

CS203 Programming Languages – Fall 2004

XML XPath Formal Semantics
XPath Type System

Emmanuel Pontikakis

Contents

1 Introduction	3
1.1 Why Formal Semantics?	3
1.2 Why Formal Notations?	3
2 XPath basics	3
3 XPath Processing Model	5
3.1 Static analysis sub-phases	6
4 The XPath Type System	7
4.1 XML Schema and XPath type system	7
4.2 Formal Types of Items	8
4.3 Content models	8
4.4 Top level definitions	9
4.5 SequenceType	9
4.6 SequenceType Matching	10
5 Raising Dynamic Errors	12
6 Typeswitch Expression	12
7 Accessing types	13
8 Type matching	15
8.1 Subtyping Semantics of XPath	16
9 Complex Types	16
9.1 Global complex type	16
9.2 Local complex type	17
9.3 Complex types with simple content	17
10 A Coq-based approach for XPath Formal Semantics	18
10.1 Modeling Containment Relation between two expressions	19
10.2 Path Equivalence Relation	20
11 Conclusions	20
12 References	21

1 Introduction

XML standard for representing documents is adopted more and more these days. XML documents are represented in a tree structure and providing information within their nodes. XML model is hierarchical in nature and in order to query this hierarchical structure there have been developed query languages such as XQuery and XPath. XPath (currently in the version 2.0) [1] is an expression language that allows the processing of values conforming to the data model. The data model except that it provides a tree representation of XML documents, it also provides values such as integers, Booleans and strings and sequences that may contain both references to nodes in an XML document and simple values. An XPath expression may result to a selection of nodes from the input documents, or a value, or any sequence allowed by the data model. XPath 2.0 is a superset of XPath 1.0, with the added capability to support a richer set of data types, such as complex data types, and to take advantage of the type information that becomes available when documents are validated using XML Schema. In this survey we will explore the type system of XPath and we will give some examples to illustrate how it works.

1.1 Why Formal Semantics?

Formal semantics are used to complement the XPath specification [2], defining the meaning of XPath expressions with mathematical notation. This rigor definition can be provided as a reference for implementation of XPath and in addition defines strictly the limits and the capabilities of the language.

1.2 Why Formal Notations?

Rigor is achieved by the use of formal notations to represent XPath objects such as expressions, XML values, and XML Schema types, and by the systematic definition of the relationships between those objects to reflect the meaning of the language. Dynamic semantics are used to map XPath expressions to the value that they evaluate. Static semantics infer the type of an expression or an error if the expression is not structured appropriately. This step can help to catch some type errors early in the static analysis. Formal Semantics uses several kinds of formal notations to define the relationships between XPath expressions, XML values, and XML Schema types.

2 XPath basics

The basic building block of XPath is the expression, which is a string of unicode characters. The language provides several kinds of expressions which may be constructed from keywords, symbols, and operands. Operands of an expression might be other expressions or sub-expressions. XPath is a functional language, which means that expressions can be nested. XPath is also a strongly-typed language in which the operands of various expressions, operators, and functions must conform to the expected types. An example of an XPath query is given below:

/authors/author/paper/title

This expression once executed will return all the titles of each author's papers. *authors* is the first node under the root of the tree and *author* is a child of *authors*, *paper* is a child of *author* and *title* is a child of *paper*.

/authors/author/paper/keywords[wireless]/abstract

The above expression will return the abstracts of papers that in their keywords we can find something about *wireless* and in that representation, *wireless* must be a node which is a child of *keywords*. The above two expressions are defined on the exact path of the tree. On the other hand, we can have relative expressions, which are expressions that are defined on a relative path on the tree. For example the expression below will return all the abstracts of the author's papers without caring if the ancestor of a paper is the node *author*. These expressions are starting with double *//*.

//paper/keywords[wireless]/abstract

Here we can see how an XML tree is structured in general and which is the relation between ancestors, descendants, parents, children, preceding-sibling and following-sibling nodes.

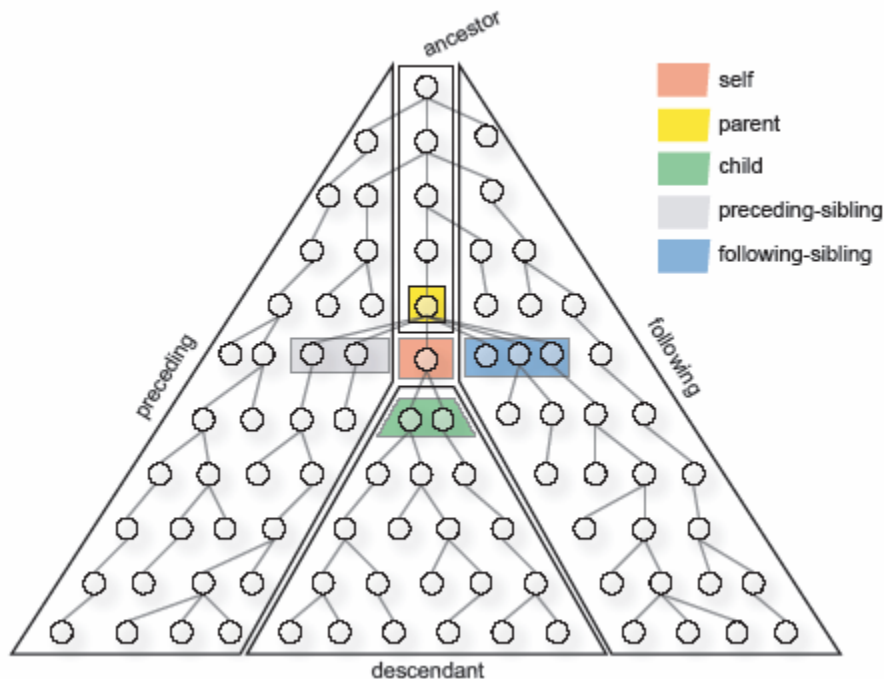


Figure 1: XML nodes relationship

3 XPath Processing Model

A schematic overview of the processing steps is given in Figure 2. XPath expression processing is presented inside the bounded area. The outside area represents the external processing steps which are not part of XPath. In external processing, data model instance represents the data to be queried, the schema import processing and serialization phase, which will not be presented in detail in this survey. The XPath processing domain, includes the static analysis and dynamic evaluation phases. Static analysis is further divided into four sub-phases. Each phase takes as input the results of the previous phase generates output for the next phase. The static analysis phase can be performed before examining any input XML document and detects many errors early, for example, syntax errors or type errors.

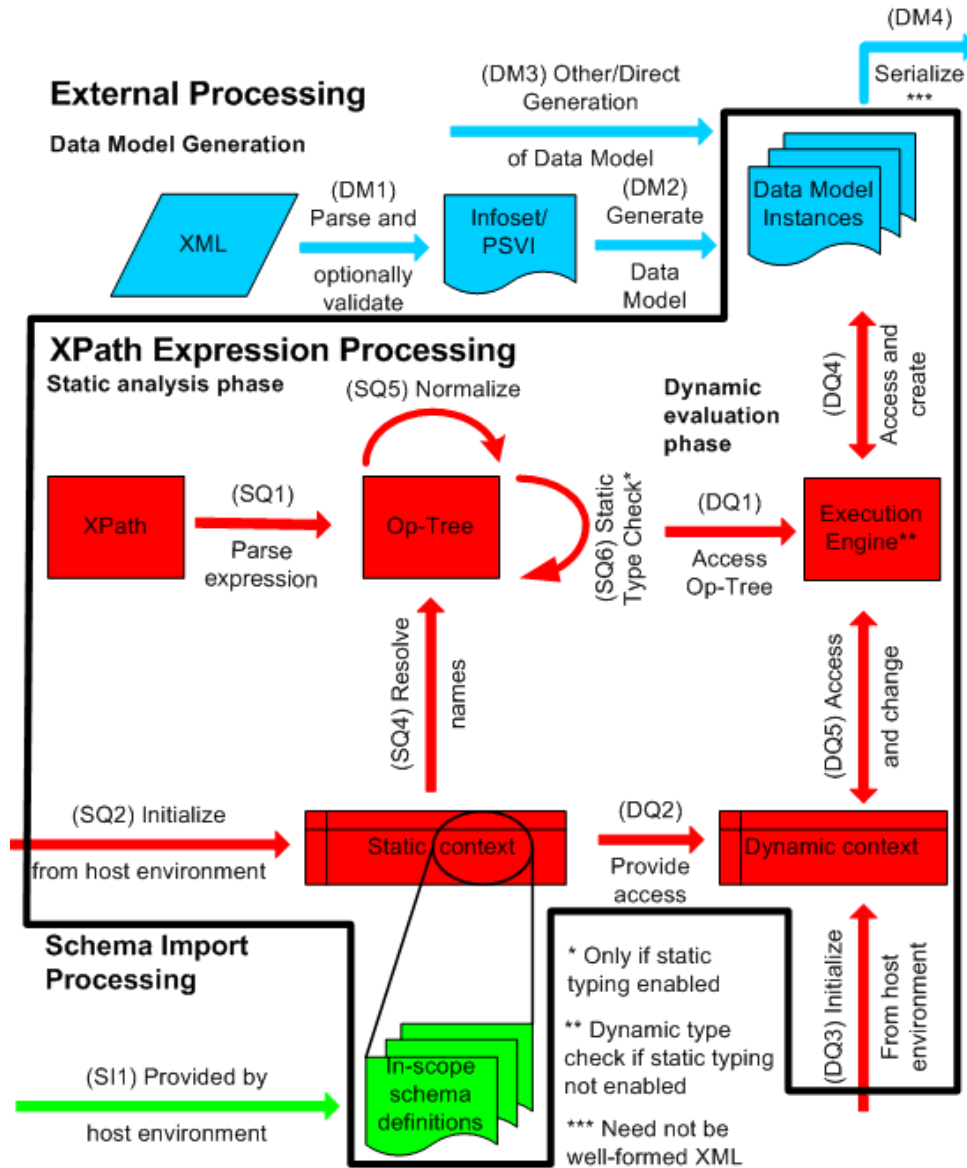


Figure 2: Processing Model Overview

3.1 Static analysis sub-phases:

1. **Parsing.** (Step SQ1 in Figure 2). In that phase, some syntactic errors may be generated, otherwise a query tree will be created and stored in memory.
2. **Static Context Processing.** (Steps SQ2, SQ3, and SQ4 in Figure 2). The static semantics of an expression depends on the static input context (or the static environment). The static environment must be generated before the expression can be analyzed. In XPath, the static environment is defined by the processing environment and it is denoted by *statEnv*.
3. **Normalization.** (Step SQ5 in Figure 2). The semantics of the expression can sometimes be redundant or complex and must be simplified before the expression is executed. For instance, a complex `for` expression might be expressed

differently by a composition of several simple `for` expressions. This procedure is called normalization and it maps the input expression to an equivalent core expression. The language composed of these simpler expressions is called the XPath/XQuery *Core language* and is described by a grammar which is a subset of the XQuery grammar.

4. **Static type analysis.** (Step SQ6 in Figure 2). Static type analysis checks whether each expression is type safe in the static environment, and if so, determines its static type. It works by bottom-up application of type inference rules over expressions, taking the type of literals and of input documents into account. If the expression does not pass from the check then it means that it is not type-safe and a *type error* will be raised. If the expression is type-safe then it is created an abstract syntax tree where each sub-expression is "annotated" with its static type.
5. **Dynamic Context Processing.** (Steps DQ2 and DQ3 in Figure 2). The dynamic semantics of the expression depends on the dynamic input context (or dynamic environment) which is denoted by *dynEnv*. It is generated before the evaluation of the expression. In XPath, the dynamic input context is defined by the processing environment.
6. **Dynamic Evaluation.** (Steps DQ4 and DQ5 in Figure 2). This phase computes the results of an expression. The semantics of evaluation are defined only for Core expression terms. Evaluation works by bottom-up application of evaluation rules over expressions, starting with evaluation of literals and variables. If the expression is correct evaluation results in a value or a sequence of values, otherwise it will raise a dynamic error, which may be a non-type error.

Static type analysis catches only some obvious type errors certain classes of errors. For instance, it can detect a type mismatch in a comparison operation applied between incompatible types (e.g., `xs:int` and `xs:date`). But some other errors, such as (an arithmetic expression on 32 bits integers (`xs:int`) yields an out-of-bound), cannot be detected during the static analysis phase but only at run-time by looking at the data. In the next section we will give greater detail about the type system of XPath language.

4 The XPath Type System

The type system of XPath is based on XML Schema and it is used to specify the dynamic and static semantics of XPath.

4.1 XML Schema and XPath type system

The relation between XML Schema and the formal XPath type system is given shortly by the following rules:

- Formal Semantics uses formal notations for types instead of XML Schema syntax.
- Formal notations are not exposed to the user.
- All anonymous types are made explicit for the clarification of the semantics.

- XML Schema identity constraints are not formally represented in the XPath type system.

4.2 Formal Types of Items

In Table 1 we describe in BNF formation all the possible item types of XML XPath query language. An item type is either an atomic type, an element type, an attribute type, a document node type, a text node type, a comment node type, or a processing instruction type.

ItemType	::=	AtomicTypeName NodeType
AtomicTypeName	::=	QName
NodeType	::=	ElementType AttributeType DocumentType "comment" "processing-instruction" "text"
ElementType	::=	"element" ElementName? TypeSpecifier?
TypeSpecifier	::=	Nillable? TypeReference
AttributeType	::=	"attribute" AttributeName? TypeReference?
Nillable	::=	"nillable"
TypeReference	::=	<"of" "type"> TypeName
DocumentType	::=	"document" ("{" Type? "}")?

Table 1. XPath Items

A type reference for an element or attribute type is optional. A name alone corresponds to a reference to a global element or attribute declaration. A name with a type reference corresponds to a local element or attribute declaration. The nillable flag of an element type indicates whether the element can be nilled or not. For a document type if no content type is given, then it refers to the “wildcard type” describing any generic document.

4.3 Content models

In general types in XPath are constructed from item types by sequencing ItemTypes. On Table 2, this architecture is illustrated. The type `empty` matches the empty sequence and the type `none` matches no values. The type `none` is raised from the error function where there is an error on the type of an expression.

Type	::=	ItemType
		(Type Occurrence)
		(Type "&" Type)
		(Type ", " Type)
		(Type " " Type)
		"empty"

			"none"
Occurrence	::=	"*" "+" "?"	

Table 2. Type Operators

The XPath type system has three binary operators for declaring types: The "," operator for sequence, the "|" operator for choice and the "&" operator for declaring types in any possible order. In addition, the type system includes three unary operators on types: "*", "+", and "?", corresponding respectively to zero or more instances of the type, one or more instances of the type, or an optional instance of the type.

4.4 Top level definitions

XPath allows the user to define his own types, elements and attributes and give derivations about the types that he defined. The table 3 shows how we can define new elements and types and give the system information about what kind of type we declared (such as string, integer, complex etc.)

Definition	::=	(<"define" "element"> ElementName Substitution? Nillable? TypeReference) (<"define" "attribute"> AttributeName TypeReference) (<"define" "type"> TypeName TypeDerivation)
Substitution	::=	<"substitutes" "for"> ElementName
TypeDerivation	::=	ComplexTypeDerivation AtomicTypeDerivation
ComplexTypeDerivation	::=	Derivation? Mixed? "{" Type? "}"
AtomicTypeDerivation	::=	"restricts" AtomicTypeName
Derivation	::=	("restricts" TypeName) ("extends" TypeName)
Mixed	::=	"mixed"

Table 3. Type declarations

When we define a new type we have to give a name and a type derivation. When the type we define is complex, derivation indicates if the type is derived by extension or restriction from its based type, gives its content model, and an optional flag indicating if it has mixed content. A type is derived either by extension or restriction from a base type.

4.5 SequenceType

SequenceTypes can be used in XPath to refer to a type imported from an XML Schema. SequenceTypes are used to declare the types of function parameters and in several kinds of XPath expressions. The grammar productions of SequenceTypes is described in Table 4.

SequenceType	::=	(ItemType OccurrenceIndicator? <"empty" "(" ")">
OccurrenceIndicator	::=	"?" "*" "+"
ItemType	::=	AtomicType KindTest <"item" "(" ")">
AtomicType	::=	QName
KindTest	::=	DocumentTest ElementTest AttributeTest ProcessingInstructionTest CommentTest TextTest AnyKindTest
ProcessingInstructionTest	::=	<"processing-instruction" "("> StringLiteral? ")"
CommentTest	::=	<"comment" "("> ")"
TextTest	::=	<"text" "("> ")"
AnyKindTest	::=	<"node" "("> ")"
DocumentTest	::=	<"document-node" "("> ElementTest? ")"
ElementTest	::=	<"element" "("> ((SchemaContextPath LocalName (NodeName ("," TypeName "nillable"?)?)? ")"
AttributeTest	::=	<"attribute" "("> ((SchemaContextPath "@" LocalName ("@" NodeName ("," TypeName?)?)? ")"
SchemaContextPath	::=	<SchemaGlobalContext "/"> <SchemaContextStep "/">*
SchemaGlobalContext	::=	QName SchemaGlobalTypeName
SchemaContextStep	::=	QName
SchemaGlobalTypeName	::=	"type" "(" QName ")"
LocalName	::=	QName
NodeName	::=	QName "*"
TypeName	::=	QName "*"

Table 4. Sequence Type grammar productions

4.6 SequenceType Matching

Sequence type matching is defined between a value and a type. When a query is being processed, it is necessary to determine if a given value matches a type that was declared using the SequenceType syntax. In order to normalize SequenceTypes to the XPath type system, the following auxiliary mapping rule specifies that *SequenceType* is mapped to *Type*, in the XPath type system.

$[SequenceType]_{sequencetype} == Type$

OccurrenceIndicators are left unchanged when normalizing SequenceTypes into [XPath/XQuery] types. Each kind of SequenceType component is normalized separately into the [XPath/XQuery] type system.

$[ItemType OccurrenceIndicator]_{sequencetype} == [ItemType]_{sequencetype} OccurrenceIndicator$

In the following examples we illustrate the normalization of elements and attributes.

$[element(ElementName)]_{sequencetype} == element ElementName$

$[attribute(@AttributeName)]_{sequencetype} == attribute AttributeName$

$[element(ElementName, TypeName)]_{sequencetype} == element ElementName of type TypeName$

$[attribute(@AttributeName, TypeName)]_{sequencetype} == attribute AttributeName of type TypeName$

Document nodes, text nodes, and atomic types are left unchanged.

$[text()]_{sequencetype} == text$ $[document()]_{sequencetype} == document$ $[AtomicType]_{sequencetype} == AtomicType$

The SequenceType components "node()" and "item()" correspond to wildcard types and can be mapped to different types in each environment. node() indicates that any node is allowed and item() indicates that any node or atomic value is allowed.

$[node()]_{sequencetype} == (element | attribute | text | document | comment | processing-instruction)$

$[item()]_{sequencetype} == (element | attribute | text | document | comment | processing-instruction | xdt:anyAtomicType)$

5 Raising Dynamic Errors

Expressions can raise errors during static analysis or dynamic evaluation. In addition, a user may raise an error explicitly by calling the function `fn:error`. There is a distinction between a static error, a type error, denoted by *typeError*, and a generic dynamic error, denoted by *dynError*. A static error is raised during static analysis. When an error is raised during evaluation of some expression *Expr₁*, the error is propagated to the expression *Expr₂* in which *Expr₁* is evaluated. The expression *Expr₂*, in turn, propagates the error to the expression in which *Expr₂* is evaluated, and so on, until the error is returned to the query environment. The following inference rule specifies that if any sub-expression *Expr_i* of expression *Expr* raises an error *dynError* then *Expr* also raises *dynError*.

$$\frac{\text{dynEnv} \mid - \text{Expr}_i \text{ raises } \text{dynError} \quad \text{Expr}_i \text{ is any subexpression of } \text{Expr}}{\text{dynEnv} \mid - \text{Expr} \text{ raises } \text{dynError}}$$

6 Typeswitch Expression

The definition of typeswitch expression is given in the following table:

```
TypeswitchExpr ::= <"typeswitch" "(" Expr ")" CaseClause+ "default"
                (" $" VarName)? "return" ExprSingle
CaseClause     ::= "case" (" $" VarName "as")? SequenceType "return"
                Expr
```

Using *typeswitch expression* users can perform different operations according to the type of an input expression. Each branch of a *typeswitch expression* may have an optional *Variable*, which assigns a specific type for the corresponding branch. Normalizing of a typeswitch expression guarantees that every branch has an associated *Variable*, otherwise an error is being raised. The following normalization rule adds *\$fs:new* which is a newly generated variable that must not conflict with any variables already in scope in order to be valid.

$$\begin{aligned} & [\text{ case } \text{SequenceType} \text{ return } \text{Expr}]_{\text{Case}} \\ & \quad == \\ & \text{case } \$fs:\text{new}_1 \text{ as } \text{SequenceType} \text{ return } [\text{Expr}]_{\text{Expr}} \\ & [\text{ case } \text{Variable} \text{ as } \text{SequenceType} \text{ return } \text{Expr}]_{\text{Case}} \\ & \quad == \\ & \text{case } \text{Variable} \text{ as } \text{SequenceType} \text{ return } [\text{Expr}]_{\text{Expr}} \\ & [\text{ default return } \text{Expr}]_{\text{Case}} \\ & \quad == \\ & \text{default } \$fs:\text{new}_1 \text{ return } [\text{Expr}]_{\text{Expr}} \\ & [\text{ default } \text{Variable} \text{ return } \text{Expr}]_{\text{Case}} \end{aligned}$$

$$\begin{array}{c}
\text{default } \textit{Variable} \text{ return } [\textit{Expr}]_{\textit{Expr}} \\
\text{====} \\
[\textit{typeswitch} (\textit{Expr}_0) \\
\quad \textit{CaseClause}_1 \\
\quad \dots \\
\quad \textit{CaseClause}_n \\
\text{default } (\textit{Variable})? \text{ return } \textit{Expr}_{n+1}]_{\textit{Expr}} \\
\text{====} \\
\textit{typeswitch} ([\textit{Expr}_0]_{\textit{Expr}}) \\
[\textit{CaseClause}_1]_{\textit{Case}} \\
\quad \dots \\
[\textit{CaseClause}_n]_{\textit{Case}} \\
[\text{ default } (\textit{Variable})? \text{ return } \textit{Expr}_{n+1}]_{\textit{Case}}
\end{array}$$

The following judgment is used in the static checking of `typeswitch`.

$$\text{statEnv} \mid - \textit{Type}_1 \text{ case } \textit{CaseClause} : \textit{Type}$$

It indicates that under the environment `statEnv`, and with the input type of the `typeswitch` being `Type1`, the given case rule yields the type `Type`. Each case clause and the default clause of the `typeswitch` is typed independently. The type of the entire `typeswitch` expression is the union of the types of all the clauses. Below are given the derivations of that procedure.

$$\begin{array}{c}
\text{statEnv} \mid - \textit{Expr}_0 : \textit{Type}_0 \\
\text{statEnv} \mid - \textit{Type}_0 \text{ case } \textit{Variable}_1 \text{ as } \textit{SequenceType}_1 \text{ return } \textit{Expr}_1 : \textit{Type}_1 \\
\quad \dots \\
\text{statEnv} \mid - \textit{Type}_0 \text{ case } \textit{Variable}_n \text{ as } \textit{SequenceType}_n \text{ return } \textit{Expr}_n : \textit{Type}_n \\
\text{statEnv} \mid - \textit{Type}_0 \text{ case } \textit{default } \textit{Variable}_{n+1} \text{ return } \textit{Expr}_n : \textit{Type}_{n+1} \\
\hline
\text{statEnv} \mid - (\textit{typeswitch} (\textit{Expr}_0) \\
\quad \textit{case } \textit{Variable}_1 \text{ as } \textit{SequenceType}_1 \text{ return } \textit{Expr}_1 \\
\quad \dots \\
\quad \textit{case } \textit{Variable}_n \text{ as } \textit{SequenceType}_n \text{ return } \textit{Expr}_n \\
\quad \textit{default } \textit{Variable}_{n+1} \text{ return } \textit{Expr}_{n+1}) \\
: (\textit{Type}_1 \mid \dots \mid \textit{Type}_{n+1})
\end{array}$$

7 Accessing types

We will explore some auxiliary judgments in order to access components of the XPath type system. The first two judgements (`derives from` and `substitutes for`) are used to access the type and element name hierarchies from an XML Schema. There are also some other judgments which we will not cover in this survey (`lookup`, `static lookup`, `extended by`, `adjusts to` and `expands to`) which are used to lookup the meaning of

element or attribute types from the schema. The *derives from* judgment holds when the first type name derives from the second type name.

$$\text{statEnv} \mid - \text{TypeName}_1 \text{ **derives from** } \text{TypeName}_2$$

It is easy to see that every type name derives from itself.

$$\text{statEnv} \mid - \text{TypeName} \text{ **derives from** } \text{TypeName}$$

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

$$\begin{array}{l} \text{statEnv.typeDefn}(\text{TypeName}) \Rightarrow \\ \text{define type } \text{TypeName} \text{ extends } \text{BaseTypeName} \text{ Mixed? } \{ \text{Type? } \} \end{array}$$

$$\text{statEnv} \mid - \text{TypeName} \text{ **derives from** } \text{BaseTypeName}$$

$$\begin{array}{l} \text{statEnv.typeDefn}(\text{TypeName}) \Rightarrow \\ \text{define type } \text{TypeName} \text{ restricts } \text{BaseTypeName} \text{ Mixed? } \{ \text{Type? } \} \end{array}$$

$$\text{statEnv} \mid - \text{TypeName} \text{ **derives from** } \text{BaseTypeName}$$

Derivation is transitive between types.

$$\begin{array}{l} \text{statEnv} \mid - \text{TypeName}_1 \text{ **derives from** } \text{TypeName}_2 \\ \text{statEnv} \mid - \text{TypeName}_2 \text{ **derives from** } \text{TypeName}_3 \\ \hline \text{statEnv} \mid - \text{TypeName}_1 \text{ **derives from** } \text{TypeName}_3 \end{array}$$

The *substitutes* judgment is used to know whether an element name is in the substitution group of another element name. The judgment

$$\text{statEnv} \mid - \text{ElementName} \text{ **substitutes for** } \text{ElementName}$$

holds when the first element name substitutes for the second element name. Substitution is a partial order. It is reflexive and transitive by the definition below. It is asymmetric because no cycles are allowed in substitution groups. The *substitutes* judgment for element names is specified by the following rules. Every element name substitutes for itself.

$$\text{statEnv} \mid - \text{ElementName} \text{ **substitutes for** } \text{ElementName}$$

Every element name substitutes for the element it is declared to substitute for.

$$\text{statEnv.elemDecl}(\text{ElementName}) \Rightarrow \text{define element } \text{ElementName} \text{ substitutes for } \text{BaseElementName} \text{ Nillable? } \text{TypeReference}$$

$$\text{statEnv} \mid - \text{ElementName} \text{ **substitutes for** } \text{BaseElementName}$$

Substitution is transitive.

$$\frac{\begin{array}{l} \text{statEnv} \mid - \text{ElementName}_1 \text{ **substitutes for** } \text{ElementName}_2 \\ \text{statEnv} \mid - \text{ElementName}_1 \text{ **substitutes for** } \text{ElementName}_3 \end{array}}{\text{statEnv} \mid - \text{ElementName}_1 \text{ **substitutes for** } \text{ElementName}_3}$$

8 Type matching

Type matching is specified by the "match" judgment. It takes as input a value and a type and either succeeds if there is a match or fails if there is a mismatch. It is used in matching parameters against function signatures, type declarations, and matching values against cases in "typeswitch". The "subtyping" judgment takes two types and succeeds if all values matching the first type also match the second. It is used to define the static semantics of operations using type matching. When the given value matches the given type the following judgment must be hold.

$$\text{statEnv} \mid - \text{Value} \text{ **matches** } \text{Type}$$

An element matches an element type if the element type resolves to another element type, and the type annotation is derived from the type annotation of the resolved type. Below the derivation is given.

$$\frac{\begin{array}{l} \text{statEnv} \mid - \text{ElementName} \text{ **lookup** } \text{ElementType} \text{ **yields** } \text{Nillable?} \text{ of type } \\ \text{BaseTypeName} \\ \text{statEnv} \mid - \text{TypeName} \text{ **derives from** } \text{BaseTypeName} \text{ Value **filter** @xsi:nil =>} \\ \text{() or false} \end{array}}{\text{statEnv} \mid - \text{element } \text{ElementName} \text{ of type } \text{TypeName} \{ \text{Value} \} \text{ **matches** } \\ \text{ElementType}}$$

The derivation rule for attributes is similar. Again, the `TypeName` has to be derived from a `BaseTypeName` in order to be valid.

$$\frac{\begin{array}{l} \text{statEnv} \mid - \text{AttributeName} \text{ **lookup** } \text{AttributeType} \text{ **yields** } \text{ of type } \\ \text{BaseTypeName} \\ \text{statEnv} \mid - \text{TypeName} \text{ **derives from** } \text{BaseTypeName} \end{array}}{\text{statEnv} \mid - \text{attribute } \text{AttributeName} \text{ of type } \text{TypeName} \{ \text{Value} \} \text{ **matches** } \\ \text{AttributeType}}$$

In the first rule, a document node matches a document type if the node's content matches the document type and in the second rule A text node matches text of the same type.

$$\text{statEnv} \mid - \text{document} \{ \text{Value} \} \text{ **matches** } \text{document} \{ \text{Type} \}$$

$$\frac{}{\text{statEnv} \mid\text{- } \text{text} \{ \text{String} \} \text{ matches } \text{text}}$$

An atomic value matches an atomic type if its type annotation derives from the atomic type.

$$\text{statEnv} \mid\text{- } \text{AtomicTypeName}_1 \text{ derives from } \text{AtomicTypeName}_2$$

$$\frac{}{\text{statEnv} \mid\text{- } \text{AtomicValue of type } \text{AtomicTypeName}_1 \text{ matches } \text{AtomicTypeName}_2}$$

8.1 Subtyping Semantics of XPath

Subtyping is used during the static type analysis, in typeswitch expressions, and to check the correctness of function applications. The following judgment must hold if the first type is a subtype of the second.

$$\text{statEnv} \mid\text{- } \text{Type}_1 <: \text{Type}_2$$

The subtype relation $<:$ is *reflexive* (to itself) and *transitive*:

$$\text{statEnv} \mid\text{- } \text{Type} <: \text{Type}$$

$$\text{statEnv} \mid\text{- } \text{Type}_1 <: \text{Type}_2 \quad \text{and} \quad \text{statEnv} \mid\text{- } \text{Type}_2 <: \text{Type}_3$$

$$\frac{}{\text{statEnv} \mid\text{- } \text{Type}_1 <: \text{Type}_3}$$

The goal of formal semantics is not to give a mean to *compute subtyping*. The definitions above refer to values which are not available at static type checking type. The structural component of the XPath type system can be modeled by regular expressions. Regular expressions can be implemented by finite state automata. A way for computing subtyping between two types is to compute if *inclusion* holds between their corresponding finite states automata.

9 Complex Types

The following grammar productions are used to describe the content of a complex type definition.

```

ComplexTypeContent ::= "annotation"? ("simpleContent"
| "complexContent" | (ChildrenContent
AttributeContent))
AttributeContent  ::= ("attribute" | "attributeGroup")*
"anyAttribute"?
ChildrenContent   ::= ("group" | "all" | "choice" | "sequence")?

```

9.1 Global complex type

We can import complex types from XML Schema definitions and these types are distinguished between global complex types (which are mapped to sort declarations) and local complex types (which are mapped to type definitions). In the case of global complex types, the mapping rules to the XPath type system are the following:

```
[<complexType MixedAttribute name = QName > ComplexTypeContent
  </complexType>]definition(targetNCName)
==
define type targetNCName:QName [MixedAttribute
  ComplexTypeContent]mixed_content(targetNCName)
```

9.2 Local complex type

In the case of a local complex types, there must not be a name attribute and the mapping rule which applies from the XML Schema to type system is the following:

```
[ <complexType MixedAttribute >
  ComplexTypeContent </complexType>]content(targetNCName)
==
[MixedAttribute ComplexTypeContent]mixed_content(targetNCName)
```

9.3 Complex types with simple content

A complex type can be of simple content. A simple content is represented in XML by the following structure.

```
<simpleContent id = ID {any attributes with non-schema namespace .
. .} >
Content: (annotation?, (restriction | extension))
</simpleContent>
```

Derivation by restriction inside a simple content is represented in XML by the following structure.

```
<restriction base = QName id = ID{any attributes with non-schema
namespace . . .}>
Content: (annotation?,
(simpleType?,
(minExclusive | minInclusive | maxExclusive | maxInclusive |
totalDigits | fractionDigits | length | minLength | maxLength |
enumeration | whitespace | pattern)*)?,
((attribute | attributeGroup)*, anyAttribute?))
</restriction>
```

Derivation by extension inside a simple content is represented in XML by the following structure.

```
<extension base = QName id = ID{any attributes with non-schema
```

```

namespace . . . } >
  Content: (annotation?, ((attribute | attributeGroup)*,
anyAttribute?))
</extension>

```

The normalization rule below maps a pair of mixed attribute and complex type content to a type derivation.

$$\begin{aligned}
& [MixedAttribute ComplexTypeContent]_{mixed_content(targetNCName)} \\
& \quad == \\
& \quad TypeDerivation
\end{aligned}$$

Complex types with simple content must not have a `mixed` attribute set to "true". If the simple content is derived by restriction, it is mapped into a simple type restriction in the type system. Only the name of the base atomic type and attributes are mapped, while the actual simple type restriction is ignored.

$$\begin{aligned}
& (mixed = "false") \left[\langle simpleContent \rangle \langle restriction \ base = QName \rangle \right. \\
& \quad \left. simpleContentRestriction \ AttributeContent \langle /restriction \rangle \right. \\
& \quad \left. \langle /simpleContent \rangle \right]_{mixed_content(targetNCName)} \\
& \quad == \\
& \quad restricts \ QName \ \{ \left[AttributeContent \right]_{content(targetNCName)} \ QName \}
\end{aligned}$$

If the simple type is derived by extension, it is mapped into an extended type specifier into the type system.

$$\begin{aligned}
& (mixed = "false") \left[\langle simpleContent \rangle \langle extension \ base = QName \rangle \right. \\
& \quad \left. AttributeContent \ \langle /extension \rangle \ \langle /simpleContent \rangle \right]_{mixed_content(targetNCName)} \\
& \quad == \\
& \quad extends \ QName \ \{ \left[AttributeContent \right]_{content(targetNCName)} \}
\end{aligned}$$

10 A Coq-based approach for XPath Formal Semantics

The Semantics implementation in Coq according to [3] requires updatable definitions of common set operations (union, intersection, inclusion) over previously defined node-sets. Two basic expressions in XPath is the `p1/p2` in which we retrieve the node set of `p2` based in the returned node set of `p1` and the `p[q]` expression in which we filter the results of `p` node-set based on the `q` restriction. This can be captured in Coq by two higher order functions, `product` and `filter`.

```

Fixpoint product (s : NodeSet) (fs : Node → NodeSet) {struct s} : NodeSet :=
  match s with
  — empty ⇒ empty
  — item a s1 ⇒ union (fs a) (product s1 fs)
  end.

```

```

Fixpoint filter (s : NodeSet) (fs : Node → bool) {struct s} : NodeSet :=
  match s with
  — empty ⇒ empty
  — item a s1 ⇒ if fs a then item a (filter s1 fs) else filter s1 fs
  end.

```

Using the above functions we can model the denotational semantics as a Fixpoint that returns a node-set with the results of the above XPath expressions. Below is a way to compute the node-set with Coq methodology.

```

Fixpoint semanS (t : Tree) (p : XPath)
(x : Node) {struct p} : NodeSet :=
  match p with
  — void ⇒ empty
  — top ⇒ XTree.roots t x
  — slash p1 p2 ⇒ product (semanS t p1 x) (semanS t p2)
  — union p1 p2 ⇒ union (semanS t p1 x) (semanS t p2 x)
  — inter p1 p2 ⇒ inter (semanS t p1 x) (semanS t p2 x)
  — qualify p1 q2 ⇒ filter (semanS t p1 x) (semanQ t q2)
  — step a n ⇒ filter (f a x t) (test_node t n)
  end

with semanQ (t : Tree) (q : XQualif) (x : Node) {struct q} :
bool :=
  match q with
  — _true ⇒ true
  — _false ⇒ false
  — not q1 ⇒ if semanQ t q1 x then false else true
  — and q1 q2 ⇒ if semanQ t q1 x then semanQ t q2 x else false
  — or q1 q2 ⇒ if semanQ t q1 x then true else semanQ t q2 x
  — leq p1 p2 ⇒ incl (semanS t p1 x) (semanS t p2 x)
  end.

```

10.1 Modeling Containment Relation between two expressions

If the set of nodes returned by an expression $p1$ is included in the set of nodes returned by another relation $p2$ the containment relation can be modeled as follows:

Variable $t:Tree$.

Variable $x:Node$.

Variable $Sle : XPath \rightarrow XPath \rightarrow Prop$.

Conjecture Sle_sound : $forall (p1 p2 : XPath),$
 $Sle p1 p2 \rightarrow incl (semanS t p1 x) (semanS t p2 x)=true$.

Conjecture $Sle_complete$: $forall (p1 p2 : XPath),$
 $incl (semanS t p1 x) (semanS t p2 x)=true \rightarrow Sle p1 p2$.

10.2 Path Equivalence Relation

The general equivalence relation between two XPath expressions can be defined in Coq as:

Inductive $Sequiv: XPath \rightarrow XPath \rightarrow Prop :=$

— $seq: forall (p1 p2 : XPath), Sle p1 p2 \rightarrow Sle p2 p1 \rightarrow Sequiv p1 p2$.

The equivalence relation is important for simplifying XPath normalization and rewriting issues. Furthermore both containment and equivalence relations have implications in integrity constraints [4] and database query optimization [5]. For example if $\forall p : XPath, p | p \equiv_s p$ holds then p/p can be replaced by just p and the whole expression will be simplified. This is a basic optimization for that query. The proof of this transformation in Coq uses two set-theoretic lemmas, the idempotence of set union and reflexivity of set inclusion.

11 Conclusions

XPath Formal Semantics are an active area of research at this moment and many researchers want to give their approach on defining the language. It seems that W3C group has the most power on recommending new semantics definitions and their approach is more simple and understandable than the other approaches that we studied. In addition, they publish their proposal on the web and anybody that has any objections or other proposals can comment on the text and though make the semantics better. XPath is a powerful query language and by opening the discussion about finalizing its semantics, will become even more popular in the future.

12 References

1. W3C Working Draft 29 October 2004, *XML Path Language (XPath) 2.0*
<http://www.w3.org/TR/xpath20/>
2. W3C Working Draft 20 February 2004, *XQuery 1.0 and XPath 2.0 Formal Semantics*
<http://www.w3.org/TR/xquery-semantic/>
3. P. Genevès and J.Y. Vion-Dury, *XPath Formal Semantics and Beyond: A Coq-Based Approach*, in International Conference on Theorem Proving in Higher Order Logics (TPHOLs), Emerging Trends, September 2004
4. A. Deutch, V. Tannen, Containment and Integrity constraints for XPath fragments, In Knowledge Representation Meets Databases, 2001
5. P. Geneves, J Vion Dury, Logic based XPath optimization, First International Workshop on High Performance XML Processing, May 2004