since some pathological Schemas lack this property (for instance, if there is a type union of integers and strings). So the essence of XML is this: the problem it solves is not hard, and it does not solve the problem well.

Nonetheless, XML and Schema are widely used standards, and there is value in modeling these standards. Here we provide a model for the core features of XML: XML values, XML types, and validation as described in XML Schema and used in XQuery and XPath.

Our model differs from previous models it two ways. First, our model captures aspects of named typing in XML Schema whereas previous models are purely structural. Second, Previous models sought to capture the notion of when a value *matches* a type. The model given here also captures how *validation* takes an external value into an internal value, and how *erasure* takes an internal value into an external value.

XML Schema is a large and complex standard – more than 300 pages in printed form. The main difficulty was to understand that the essence of XML Schema lies in the way type names and structures interact. Our first surprise has been to realize that once we captured named typing and validation, most of the myriad other features of XML Schema fit neatly into the simple framework presented here.

We present a theorem characterizing validation in terms of erasure and matching. Our second surprise has been to realize that, despite XML Schema's complexity, the resulting theorem turns to be simple.

Named and structural types. 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 [18, 19, 10, 1, 17].

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. [15].)

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 SML 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) [3].

There are a number of type systems for XML, including: DTDs, part of the original W3C recommendation defining XML [3]; XML Schema, a W3C recommendation which supersedes DTDs [22]; Relax NG, an Oasis standard [7]; Relax [13] and TREX [6], two ancestors of Relax NG; and the type systems of XDuce [11] and YATL [8]. 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 [5, 12]; XQuery, an analogue of SQL for XML data [24]; and XPath, the common core of XSLT and XQuery, which is jointly managed by the working groups responsible for the other two languages [23]. All three of these are functional languages. XSLT 1.0 and XPath 1.0 became W3C 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 [25], 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.

XQuery has both a specification in prose [24] and a formal semantics [25], 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 [26].

An earlier formal specification of XML Schema [4] was influenced by XDuce [11]; it ignored the named aspects of Schema and took a purely structural approach. The specification of Relax NG [7] also uses formal methods, also is purely structural, and was influenced by the earlier work on XML Schema [4].

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*

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

if and only if

$$\begin{array}{c} \mathit{Value} \text{ matches } \mathit{Type} \\ \mathit{Value} \text{ erases to } \mathit{UntypedValue}. \end{array}$$

(Here the stack of two judgments stands for their conjunction.)

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 [25]. 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 inferred types may be ambiguous (for instance, see the discussion of conditionals 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 [21]. This version has revised material in Sections 1, 4, 6, and 7, entirely new material in Section 9, 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 judgments. Section 5 describes matching. Section 6 describes erasure. Section 7 describes validation. Section 8 presents the validation theorem and its application to roundtripping. Section 9 describes subtyping, constraints on sensible types, and and optimized matching. Section 10 concludes.

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" }
}
```

Global and local declarations. Element declarations may be *global* — at the top-level — or *local* — nested within a type declaration. Here is the above description rewritten, with paper declared globally, while title and author are declared locally.

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

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 the configuration example below.

In XML Schema, the type of an element can be specified inline, without giving it a name, called an *anonymous type*. In such cases, we presume that the translation to our notation invents a suitable name. Thus every element has a named type, a property referred to in XQuery as *pure named typing*.

The combination of anonymous types and local types even further extends the power of the type system, so that types must be characterized using tree automata [16, 9]. Because we use pure named typing, every type has a global name and only traditional automata over words, not trees, are sufficient.

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.

```
define element ints of type intList
define type intList { xs:integer+ }
```

In XML notation, lists are written space-separated.

```
validate as ints { <ints>1 2 3</ints> }
⇒
element ints of type intsType { 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 fact of type intOrStrList
define type intOrStrList { ( xs:integer | xs:string ) * }

validate as fact { <fact>I saw 8 cats</fact> }
⇒
element fact of type intOrStrList "I", "saw", 8, "cats"
```

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 of type configurationType
define type configurationType {
  element shuttle of type shuttleType
  element laser of type laserType
}
define type shuttleType {
  element height of type miles
}
define type laserType {
  element height of type miles
}
```

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 of type configurationType {
  element shuttle of type shuttleType {
    element height of type miles { 120 }
  } ,
  element laser of type laserType {
    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 (
  $c as element configuration
) as 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 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
```

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 (
  $p as element of type publicationType
) as 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 (
  $p as element of type pointType
) as xs:integer {
  count($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 (
  $x as element of type xs:anyType
) as xs:integer {
  count($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 definition 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.

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

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 describe XML documents before validation. Every untyped value is a value.

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

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

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

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 [20, 9]. 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 (*).

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

An item type is an element type or an atomic type. Atomic types are specified by an *AtomicTypeName*;, 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 element name and an optional type name. An element name alone refers to a global declaration (element author). An element name with a type name is a local declaration (element author of type xs:string). A type name alone matches any element with a type derived from the specified type (element of type publicationType). The word element alone matches any element.

$$
\begin{array}{lcl}
\mathit{ElementType} & ::= & \textsf{element}\ \mathit{ElementName}?\ \mathit{OfType}?
\end{array}
$$

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

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

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.

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

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.

$$
\begin{array}{lcl}
\mathit{TypeDerivation} & ::= & \textsf{restricts}\ \mathit{AtomicTypeName} \\
 & | & \textsf{restricts}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \} \\
 & | & \textsf{extends}\ \mathit{TypeName}\ \{\ \mathit{Type}\ \}
\end{array}
$$

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

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

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.

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

The element type yields a base element name and a base type name, and the base type name resolves to a type. Then the given element matches the element type if three things hold: the element name must be substitutable for the base element name, 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

$$\mathit{ElementType} \textsf{ yields } \mathit{ElementName}\ \mathit{OfType}$$

takes an element type and yields an element name and a type name. We assume that the element names include a special wildcard name *, which is returned if the element type does not specify an element name. 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.)

$$\frac{\textsf{define element } \mathit{ElementName}\ \mathit{OfType}}{\textsf{element } \mathit{ElementName} \textsf{ yields } \mathit{ElementName}\ \mathit{OfType}}$$

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

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

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

$$\frac{}{\textsf{element } \mathit{OfType} \textsf{ yields } * \ \mathit{OfType}}$$

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

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

4.2 Resolution

The judgment

$$\mathit{TypeName} \textsf{ resolves to } \mathit{Type}$$

resolves a type name to a type. For example,

```
feet resolves to feet
```

and

```
  bookType
resolves to
  element author + ,
  element title ,
  element year
```

and

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

If the type name is defined by restriction of an atomic type, then return the atomic type.

$$\frac{\textsf{define type } \mathit{TypeName} \textsf{ restricts } \mathit{AtomicTypeName}}{\mathit{TypeName} \textsf{ resolves to } \mathit{AtomicTypeName}}$$

If the type name is defined by restriction of a non-atomic type, then return the restricted given type.

$$\frac{\textsf{define type } \mathit{TypeName} \textsf{ restricts } \mathit{BaseTypeName} \textsf{ \{ } \mathit{Type} \textsf{ \}}}{\mathit{TypeName} \textsf{ resolves to } \mathit{Type}}$$

If the type name is defined by extension, then resolve the base type name to get the base type, and return the concatenation of the base type and the given type.

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

4.3 Substitution

The judgment

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

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 wildcard element name *. For example:

```
book substitutes for book
book 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.

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

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

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

4.4 Derives

The judgment

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

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.

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

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

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

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

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

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

5. MATCHES

The judgment

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

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

```
  element configuration of type configurationType {
    element shuttle of type shuttleType {
      element height of type miles { 120 }
    } ,
    element laser of type laserType {
      element height of type feet { 10023 }
    }
  }
matches
  element configuration
```

and

```
  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
```

The empty sequence matches the empty sequence type.

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

6. ERASURE

The judgment

$$\textit{Value} \text{ erases to } \textit{UntypedValue}$$

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

```
  element configuration of type configurationType {
    element shuttle of type shuttleType {
      element height of type miles { 120 }
    } ,
    element laser of type laserType {
      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,

```
  element ints of type intsType { 1, 2, 3 }
erases to
  element ints  "1 2 3"
```

and

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

and

```
  element mixed of type xs:anyType {
    "I saw", "eight ", element em of type xs:string { "cats" }
  }
erases to
  element mixed { "I saw eight ", element em { "cats" } }
```

(In the last, note no space is inserted before the element.)

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.

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

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.

$$\frac{\begin{array}{c}\textit{Value}_1 \text{ erases to } \textit{UntypedValue}_1 \text{ , } \textit{String}_1 \\ \textit{Value}_2 \text{ erases to } \textit{String}_2 \text{ , } \textit{UntypedValue}_2 \\ \textit{String}_3 = \text{concat}(\textit{String}_1, \text{" "}, \textit{String}_2)\end{array}}{\begin{array}{c}\textit{Value}_1 \text{ , } \textit{Value}_2 \\ \text{erases to} \\ \textit{UntypedValue}_1 \text{ , } \textit{String}_3 \text{ , } \textit{UntypedValue}_2\end{array}}$$

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

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

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

A string erases to itself.

$$\frac{}{\textit{String} \text{ erases to } \textit{String}}$$

An integer erases to any string that represents it.

$$\frac{}{\text{integer-of-string}(\textit{String}) \text{ erases to } \textit{String}}$$

7. VALIDATION

The judgment

$$\text{validate as } \textit{Type} \text{ \{ } \textit{UntypedValue} \text{ \}} \Rightarrow \textit{Value}$$

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

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

and

```
validate as element ints {
  element ints { "1 2 3" }
} ⇒
  element ints of type intsType { 1, 2, 3 }
```

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

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

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.

$$\frac{\begin{array}{c}\mathsf{validate\ as}\ \mathit{Type}_1\ \{\ \mathit{UntypedValue}_1\ ,\ \mathit{String}_1\ \}\Rightarrow\ \mathit{Value}_1\\ \mathsf{validate\ as}\ \mathit{Type}_2\ \{\ \mathit{UntypedValue}_2\ ,\ \mathit{String}_2\ \}\Rightarrow\ \mathit{Value}_2\\ \mathit{String}_3=\mathsf{concat}(\mathit{String}_1,\text{" "},\mathit{String}_2)\end{array}}{\begin{array}{c}\mathsf{validate\ as}\ \mathit{Type}_1\ ,\ \mathit{Type}_2\ \{\\ \mathit{UntypedValue}_1\ ,\ \mathit{String}_3\ ,\ \mathit{UntypedValue}_2\\ \}\Rightarrow\ \mathit{Value}_1\ ,\ \mathit{Value}_2\end{array}}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The element type yields a base element name and a base type name, and the base type name resolves to a type. Then the given element validates against the element type if three things hold: the element name must be substitutable for the base element name, the type name must derive from the base type name, and the untyped value must validate against the type. The resulting element has the element name, the type name, and the validated value.

8. THE VALIDATION THEOREM

We characterize validation in terms of erasure and matching, and show that roundtripping holds for unambiguous types.

Unambiguous for validation.

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

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

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" } }
```

This type is permitted by our model, but prohibited by XML Schema. So in practice this sort of ambiguity does not arise.

Here is an example of an ambiguous simple type:

```
validate as intOrStrList { "one 2 3" } ⇒ "one 2 3"
validate as intOrStrList { "one 2 3" } ⇒ "one 2", 3
validate as intOrStrList { "one 2 3" } ⇒ "one", "2", 3
validate as intOrStrList { "one 2 3" } ⇒ "one", 2, 3
```

This type is permitted by our model, and also permitted by XML Schema. XML Schema requires that simple list types are always space separated, and that the simple union types always return the first of the possible unions. So XML Schema is unambiguous, in that it specifies that the last of the above interpretations must be chosen. Nonetheless, this sort of ambiguity does lead to practical problems, as discussed below.

Our formal model differs from Schema for two reasons. First, while Schema is concerned solely with validation of a value against a user defined type, 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 below.

Unambiguous for erasure. Erasure is a judgment that relates a value to an untyped value.

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

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 judgment 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*.

The validation theorem. We can characterize validation in terms of erasure and matching.

THEOREM 1. *(Validation) We have that*

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

if and only if

$$\begin{array}{c} \mathit{Value} \ \textsf{matches} \ \mathit{Type} \\ \mathit{Value} \ \textsf{erases to} \ \mathit{UntypedValue}. \end{array}$$

PROOF. By induction over derivations. □

Roundtripping. 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 external value back to an internal value (using validation against that type) that we again end up back where we started, so long as the type is unambiguous. This follows immediately from the validation theorem.

COROLLARY 1. *(Roundtripping) If*

$$\begin{array}{c} \mathit{Value} \ \textsf{matches} \ \mathit{Type} \\ \mathit{Value} \ \textsf{erases to} \ \mathit{UntypedValue} \\ \textsf{validate as} \ \mathit{Type} \ \{ \ \mathit{UntypedValue} \ \} \Rightarrow \ \mathit{Value}' \\ \mathit{Type} \ \text{is unambiguous for validation} \end{array}$$

then

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

PROOF. By the validation theorem, we have that the first two hypotheses are equivalent to

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

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. □

For example, we have

```
  element author of type string { "John Reynolds" }
erases to
  <author>John Reynolds</author>
```

and

```
validate as element author of type string {
  <author>John Reynolds</author>
} ⇒
  element author of type string { "John Reynolds" }
```

which satisfies roundtripping.

On the other hand, we have

```
  element fact of type intOrStr { "one", "2", 3 }
erases to
  <fact>one 2 3</fact>
```

and

```
validate as element fact of type intOrStr {
  <fact>one 2 3</fact>
} ⇒
  element fact of type intOrStr { "one", 2, 3 }
```

which does *not* satisfy roundtripping. Because XML Schema prohibits ambiguous complex types, the only counterexamples to roundtripping involve simple types that are lists or unions. Users will stub their toes on this rarely, but when they do it will hurt!

Reverse roundtripping. Similarly, 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. Again, this follows immediately from the validation theorem.

COROLLARY 2. *(Reverse roundtripping) If*

$$\begin{array}{c} \textsf{validate as} \ \mathit{Type} \ \{ \ \mathit{UntypedValue} \ \} \Rightarrow \ \mathit{Value} \\ \mathit{Value} \ \textsf{erases to} \ \mathit{UntypedValue}' \\ \mathit{Type} \ \text{is unambiguous for erasure} \end{array}$$

then

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

PROOF. From the first hypothesis and the validation theorem we have that

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

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

For example, we have

```
validate as element author of type string {
  <author>John Reynolds</author>
} ⇒
  element author of type string { "John Reynolds" }
```

and

```
  element author of type string { "John Reynolds" }
erases to
  <author>John Reynolds</author>
```

which satisfies reverse roundtripping.

On the other hand, we have

```
validate as element height of type feet {
  <height>007</height>
} ⇒
  element height of type feet { 7 }
```

and

```
  element height of type feet { 7 }
erases to
  <height>7</height>
```

which does *not* satisfy reverse roundtripping.

The only counterexamples to reverse roundtripping involve leading zeros or other cases where base types have multiple representations. Unlike roundtripping, this is not a serious problem.

9. SENSIBILITY

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 defines sensible values, and observes that the value returned by validation is always sensible. This fact can be used to optimize matching.

9.1 Subtype

We can easily define a notion of structural subtyping, similar to that used in XDuce [11]. 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.

The judgment

$$\mathit{Type}_1 \ \mathsf{subtype} \ \mathit{Type}_2$$

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 judgment that is not defined by structural inference rules. Instead, it is defined by a logical equivalence. We have

$$\mathit{Type}_1 \ \mathsf{subtype} \ \mathit{Type}_2$$

if and only if for every *Value* we have

$$\mathit{Value} \ \mathsf{matches} \ \mathit{Type}_1 \text{ implies } \mathit{Value} \ \mathsf{matches} \ \mathit{Type}_2.$$

Subtyping can be checked by straightforward modification to well-known algorithms for checking for inclusion between the languages generated by two regular expressions [2].

Other type systems for XML (such as XDuce or YATL) require subtyping algorithms based on tree regular expressions and finite state tree automata [11, 14, 20, 9]. Here we can use ordinary regular expressions and ordinary finite state automata because pure named typing guarantees that one can check that an element matches against a type by looking only at the element name (not the element contents) as described in 9.4.

9.2 Sensible definition

The judgment

$$\mathit{Definition} \ \mathsf{ok}$$

holds if an element or type definition is sensible.

An element definition is always sensible.

$$\frac{}{\mathsf{define\ element} \ \mathit{ElementName} \ \mathit{OfType} \ \mathsf{ok}}$$

A restriction of an atomic type is always sensible.

$$\frac{}{\mathsf{define\ type} \ \mathit{TypeName} \ \mathsf{restricts} \ \mathit{AtomicTypeName} \ \mathsf{ok}}$$

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

$$\frac{\begin{array}{c}\mathit{BaseTypeName} \ \mathsf{resolves\ to} \ \mathit{BaseType} \\ \mathit{Type} \ \mathsf{subtype} \ \mathit{BaseType}\end{array}}{\mathsf{define\ type} \ \mathit{TypeName} \ \mathsf{restricts} \ \mathit{BaseTypeName} \ \{\ \mathit{Type}\ \} \ \mathsf{ok}}$$

An extension is always sensible.

$$\frac{}{\mathsf{define\ type} \ \mathit{TypeName} \ \mathsf{extends} \ \mathit{BaseTypeName} \ \{\ \mathit{Type}\ \} \ \mathsf{ok}}$$

9.3 Sensible value

The judgment

$$\mathit{Value} \ \mathsf{ok}$$

holds if a value is sensible.

The empty sequence is sensible.

$$\frac{}{() \ \mathsf{ok}}$$

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

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

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

$$\frac{\begin{array}{c}\mathit{TypeName} \ \mathsf{resolves\ to} \ \mathit{Type} \\ \mathit{Value} \ \mathsf{ok} \\ \mathit{Value} \ \mathsf{matches} \ \mathit{Type}\end{array}}{\mathsf{element} \ \mathit{ElementName} \ \mathsf{of\ type} \ \mathit{TypeName} \ \{\ \mathit{Value}\ \} \ \mathsf{ok}}$$

An atomic value is sensible.

$$\frac{}{\mathit{Atom} \ \mathsf{ok}}$$

9.4 Optimized matching

Validation always yields a sensible value.

Theorem 2. *(Sensibility) If*

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

then

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

Proof. By induction on derivations. □

As a corollary, matching against an element can be greatly simplified.

Corollary 3. *(Optimized matching) In the rule*

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

the hypothesis Value matches *Type need not be tested if the element is the result of validation.*

Optimized matching is both easier to implement and more efficient to execute. The simplification is a consequence of pure named typing, which ensures that every validated element contains a type name that accurately characterizes its contents.

10. CODA

In October 2002 the XQuery working group decided to adopt pure named typing. The final step before adoption was a presentation describing how pure named typing simplifies the formal semantics. The presentation was followed by unanimous agreement to adopt pure named typing. In the two-day meeting, this was the only decision adopted without dissent — a resounding demonstration of the value of formal semantics!

11. REFERENCES

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

[2] Alfred Aho, John Hopcroft, Jeffrey Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

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

[4] 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.

[5] James Clarke. XSL Transformations (XSLT) version 1.0. W3C Recommendation, November 1999.

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

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

[8] 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.

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

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

[11] 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.

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

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

[14] Frank Neven. Automata Theory for XML Researchers. *SIGMOD Record* 31(3), September 2002.

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

[16] 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.

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

[18] John C. Reynolds. Syntactic Control of Interference In *Fifth ACM Symposium on Principles of Programming Languages*, Tucson, AZ, Jan 1978, pages 36–49.

[19] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages.* North-Holland, Amsterdam, 1981, pages 345-372.

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

[21] 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.

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

[23] XPath 2.0. W3C Working Draft, November 2002.

[24] XQuery 1.0: An XML Query Language. W3C Working Draft, November 2002.

[25] XQuery 1.0 Formal Semantics. W3C Working Draft, November 2002.

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