XPath Formal Semantics and Beyond: a Coq based approach

Pierre Genevès[1] and Jean-Yves Vion-Dury[1,2]

[1] WAM Project, INRIA Rhône-Alpes
[2] Xerox Research Centre Europe

Abstract. XPath was introduced as the standard language for addressing parts of XML documents, and has been widely adopted by practioners and theoreticaly studied. We aim at building a logical framework for formal study and analysis of XPath and have to face the combinatorial complexity of formal proofs caused by XPath expressive power. We chose the Coq proof assistant and its powerful inductive constructions to rigorously investigate XPath peculiarities. We focus in this paper on a basic modeling of XPath syntax and semantics, and make two contributions. First, we propose a new formal semantics, which is an interpretation of paths as first order logic propositions that turned out to greatly simplify our formal proofs. Second, we formally prove that this new interpretation is equivalent to previously known XPath denotational semantics [20, 18], opening perspectives for more ambitious mathematical characterizations. We illustrate our Coq based model through several examples and we develop a formal proof of a simple yet significant XPath property that compare quite favorably to a former informal proof proposed in [18].

1 Introduction

XML [4] is now becoming the de facto standard for both representing structured documents and exchanging information. This success impacts major parts of the computing infrastructure such as the future world wide web, information systems, and databases. XPath [6] was introduced by the W3C [16] for specifying node selection, matching conditions, and for computing values from an XML document. XPath is part of other XML-related standards such as the transformation language XSLT [5], the modeling language XML Schema [12], the linking standard XLink [8] and the forthcoming XQuery [3] database access language, that is triggering considerable attention from big industrial players. Because of its fundamental role, we see XPath as a cornerstone of XML technologies.

Motivation. We aim at building a rigorous framework for formal study and analysis of XPath. This paper focuses on a basic modeling of XPath data model, syntax and semantics as a first step toward a more ambitious goal, which is to axiomatize and characterize the containment and equivalence relations over XPath expressions. The first problem to address is the combinatorial complexity of proofs caused by XPath structure (e.g. cases analysis, structural inductions). The second problem is to handle incremental variations (and extensions) of the language fragment we want to deal with while maintaining the established properties. These two difficulties are clearly in favor of using mechanized proofs, but require a proof assistant offering powerful data structure modeling capabilities and providing a specialized language for building complex and modular proof tactics. We chose the Coq proof assistant [7] because of *(i)* its powerful inductive constructions, *(ii)* its type system and *(iii)* its tactics language. Another important point for the authors was the availability of module abstractions (clearly in favor of large project developments) and also of a very good documentation [2] that considerably eased entering Coq's arcanes. Last but not least, Coq is currently a large and active research project offering long term perspectives as well as a good support to a growing user community. Usually, proof assistants allow enforcing and verifying known mathematical results or proving simple but important algorithms. The authors expect from this exploratory work an ambitious step toward offering a common framework to theoreticians and engineers working around XML technologies. We consider XQuery as a potential target since it comes with a very large and complex formal semantics [11] while being probably too complex to support mathematical treatments without the help of a scalable and typed proof assistant (for instance, proving a worthwhile weak type soundness for the query language, or reasoning formally about normalization and optimization).

Contribution. As a first result, we propose a new formal semantics for the XPath language, which is basically an interpretation of XPath expressions in first-order logic. One of the main advantages of this semantics is that both paths and qualifiers get an unified interpretation; thus the general complexity of proofs involving XPath interpretation is greatly reduced. The other expected benefit is to abstract over the usual computational vision and to focus on the intrinsic meaning of the language. Our second contribution is a formal proof of the equivalence of semantics that enables further construction on top of this simple logical interpretation.

Related Work. The first version of the XPath specification [6], published in 1999, describes the meanings of XPath constructs and operators in more than thirty pages of english. A formal semantics of XPath was given in 2000 by Wadler in [20]. This denotational semantics inspired works on theoretical issues around XPath: rewriting [18], query containment [19] and algorithmic complexity [15]. However, this semantics conveys a computational vision and has often been directly translated into poorly efficient functional algorithms [15]. Several authors adopted simpler semantics, focusing on boolean tests or tree patterns [13] thus missing the most innovative and core XPath feature: node-set selection. Recent work on the forthcoming XPath 2.0 language formally defined static and operational semantics [10]. While being able to deal with complex typing issues raised by substantial evolution of the language specification, these semantics are probably too complex for being directly used in useful manual proofs.

Works on XPath containment and equivalence problems identified and conjectured complexity classes for several XPath fragments (see [17] for an overview). However, most of these works rely on manual proofs-by-reduction that do not help for finding sound and complete algorithms on a significant XPath subset. On the opposite, we aim at building a logical and formal framework for studying XPath, and especially for investigating XPath containment in a constructive way.

Outline We first introduce XPath and its data model in section 2. Section 3 presents the basics of XPath semantics: query results, axes and node tests. A denotational semantics of paths inspired from established contributions is then described in section 4, which also highlights its drawbacks for formal proofs. Section 5 introduces our new logical semantics and illustrates the interest of its Coq modeling through the demonstration of an XPath property. Before concluding, section 6 summarizes the formal proof of the equivalence of both semantics, constructed using the Coq proof system.

2 XPath Syntax and Data Model

A tree document model. XPath considers an XML document as a tree with several kinds of nodes (root, element, text, attribute, namespace, processing instruction, and comment). The tree is built by a successful parsing of a well-formed XML document. The tree contains only one root node, which has no parent, no attribute and no namespace node, but that may have any other kind of nodes as children. Only elements can have children. Nodes are fully connected using the relation $\multimap$ that maps a node to its children, and the reflexive and transitive closure $\multimap^*$ of this relation. Moreover, a total ordering relation $\ll$ between any two elements reflects the depth-first traversal order of the tree. We implemented this document model in Coq as two separate modules "XNodes" and "XTree" that respectively define the types "Node" and "Tree" which we refer to in this paper.

XPath expressions. In their simplest form XPath expressions look like "directory navigation paths". For example, the XPath expression

$$/\text{book}/\text{chapter}/\text{section}$$

navigates from the root of a document (designated by the leading slash "/") through the top-level "book" element to its "chapter" child elements and on to its "section" child elements. The result of the evaluation of the entire expression is the set of all the "section" elements that can be reached in this manner, returned in the order they occurred in the document. At each step in the navigation the selected nodes for that step can be filtered using qualifiers. A qualifier is a boolean expression between brackets that can test path existence. So if we ask for

$$/\text{book}/\text{chapter}/\text{section}[\text{citation}]$$

then the result is *all* "section" elements that have at least one child element named "citation". The situation becomes more interesting when combined with XPath's capability of searching along "axes" other than the shown "children of" axis. Indeed the above XPath is a shorthand for

$$/\text{child::book}/\text{child::chapter}/\text{child::section}[\text{child::citation}]$$

where it is made explicit that each *path step* is meant to search the "child" axis containing all children of the previous context node. If we instead asked for

$$/\text{child::book}/\text{descendant::*}[\text{child::citation}]$$

then the last step selects nodes of any kind that are among the descendants of the top element "book" and have a "citation" child element. Previous examples are all *absolute* XPath expressions (since they involve a leading "/"). The general meaning of an expression is defined relatively to a context node in the tree. Starting from a particular context node in the tree, every other nodes can be reached. This is because XPath defines powerful navigational capabilities, including a full set of axes, as captured on figure 1. For more informal details on the complete XPath language, the reader can refer to the specification [6].

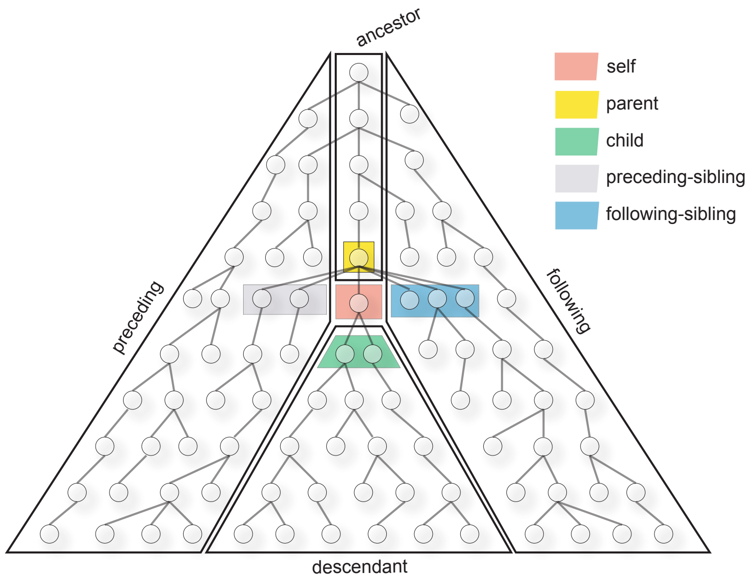

Fig. 1. Axes: partitions of document nodes from a particular context node.

Abstract syntax: a compositional fragment. For the remaining part of the paper, we focus on a restricted but significant fragment of XPath, composed of all XPath axes. The abstract syntax of the fragment is given on figure 2. In order to make the XPath syntax fully compositional, two variants are included: the void path $\perp$ and the explicit root node $\wedge$ (respectively proposed in [18] and [19]). An other extension concerning qualifiers is the inclusion constraint $p_1 \sqsubseteq p_2$ over set of nodes selected by p_1 and p_2. First defined in [19], the authors believe that this feature brings useful expressive power without increasing cost of formal treatment (however this will be verified along our ongoing work on path containment). Note that it turns the construct $p_1[p_2]$ into a syntactic sugar for $p_1[\textbf{not}\ (p_2 \sqsubseteq \perp)]$. Although the XPath fragment we consider already covers a significant range of real world use cases, our intent is to extend it to cover the XPath standard as much as possible.

$$\begin{array}{lll}
\textit{Path} & p & ::= p/p \mid p[q] \mid p \shortmid p \mid p \cap p \mid (p) \mid a{::}N \mid \bot \mid \wedge \\[1em]
\textit{Qualifier} & q & ::= q \;\textbf{and}\; q \mid q \;\textbf{or}\; q \mid \textbf{not}\; q \mid p \mid p \sqsubseteq p \mid \textbf{true} \mid \textbf{false} \\[1em]
\textit{Axis} & a & ::= \text{child} \mid \text{descendant} \mid \text{self} \mid \text{descendant-or-self} \\
 & & \quad \mid \text{following-sibling} \mid \text{following} \mid \text{parent} \mid \text{ancestor} \\
 & & \quad \mid \text{preceding-sibling} \mid \text{preceding} \mid \text{ancestor-or-self} \\[1em]
\textit{NodeTest} & N & ::= n \mid * \mid \textbf{text}() \mid \textbf{comment}() \mid \textbf{element}() \\
 & & \quad \mid \textbf{processing-instruction}() \mid \textbf{node}()
\end{array}$$

Fig. 2. XPath Abstract Syntax.

Our syntactic modeling in Coq is directly inspired from the abstract syntax. A cross-inductive set definition (see figure 3) models XPath expressions: $\bot$, $\wedge$, $a{::}N$ are path atoms and **true**, **false** are qualifier atoms, whereas other operators are binary constructors. The definition relies on the definitions of "Axis" and "NodeTest" which are simple set enumerations.

Inductive *XPath* : *Set* :=
 — *void* : *XPath*
 — *top* : *XPath*
 — *union* : *XPath* → *XPath* → *XPath*
 — *inter* : *XPath* → *XPath* → *XPath*
 — *slash* : *XPath* → *XPath* → *XPath*
 — *qualif* : *XPath* → *XQualif* → *XPath*
 — *step* : *Axis* → *NodeTest* → *XPath*
with XQualif : *Set* :=
 — *not* : *XQualif* → *XQualif*
 — *and* : *XQualif* → *XQualif* → *XQualif*
 — *or* : *XQualif* → *XQualif* → *XQualif*
 — *leq* : *XPath* → *XPath* → *XQualif*
 — *_true* : *XQualif*
 — *_false* : *XQualif*.

Fig. 3. Set of all XPath expressions in Coq.

Paths inside qualifiers (as p_2 in $p_1[p_2]$) are modeled through a syntaxic sugar:

Definition *path* (*p* : *XPath*) : *XQualif* := *not* (*leq p void*).

At this stage, XPath expressions can be instanciated using functional notation, for example:

slash root (qualif (step child book) (path (step child chapter))) or even with the familiar infix notation:

$$\wedge/\text{book[chapter]}$$

made possible by Coq's notation mechanism and definitions of operators associativity. Although some syntactic properties can already be worked out, involving results of XPath expressions requires further modeling. We formalize and model the interpretation of XPath expressions in the next sections.

3 XPath Semantics: Basics

Result of an expression. The evaluation of an XPath expression returns a node-set: an unordered collection of nodes without duplicates. We chose to model a node-set in Coq as a custom list type (shown on figure 4) rather than a set. This is in order to cope with the "position()" feature in qualifiers [6] and sequences of the forthcoming XPath 2.0 language [1]. Indeed, the "position()" feature requires an ordered representation of selected nodes for filtering purposes. Moreover, XPath 2.0 handles node sequences (ordered collections of zero or more items, with possible duplicates) instead of node-sets. Thus, our Coq modeling of node-sets presently uses a list together with an associated predicate for forcing uniqueness of nodes in the node-set.

```
Inductive NodeSet : Set :=
  — empty : NodeSet
  — item : Node → NodeSet → NodeSet.
```

Fig. 4. Coq modeling of node-sets.

Axes and node tests. The path step (a::N) is the most basic XPath construct that allows to navigate in the tree in order to retrieve a node-set. Its semantics relies on two functions f and $\mathcal{T}$ that respectively define the semantics of an axis a and a node test N. The navigational semantics of axes can be pictured using the tree document model (see figure 1); and more formally defined using the *parent/child* relation (as usual $\rightarrowtriangle^+$ means $\rightarrowtriangle\rightarrowtriangle^*$), and the irreflexive ordering relation $\ll$. The function f retrieves a node-set starting from a context node x:

a	$f(a)_x$
self	$\{x\}$
child	$\{y \| x \rightarrowtriangle y\}$
parent	$\{y \| y \rightarrowtriangle x\}$
descendant	$\{y \| x \rightarrowtriangle^+ y\}$
ancestor	$\{y \| y \rightarrowtriangle^+ x\}$
descendant-or-self	$\{y \| x \rightarrowtriangle^* y\}$
ancestor-or-self	$\{y \| y \rightarrowtriangle^* x\}$
following-sibling	$\{y \| y \in$ sibling$(x) \wedge x \ll y\}$
preceding-sibling	$\{y \| y \in$ sibling$(x) \wedge y \ll x\}$
preceding	$\{y \| y \ll x\}$
following	$\{y \| x \ll y\}$
attribute	$\{y \| x \rightarrowtriangle y \wedge$ is-attribute$(y)\}$
namespace	$\{y \| x \rightarrowtriangle y \wedge$ is-namespace$(y)\}$
with sibling(x)=	$\{y \| \exists z \;\; z \rightarrowtriangle x \;\wedge\; z \rightarrowtriangle y\}$

The node test part of a step is useful to filter the nodes according to their kind. The function $\mathcal{T}$ performs the test by attempting to match a node x with the node test N used in the step, according to the table below. The matching depends on the axis used in the step:

N	a	$\mathcal{T}(a, N, x)$
n		name(x)=n
$*$	attribute	is-attribute(x)
$*$	namespace	is-namespace(x)
$*$	other	is-element(x)
text()		is-text(x)
comment()		is-comment(x)
processing-instruction()		is-pi(x)
element()		is-element(x)
node()		true

The functions f and $\mathcal{T}$ are directly translated into Coq definitions that drive our "XTree" document model. The composition of f and $\mathcal{T}$ allows to define the interpretation of a path step, which is an essential aspect of path semantics.

4 Denotational Semantics of Paths and Qualifiers

A classic formal semantics of paths finds its origins in [20], [18] and [19]. A formal semantics function $\mathcal{S}$ computes the node-set selected by a path p starting from a context node x in the tree:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{S} : \textit{Path} \longrightarrow \textit{Node} \longrightarrow \mathrm{Set}(\textit{Node})} \\
\mathcal{S}[\![\wedge]\!]_x & = & \{x_1 \mid x_1 \rightarrowtriangle^* x \wedge \mathrm{root}(x_1)\} \\
\mathcal{S}[\![\bot]\!]_x & = & \emptyset \\
\mathcal{S}[\![p_1 \mid p_2]\!]_x & = & \mathcal{S}[\![p_1]\!]_x \cup \mathcal{S}[\![p_2]\!]_x \\
\mathcal{S}[\![p_1 \cap p_2]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}[\![p_1]\!]_x \wedge x_1 \in \mathcal{S}[\![p_2]\!]_x\} \\
\mathcal{S}[\![p_1 / p_2]\!]_x & = & \{x_2 \mid x_1 \in \mathcal{S}[\![p_1]\!]_x \wedge x_2 \in \mathcal{S}[\![p_2]\!]_{x_1}\} \\
\mathcal{S}[\![(p)]\!]_x & = & \mathcal{S}[\![p]\!]_x \\
\mathcal{S}[\![p[q]]\!]_x & = & \{x_1 \mid x_1 \in \mathcal{S}[\![p]\!]_x \wedge \mathcal{Q}[\![q]\!]_{x_1}\} \\
\mathcal{S}[\![a{::}N]\!]_x & = & \{x_1 \mid x_1 \in f(a)_x \wedge \mathcal{T}(a, N, x_1)\}
\end{array}
$$

The interpretation of a qualified path $p[q]$ uses the dual formal semantics function $\mathcal{Q}$ for qualifiers. $\mathcal{Q}$ returns the boolean evaluation of a qualifier q from a context node x:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{Q} : \textit{Qualifier} \longrightarrow \textit{Node} \longrightarrow \textit{Boolean}} \\
\mathcal{Q}[\![\mathbf{true}]\!]_x & = & \mathrm{true} \\
\mathcal{Q}[\![\mathbf{false}]\!]_x & = & \mathrm{false} \\
\mathcal{Q}[\![q_1 \text{ }\mathbf{and}\text{ } q_2]\!]_x & = & \mathcal{Q}[\![q_1]\!]_x \wedge \mathcal{Q}[\![q_2]\!]_x \\
\mathcal{Q}[\![q_1 \text{ }\mathbf{or}\text{ } q_2]\!]_x & = & \mathcal{Q}[\![q_1]\!]_x \vee \mathcal{Q}[\![q_2]\!]_x \\
\mathcal{Q}[\![p]\!]_x & = & \mathcal{Q}[\![\mathbf{not}\ (p \sqsubseteq \bot)]\!]_x \\
\mathcal{Q}[\![(q)]\!]_x & = & \mathcal{Q}[\![q]\!]_x \\
\mathcal{Q}[\![\mathbf{not}\ q]\!]_x & = & \neg \mathcal{Q}[\![q]\!]_x \\
\mathcal{Q}[\![p_1 \sqsubseteq p_2]\!]_x & = & \mathcal{S}[\![p_1]\!]_x \subseteq \mathcal{S}[\![p_2]\!]_x
\end{array}
$$

The implementation of $\mathcal{S}$ in Coq requires updatable definitions of common set operations (union, intersection, inclusion) over previously defined node-sets. More interesting are the two XPath-specific constructs p_1/p_2 and $p[q]$ that require an ordered evaluation of subterms. Indeed, the node-set retrieval driven by p_2 and the filter performed by q respectively operate on the results of p_1 and p. This can be captured in Coq via two higher order functions. These functions abstract over the context node used for the evaluation of p_2 and q:

```
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.
```

The denotational semantics can then be modeled as a fixpoint that returns the node-set selected by a path p from a context node x in a tree t as shown on figure 5.

```
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)
  — qualif 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.
```

Fig. 5. XPath Denotational Semantics in Coq.

At this stage, XPath interpretation can be used for studying properties involving query results. Consider for example the containment relation, which holds between two XPath expressions p_1 and p_2 when the set of nodes returned by p_1 is included in the set of nodes returned by p_2, for all trees and context nodes. The containment relation can be formally modeled as follows:

```
Variable t:Tree.
Variable x:Node.

 Variable Sle : XPath → XPath → Prop.

 Conjecture Sle_sound: forall (p1 p2 : XPath),
  Sle p1 p2 → 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 → Sle p1 p2.
```

The general path equivalence relation $\equiv_{\mathcal{S}}$, that holds between two paths that always have the same interpretation, can then be defined:

```
Inductive Sequiv: XPath → XPath → Prop :=
```

— *seq*: *forall* (*p1 p2* : *XPath*), *Sle p1 p2* → *Sle p2 p1* → *Sequiv p1 p2*.

Identifying path equivalence classes is of very first importance for simplifying general formal treatment of XPath. The equivalence relation is particularly crucial for XPath normalization and rewriting issues (see [18] for an application motivated by streaming XML querying). In addition, both equivalence and containment relations are currently of great interest for XML researchers notably because of their implications for integrity constraints checking [9] and database query optimization [14]. Consider the following basic example: if $\forall p$: *XPath*$, p|p \equiv_{\mathcal{S}} p$ holds then $p|p$ can securely be replaced by p for optimization purposes while preserving query semantics. Using the Coq modeling, the proof of $p|p \equiv_{\mathcal{S}} p$ relies on two set-theoretic lemma (idempotence of set union and reflexivity of set inclusion):

Lemma *opt* : *forall* (*p* : *XPath*), *Sequiv* (*union p p*) *p*.
Proof.
intro;*constructor*; *apply Sle_complete*; *simpl*;*rewrite union_idem*;*apply incl_reflexive*.
Qed.

Now consider a more general XPath property, often named "qualifier flattening", that was first given in [18]. This property basically states that nested qualifiers can be seen as paths:

$$\forall p, p_1, p_2 : \textit{Path} \quad p[p_1[p_2]] \equiv_{\mathcal{S}} p[p_1/p_2] \tag{1}$$

This property can be formulated as follows:

Lemma *flatten_qualifs*: *forall* (*p p1 p2*:*XPath*),
Sequiv (*qualif p* (*path* (*qualif p1* (*path p2*)))) (*qualif p* (*path* (*slash p1 p2*))).

The Coq modeling of the denotational semantics allows to prove this property. However, using the denotational semantics in proofs means dealing with combined node-set computation and boolean evaluation. Indeed, the denotational semantics relies on node-set construction for evaluating paths and boolean evaluation for interpreting qualifiers. Subsequently, ad-hoc auxiliary lemma are required for characterizing these two different computational visions, together with their compositional peculiarities. As a consequence, a major drawback is that intrinsic complexity of proofs becomes hidden behind numerous operational considerations. This causes rather long and complex proof terms. Consider for example the proof of (1); it could begin with the following tactic applications:

intros; *constructor*; *apply Sle_complete*.
 simpl.

This generates two subgoals that require to deal with mixed node-set construction and boolean evaluation (see appendix A). In the next section, we present a new simple XPath semantics designed to eliminate this computational overload.

5 A Relational Semantics in First-Order Logic

We propose to translate an XPath expression p into a dyadic formula of the first order logic (*FOL*). The semantics function $\mathcal{R}_p$ defines the interpretation of paths in the first order logic. $R_p(x,y)$ holds for all pairs x, y of nodes such that y is accessed from x through the path p:

$$\mathcal{R}_p : \textit{Path} \longrightarrow \textit{Node} \longrightarrow \textit{Node} \longrightarrow \textit{FOL}$$

$$\begin{array}{lcl}
\mathcal{R}_p[\![\wedge]\!]_x^y & = & y \multimap^* x \wedge \mathrm{root}(y) \\
\mathcal{R}_p[\![\bot]\!]_x^y & = & \mathrm{false} \\
\mathcal{R}_p[\![p_1 \mid p_2]\!]_x^y & = & \mathcal{R}_p[\![p_1]\!]_x^y \vee \mathcal{R}_p[\![p_2]\!]_x^y \\
\mathcal{R}_p[\![p_1 \cap p_2]\!]_x^y & = & \mathcal{R}_p[\![p_1]\!]_x^y \wedge \mathcal{R}_p[\![p_2]\!]_x^y \\
\mathcal{R}_p[\![p_1/p_2]\!]_x^y & = & \exists z \quad \mathcal{R}_p[\![p_1]\!]_x^z \wedge \mathcal{R}_p[\![p_2]\!]_z^y \\
\mathcal{R}_p[\![(p)]\!]_x^y & = & \mathcal{R}_p[\![p]\!]_x^y \\
\mathcal{R}_p[\![p[q]]\!]_x^y & = & \mathcal{R}_p[\![p]\!]_x^y \wedge \mathcal{R}_q[\![q]\!]_y \\
\mathcal{R}_p[\![a{::}N]\!]_x^y & = & y \in f(a)_x \wedge \mathcal{T}(a,N,y)
\end{array}$$

The dual formal semantics function R_q translates qualifiers into monadic formulæ. $R_q(x)$ holds for all nodes x such that the qualifier q is true from the context node x:

$$\mathcal{R}_q : \textit{Qualifier} \longrightarrow \textit{Node} \longrightarrow \textit{FOL}$$

$$\begin{array}{lcl}
\mathcal{R}_q[\![\textbf{true}]\!]_x & = & \mathrm{true} \\
\mathcal{R}_q[\![\textbf{false}]\!]_x & = & \mathrm{false} \\
\mathcal{R}_q[\![q_1 \textbf{ and } q_2]\!]_x & = & \mathcal{R}_q[\![q_1]\!]_x \wedge \mathcal{R}_q[\![q_2]\!]_x \\
\mathcal{R}_q[\![q_1 \textbf{ or } q_2]\!]_x & = & \mathcal{R}_q[\![q_1]\!]_x \vee \mathcal{R}_q[\![q_2]\!]_x \\
\mathcal{R}_q[\![p]\!]_x & = & \mathcal{R}_q[\![\textbf{not } (p \sqsubseteq \bot)]\!]_x \\
\mathcal{R}_q[\![(q)]\!]_x & = & \mathcal{R}_q[\![q]\!]_x \\
\mathcal{R}_q[\![\textbf{not } q]\!]_x & = & \neg\mathcal{R}_q[\![q]\!]_x \\
\mathcal{R}_q[\![p_1 \sqsubseteq p_2]\!]_x & = & \forall z \quad \mathcal{R}_p[\![p_1]\!]_x^z \Rightarrow \mathcal{R}_p[\![p_2]\!]_x^z
\end{array}$$

This semantics abstracts over the usual computation of node-sets. It gives an unified interpretation of paths and qualifiers. This enables further studying and manipulatation of XPath with an exclusive logical vision. The Coq implementation of this semantics, shown on figure 6, basically translates an XPath expression into a logical proposition. Capturing XPath semantics using Coq's basic "Prop" sort greatly reduces the complexity of proof terms. Indeed, dealing with set-handling peculiarities (such as "product" or "filter") is no more required. Proofs involving query results can be accomplished by using built-in Coq's tactics. For example, let us model the containment relation (as "Rle") and the path equivalence relations $\equiv_{\mathcal{R}}$ (as "Requiv") on top of this new logical interpretation:

Variable *Rle* : *XPath* → *XPath* → *Prop*.

Conjecture Rle_sound: *forall* (*p1 p2* : *XPath*),
Rle p1 p2 → (*forall y*:*Node*, *Rp t p1 x y* → *Rp t p2 x y*).

```
Fixpoint Rp (t : Tree) (p : XPath) (x y : Node) {struct p} : Prop :=
  match p with
  — void ⇒ False
  — top ⇒ s_in y (XTree.roots t x)=true
  — union p1 p2 ⇒ Rp t p1 x y ∨ Rp t p2 x y
  — inter p1 p2 ⇒ Rp t p1 x y ∧ Rp t p2 x y
  — slash p1 p2 ⇒ exists z : Node, Rp t p1 x z ∧ Rp t p2 z y
  — qualif p q ⇒ Rp t p x y ∧ Rq t q y
  — step a n ⇒ (s_in y (f a x t))=true ∧ (test_node t n y)=true
  end

 with Rq (t : Tree) (q : XQualif) (x : Node) {struct q} : Prop :=
  match q with
  — _true ⇒ True
  — _false ⇒ False
  — not q ⇒ ¬ Rq t q x
  — and q1 q2 ⇒ Rq t q1 x ∧ Rq t q2 x
  — or q1 q2 ⇒ Rq t q1 x ∨ Rq t q2 x
  — leq p1 p2 ⇒ forall z : Node, Rp t p1 x z → Rp t p2 x z
  end.
```

Fig. 6. XPath Logical Semantics in Coq.

```
 Conjecture Rle_complete: forall (p1 p2 : XPath),
  (forall y:Node, Rp t p1 x y→ Rp t p2 x y) → Rle p1 p2.

 Inductive Requiv: XPath → XPath → Prop :=
  — req: forall (p1 p2 : XPath),    Rle p1 p2 → Rle p2 p1 → Requiv p1 p2.
```

The "flattening qualifiers" property can now be expressed as follows:

$$\forall p, p_1, p_2 : \mathit{Path} \quad p[p_1[p_2]] \equiv_{\mathcal{R}} p[p_1/p_2] \tag{2}$$

As opposed to the lemma (1), the lemma (2) based on $\equiv_{\mathcal{R}}$ can be proved with a few applications of Coq's built-in tactics only:

```
Lemma flatten_qualifs2: forall (p p1 p2:XPath),
Requiv (qualif p (path (qualif p1 (path p2)))) (qualif p (path (slash p1 p2))).
Proof.
intros; constructor; apply Rle_complete; simpl; intros y H; elim H;
intro H0; split; try assumption; intro H2;apply H1; intros z H3; elim H3;
intros H4 H5; elim H5; intros H6 H7; [ elim (H2 H6); exists z — elim (H2 H4)];
split; try assumption; intro H8;apply (H8 z);assumption.
Qed.
```

The reader will notice that the proof of (2) is even comparable in size with the *manual* proof of (1), found in [18], that expands the denotational semantics:

$$\begin{aligned}\mathcal{S}[\![p[p_1[p_2]]]\!]_x &= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge \mathcal{Q}[\![p_1[p_2]]\!]_{x_1}\}\\ &= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge (\mathcal{S}[\![p_1[p_2]]\!]_{x_1} \neq \emptyset)\}\\ &= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge (\{x_2 | x_2 \in \mathcal{S}[\![p_1]\!]_{x_1} \wedge (\mathcal{S}[\![p_2]\!]_{x_2} \neq \emptyset)\} \neq \emptyset)\}\\ &= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge (\{x_2 | x_2 \in \mathcal{S}[\![p_1]\!]_{x_1} \wedge x_3 \in \mathcal{S}[\![p_2]\!]_{x_2}\} \neq \emptyset)\}\\ &= \{x_1 | x_1 \in \mathcal{S}[\![p]\!]_x \wedge (\mathcal{S}[\![p_1/p_2]\!]_{x_1} \neq \emptyset)\}\\ &= \mathcal{S}[\![p[p_1/p_2]]\!]_x.\end{aligned}$$

To summarize, the Coq proof system and our modeling of XPath offer the major advantages we are interested in:

– rigour of a mechanized inference system in a precisely defined logic framework;
– ability to tackle combinatorial issues by using tactic composition;
– ability to achieve "incremental proving" thanks to proof replaying and updating facilities.

Incremental proving is convenient since it allows to handle the XPath language progressively and to update the semantics accordingly. Last but not least, all these advantages come at a low cost when using our logical semantics, which greatly simplifies proof development.

6 Equivalence of Denotational and Logical Semantics

To ensure that the formal semantics function R_p really captures XPath semantics, we built a formal proof with Coq that shows that denotational and logical semantics are equivalent:

Proposition 1. *Equivalence of semantics.* $\forall p$: *Path*, $\forall x, y$: *Node*, $y \in \mathcal{S}[\![p]\!]_x \;\Leftrightarrow\; \mathcal{R}_p[\![p]\!]_x^y$

The proof uses the modelings presented in sections 4 and 5. Proposition 1 is formulated as follows:

Theorem *sem_equivalence*:
forall (*p* : *XPath*) (*x y* : *Node*) , *s_in y* (*semanS t p x*)=*true* ↔ *Rp t p x y*.

Where "s_in" simply tests the membership of a node in a given node-set. Since paths are inductively defined, the proof naturally uses an induction on p. However, because the definition of paths is cross-inductive with the definition of qualifiers (see figure 3), a mutual induction scheme is used. It is required to prove property 1 for the inductive case $p[q]$, otherwise not possible without assuming the dual property for qualifiers. The appropriate mutual induction scheme (*XJ1*) can be automatically built by Coq from the definition of paths:

Scheme *XJ1* := *Induction for XPath Sort Prop*
  with XJ2 := *Induction for XQualif Sort Prop*.

The dual property for qualifiers is defined:

```
Definition sem_equivalence_for_qualifs (q : XQualif) : Prop :=
 forall x : Node, (semanQ env t q x)=true ↔ Rq t q x.
```

The proof of proposition 1, whose skeleton is shown on figure 7, can then begin by applying the mutual induction scheme on p. We attempted to build the proof in a modular way, so that when XPath constructs are changed or added, proof parts of unchanged constructs remain valid. To this end, several tactics named "Solve_X" are defined with the intent to deal with a particular subgoal of the proof. The main proof body (see figure 7) consists in composing these tactics. Each tactic is applied in a way that either completely solve a subgoal or does not modify it at all. This allows to control which parts of the proof require an update when the underlying definitions evolve. Each tactic first attempts to match the goal it is intended to solve and the corresponding

```
Theorem sem_equivalence:
forall (p : XPath) (x y : Node) , s_in y (semanS t p x)=true ↔ Rp t p x y.
Proof.
intro p.
pattern p in ⊢ ×.
apply XJ1 with sem_equivalence_for_qualifs; intros; split;intros;
  try solve_void1;try solve_void2;
  try solve_top1; try solve_top2;
  try solve_union1; try solve_union2;
  try solve_inter1; try solve_inter2;
  try solve_product1; try solve_product2;
  try solve_qualif1; try solve_qualif2;
  try solve_step1; try solve_step2;
  try solve_not1; try solve_not2;
  try solve_and1; try solve_and2;
  try solve_or1; try solve_or2;
  try solve_leq1; try solve_leq2;
  try solve [simpl;auto];
  try solve [simpl;reflexivity];
  try solve [simpl in H;auto;discriminate];
  try solve [simpl in H;auto].
Qed.
```

Fig. 7. Main body of the modular proof of semantics equivalence.

hypotheses. For example, the tactic named "Solve_product1" (see figure 8) isolates the proof of the first inductive case for the "product" construct, whereas the tactic named "Solve_product2" contains the proof of the reciprocal property. In each tactic, the variable names used for matching purposes (e.g. strings after the "?") in the proof context directly correspond to the names that Coq would generate if the proof is manually achieved step by step. Preserving compatibility of names is convenient for updating

```
Ltac solve_product1:=
  match goal with
  — H1: s_in ?y (semanS ?t (slash ?x ?x0) ?x1) = true,
    H: (forall (gx0 gy : Node)(gt : Tree),
        ((s_in gy (semanS gt ?x gx0) = true) ↔ Rp gt ?x gx0 gy)),
    H0:(forall (hx hy : Node)(ht : Tree),
        ((s_in hy (semanS ht ?x0 hx) = true) ↔ Rp ht ?x0 hx hy))
    ⊢ Rp ?t (slash ?x ?x0) ?x1 ?y
  ⇒ simpl in ⊢ ×; simpl in H1;
assert (H2 := in_product1 y (semanS t x x1) (semanS t x0) H1);
elim H2;intros x2 H3; elim H3; intros H3A H3B;exists x2;
elim (H x1 x2 t); intros HE1 HE2;
elim (H0 x2 y t); intros HF1 HF2;
split; [ apply HE1;assumption —apply HF1;assumption]
end.
```

Fig. 8. A tactic for solving a specific subgoal.

proofs, as the proof script can simply be copied and pasted to and from the proof engine. Tactics can use auxiliary lemma that characterize peculiarities of the denotational

```
Lemma in_product1: forall (y : Node)(s : NodeSet)(f:Node→NodeSet),
s_in y (product s f) = true → exists z : Node, s_in z s=true ∧ s_in y (f z) = true.
Proof.
induction s;
[ intros; rewrite product_empty in H; rewrite in_sem1 in H; discriminate
— intros;simpl;cut ({s_in y (product s f) = true} + {s_in y (product s f) = false});
  [ intros HC; elim HC; intros HC1;
    [ elim (IHs f); intros;
      [ exists x; elim H0; intros; split;
        [ apply in_sem5; assumption — assumption] — assumption]
    — exists a; split; [ apply in_sem2
                           — eapply in_Lunion;[ apply H; assumption — assumption]]]
  —apply in_dec]].
Qed.
```

Fig. 9. Lemma for characterizing a peculiarity of the denotational semantics.

semantics. For example, the lemma “in_product1”, shown on figure 9 is used by the tactic “Solve_product1” (figure 8). “in_product1” basically states that when the result of a path construct p_1/p_2 is not empty then at least one result node of p_1 is used for evaluating p_2. This is proved using several trivial lemmas on node-sets pictured on figure 10. Proposition 1 allows to securely take advantage of the logical semantics.

Lemma *product_empty* : *forall f* : *Node* $\rightarrow$ *NodeSet*, *product empty f* = *empty*.
Lemma *in_sem1* : *forall a* : *Node*, *s_in a empty* = *false*.
Lemma *in_sem2* : *forall* (*a* : *Node*) (*s* : *NodeSet*), *s_in a* (*item a s*) = *true*.
Lemma *in_sem5* : *forall* (*a b* : *Node*) (*s* : *NodeSet*), *s_in a s* = *true* $\rightarrow$ *s_in a* (*item b s*) = *true*.
Lemma *in_Lunion* : *forall* (*a* : *Node*) (*s1 s2* : *NodeSet*),
s_in a (*union s1 s2*) = *true* $\rightarrow$ *s_in a s2* = *false* $\rightarrow$ *s_in a s1* = *true*.
Lemma *in_dec* : *forall* (*s* : *NodeSet*) (*a* : *Node*), {*s_in a s* = *true*} + {*s_in a s* = *false*}.

Fig. 10. Trivial lemma on node-sets used by proof of "in_product1".

7 Conclusion

In this paper, we focused on a basic modeling of XPath syntax and formal semantics for using the Coq proof system. We introduced a new formal semantics for XPath, that has two main advantages: first, it unifies path and qualifier interpretations. Second, it allows to focus on the intrinsic meaning of XPath from a pure logic point of view. These advantages allow significant simplifications in formal proofs. In addition, we formally proved that this new interpretation is equivalent to the previously known XPath semantics.

Lessons learned. Modeling XPath within the Coq proof system has shown to be a good choice for building a scalable logical framework around XPath. Indeed, Coq's tactic composition features are a realistic way to cope with combinatorial issues raised by XPath expressions. Moreover, Coq provides facilities for incrementally updating proofs when our XPath fragment evolves.

Future Directions We plan to take part of this framework for studying longer and more complex proofs around XPath open questions. Especially, our intent is to axiomatize the containment relation over XPath expressions; and then to demonstrate the soundness and possibly the completeness of the relation. This characterization will strongly rely on the Coq modeling of our logical semantics. After defining the relation, we plan to demonstrate the properties "Rle_sound" and "Rle_complete" presented as conjectures in section 5. The next step is to progressively extend the XPath fragment to support significant real world applications.

References

1. A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) 2.0, W3C working draft, August 2003. http://www.w3.org/TR/2003/WD-xpath20-20030822.
2. Y. Bertot and P. Castéran. *Coq'Art*, chapter To appear. Springer-Verlag, 2004.
3. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language, W3C working draft, November 2003. http://www.w3.org/TR/xquery/.
4. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (third edition), W3C recommendation, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204/.

5. J. Clark. XSL transformations (XSLT) version 1.0, W3C recommendation, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.
6. J. Clark and S. DeRose. XML path language (XPath) version 1.0, W3C recommendation, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.
7. The coq proof assistant, 2003. http://coq.inria.fr.
8. S. DeRose, E. Maler, and D. Orchard. XML Linking Language (XLink) version 1.0, W3C Recommendation, June 2001. http://www.w3.org/TR/xlink/.
9. A. Deutsch and V. Tannen. Containment and integrity constraints for xpath fragments. In *Knowledge Representation Meets Databases*, 2001.
10. D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C working draft, February 2004. http://www.w3.org/TR/xquery-semantics/.
11. D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. Xquery 1.0 and xpath 2.0 formal semantics, February 2004. http://www.w3.org/TR/xquery-semantics/.
12. D. C. Fallside. XML Schema part 0: Primer, W3C recommendation, May 2001. http://www.w3.org/TR/xmlschema-0/.
13. S. Flesca, F. Furfaro, and E. Masciari. Minimization of tree patterns queries. In *Proceedings of the 29th VLDB Conf.*, pages 497–508, January 2000.
14. P. Genevès and J.-Y. Vion-Dury. Logic-based XPath optimization. In *First International Workshop on High Performance XML Processing*, May 2004.
15. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. 28th Int. Conf. on Very Large Data Bases (VLDB 2002)*, pages 95–106, Hong Kong, China, 2002. Morgan Kaufmann.
16. MIT, ERCIM, and Keio. The World Wide Web Consortium (W3C), 1994. http://www.w3.org.
17. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proceedings of the 9th International Conference on Database Theory*, pages 315–329. Springer-Verlag, 2002.
18. D. Olteanu, H. Meuss, T. Furche, and F. Bry. Symmetry in XPath. In *Proceedings of Seminar on Rule Markup Techniques, no. 02061, Schloss Dagstuhl, Germany (7th February 2002)*, 2001.
19. J.-Y. Vion-Dury and N. Layaïda. Containment of XPath expressions: an inference and rewriting based approach. In *Extreme Markup Languages*, August 2003.
20. P. Wadler. Two semantics for XPath. http://www.research.avayalabs.com/user/wadler/papers/xpath-semantics/xpath-semantics.pdf, January 2000.

A Denotational interpretations of paths and qualifiers mixed in a proof.

```
2 subgoals

  p : XPath
  p1 : XPath
  p2 : XPath
  ============================
   incl
     (filter (semanS t p x)
        (fun x0 : Node =>
         if incl
              (filter (semanS t p1 x0)
                 (fun x1 : Node =>
                  if incl (semanS t p2 x1) empty
                  then false
                  else true)) empty
         then false
         else true))
     (filter (semanS t p x)
        (fun x0 : Node =>
         if incl (product (semanS env t p1 x0) (semanS env t p2))
              empty
         then false
         else true)) = true

subgoal 2 is:
 Sle (qualif p (path (slash p1 p2))) (qualif p (path (qualif p1 (path p2))))
```