XPath-Logic and XPathLog: A Logic-Based Approach for Declarative XML Data Manipulation

Wolfgang May
may@informatik.uni-freiburg.de

Technical Report No. 149

Institut für Informatik
Albert-Ludwigs-Universität
Georges-Koehler-Allee
79110 Freiburg, Germany

Feb. 26, 2001

Abstract

In this work, a logic-based framework for handling XML data is proposed. XPath-Logic embeds an extension of the XPath query language into first-order logic. We give a model-theoretic semantics of XPath-Logic formulas based on answer-sets. XPathLog is the Horn fragment of XPath-Logic, providing a logic-based language for manipulating and integrating XML data. Due to the close relationship with XPath, the semantics of rules is easy to grasp. In contrast to other approaches, the XPath syntax and semantics is also used for a declarative specification how the database should be *updated*: when used in rule heads, XPath filters are interpreted as specifications of elements and properties which should be added to the database. The formal semantics is defined wrt. a Herbrand structure which covers the XML data model. XPathLog has been implemented in LoPiX.

Contents

1 Introduction

XML has been designed and accepted as *the* framework for semi-structured data where it plays the same role as the relational model for classical databases. Specialized languages are available for XML querying, e.g., XQL [Rob99], XML-QL [DFF$^+$99], Quilt [CRF00] and for transformations of XML sources, e.g., XSL(T) [XSL99] (also XML-QL and Quilt can be used for generating new XML documents from given ones since their output format is XML), but yet none of them can be seen as an XML data *manipulation* language. For writing applications for creating and manipulating XML data, the dominating language is Java, regarding the DOM model as a data structure where applications are built on.

In this paper, we present a logic-based approach: An XML instance is mapped to a semantical structure which serves for interpreting *XPath-Logic* formulas. XPath-Logic is based on (i) first-order logic, and (ii) XPath reference expressions extended with variables. XPathLogic formulas can e.g. be used for specifying constraints and reasoning on XML documents.

The Horn fragment of XPath-Logic, called *XPathLog*, provides a declarative, Prolog-style language for manipulation and integration of XML documents. The syntax and querying semantics is based on XPath [XPa99]. Whereas XSLT, XML-QL [DFF$^+$99], and Quilt [CRF00] use XML patterns for generating output (with the consequence that their output can only *generate* XML, but it cannot be used for *manipulating* an existing XML instance), our language deviates from these approaches: XPathLog works on an abstract, Herbrand-like model where an XPath-like syntax is used for querying (rule bodies) *and* generating/manipulating the data (rule heads). The semantics is given as sets of variable bindings; only when final output is produced, the well-known XML syntax/model is used.

Structure of the paper. An introduction to XPathLog by examples is presented in Section 2. The formal part starts with Section 3, defining X-structures as semantical structures which represent XML documents. XPath-Logic is then defined in Section 4 as a logic which is interpreted over X-structures. The Horn fragment of XPath-Logic, XPathLog is investigated as a rule-based XML data manipulation language in Section 5, starting with the querying semantics. Section 6 defines the semantics of XPathLog rules, focusing on the semantics of rule heads for generating and modifying XML data, and the semantics and evaluation of XPathLog programs. Section 8 concludes the paper.

2 XPathLog by Examples

In this section, XPathLog is presented from an intuitive point of view as a logic-based XML query language. Sections 3, 4, and 5 will provide a formal definition.

2.1 XPath: The Core Language

XPath [XPa99] is the common language for addressing node sets in XML documents. It is based on navigation through the XML tree by path expressions of the form *//step/step/...step*. Every single step is of the form *axis::nodetest[filter]*, denoting that navigation goes along the given axis (i.e., child (default), descendant, parent, ancestor, preceding/following sibling, or attribute) in the XML document. Along the chosen axis, all elements which satisfy the *nodetest* (in most cases, the nodetest specifies an elementtype which subelements should be considered) are selected. From these, the ones qualify which satisfy the given *filter* – roughly spoken, a predicate over XPath expressions which state conditions on a node (for details, see [XPa99], or the formal semantics given in [Wad99a, Wad99b]). Starting with this (local) result set, the next step is applied.

The most frequently used axes are abbreviated as *path*/nodetest for *path*/child::nodetest, *path*//nodetest for *path*/descendant::nodetest, and *path*/@nodetest for *path*/attribute::nodetest.

Example 2.1 (XPath, Result Sets)
Consider the following excerpt of the MONDIAL *database [Mon] for illustrations.*

```
<!ELEMENT mondial (country+, organization+, ...)>
<!ELEMENT country (name, city+, ...)>
    <!ATTLIST country   car_code ID #REQUIRED        memberships IDREFS #IMPLIED
                        capital IDREFS #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT city (name, population*)>      <!ATTLIST city ...>
<!ELEMENT population (#PCDATA)>      <!ATTLIST population year CDATA #IMPLIED>
<!ELEMENT organization (name, abbrev, established?, members*)>
    <!ATTLIST organization   id ID #REQUIRED   seat IDREF #IMPLIED>
<!ELEMENT abbrev (#PCDATA)>
<!ELEMENT members EMPTY>
    <!ATTLIST members   type CDATA #REQUIRED   country IDREFS #REQUIRED>
<country car_code="CH" capital="cty-Bern" memberships="org-efta org-un ...">
    <name>Switzerland</name>
    <city id="cty-Bern" name="Bern"> <population year="91">134393</population> ...
    </city>
        ...
</country>
<organization id="org-un" seat="cty-USA-NewYork">
    <name>United Nations</name>  <abbrev>UN</abbrev>
    <members type="member" country="AL GR MK AND F E A CZ D H I FL SK SLO ..."/>
    <members type="observer" country="CH V"/>
</organization>
```

The XPath expression

```
//country[name]/city[population!text()>100000 and @zipcode]/name!text()
```

returns all names of cities s.t. the city belongs (i.e., is a subelement) to a country where a name subelement exists, the city's population is higher than 100000, and its zipcode is known.

XPath is only an addressing mechanism, not a full querying language like, e.g., the SQL querying construct. It provides the base for most XML querying languages, which extend it with their special constructs (e.g., functional style in XSL(T), and SQL/OQL style (e.g., joins) in Quilt). In the case of XPathLog, the extension feature are Prolog/Datalog style variable bindings, joins, and rules. A first step towards variable bindings in XPath has been introduced by XQL's return operators:

Example 2.2 (Output Operators)
The following XPath expression with return operators returns pairs (x_1, x_2) s.t. x_1 is the name of the country, and x_2 is the name of the city:

//country[@name?]//city[@population $>$ 100000 and @zipcode]/@name?

Here, the following "variables" are used:

- *country[@name?] defines an unnamed return variable (position 1 of the return tuple),*
- *@population $>$ 100000 defines an implicit local variable whose value is used only in the comparison,*
- *@zipcode defines a implicit* don't care variable *whose binding must be present but is never used,*
- *city/@name? defines another unnamed return variable (position 2 of the return tuple).*

2.2 XPathLog: Adding Variable Bindings to XPath

In Logic Programming languages, instead of a result set, for every match, a tuple of *variable bindings* is returned which can be used in the rule head. We extend the XPath syntax with the Prolog-style variable concept (and with implicit dereferencing):

Definition 2.1 (XPathLog: Syntax as derived from XPath)

- An XPathLog *reference expression* is an XPath expression where
  - every nodetest in a navigation step may be replaced by $\mathit{variable}_1$, $\mathit{nodetest}{\rightarrow}\mathit{variable}_2$, or even $\mathit{variable}_1{\rightarrow}\mathit{variable}_2$. Here, $\mathit{variable}_1$ is bound to the element name, extending the "*" navigation wildcard, and $\mathit{variable}_2$ is bound to the node.
  - for reference attributes @*name*, additional navigation steps may follow (implicit dereferencing).
  - the id function is not used (since XPathLog supports implicit dereferencing).
- A *predicate* is an expression of the form $p(X_1, \ldots, X_n)$ where the X_i are XPathLog reference expressions.
- XPathLogRules are logical rules over the above kinds of atoms (see Section 6.3). □

The variables are bound to the names/nodes/literals (for i.e., `CDATA` or `NMTOKENS` attributes) which result from the respective match; the formal semantics is based on that of XPath given in [Wad99a, Wad99b].

Example 2.3 (XPathLog)

Pure XPath expressions: *pure XPath expressions (i.e., without variables) are interpreted as* existential queries *which return* true *if the result set is non-empty:*

?- //country[name/text() = "Switzerland"]//city/name/text().
true

since the country element which has a name element with the text contents "Switzerland" contains at least one city descendant with a name element with non-empty text contents.

Output Result Set: *The query "?- xpath→N" for any xpath binds N to all nodes belonging to the result set of xpath:*

?- //country[name/text() = "Switzerland"]//city/name/text()→N.
N/"Bern"
N/"Geneva"
⋮

respectively, for a result set consisting of elements, logical ids are returned:

?- //country[name/text() = "Switzerland"]//city/name→N.
C/*bern*
C/*geneva*
⋮

Additional Variables: *XPathLog allows to bind all nodes which are considered by an expression (both by the access path to the result set, and in the filters): The following expression returns all tuples $(N1, C, N2)$ such that the city with name $N2$ belongs to the country with name $N1$ and car code C:*

?- //country[name/text()→N1 and @car_code→C]//city/name/text()→N2.
N2/"Bern" C/"CH" N1/"Switzerland"
N2/"Geneva" C/"CH" N1/"Switzerland"
⋮

Local Variables: *The following XPath expression returns all names of cities s.t. the city belongs to a country whose name is known and its population is higher than 100000, and its zipcode is known:*

?- //country[name/text()→N1]//city[population/text()→_P and @zipcode→_]/name/text()→N2,
_P > 100000.

The semantics of this query is a set of variable bindings for $N1$ and $N2$.

Dereferencing: *For every organization, give the name of the seat city and all names and types of members:*

?- //organization[name/text()→N and @abbrev/text()→A and @seat/@name/text()→SN]
/members[@type→MT]/@country/name/text()→MN.

One element of the result set is e.g.,

N/"..." A/"UN" SN/"New York" MT/"observer" MN/"Switzerland"

Navigation Variables: *Are there elements which have a name subelement with the* ***PCDATA*** *contents "Monaco", and of which type are they?*

?- //Type→X[name/text()→"Monaco"].
Type/country X/*country-monaco*
Type/city X/*city-monaco*

Schema Querying: *The use of variables at name positions further allows schema querying, e.g., to give all names of subelements of elements of type* city*:*

?- //city/N.
N/name
N/population
⋮

The schema querying functionality can also be used for validation wrt. a DTD or a given XML Schema (which can be present as an XML tree in the same XPathLog database).

Recall that XPath and related approaches do not support implicit dereferencing, but use the id(...) function which leads to confusing expressions if navigation is applied along several references.

Example 2.4 (Dereferencing)
The XPathLog expression

?- //organization[@seat = members/@country/@capital]/@seat/name/text()→N.

(all city names which are seats of an organization and the capital of one of its members) is equivalent to the XPath expression

id(//organization[id(./@seat) = id(id(./members/@country)/@capital)]/@seat)/name/text()

Aside: a similar "→"-operator for explicitly dereferencing has been defined in Quilt [CRF00]:

//organization[@seat = members→@country→@capital]→@seat/name/text()

We can additionally add variables to bind the names of the organization and of the country:

?- //organization[name/text()→ON;
@seat = members/@country[name/text()→CN]/@capital]/@seat/name/text()→N.

Further examples can be found and executed with the LoPiX *system [LoP].*

3 XML Instances as Semantical Structures

Logic-oriented frameworks are in general based on a semantical structure (from which a suitable Herbrand-style structure is derived). as a theoretical foundation. When considering XML instances, we see an XML instance as an abstract structure which can be queried by XPathLog.

- The universe consists of all nodes of the XML instance; additionally, it contains all splitted multivalued attribute values. In this paper, we use mnemonic ids for elements (which also shows the relationship to object-oriented models).
- elements have properties, defined by (i) subelements and (ii) attributes. Reference attributes are silently resolved by the corresponding internal ids.
- multivalued attributes (`NMTOKENS` and `IDREFS`) are silently split.

Definition 3.1 (X-Structure)
An X-structure is a tuple

$$\mathcal{X} = (\mathcal{N}, \mathcal{V}, \mathcal{L}, \mathcal{E}, \mathcal{A}, \mathcal{I})$$

where

- $\mathcal{N}$ is a set of names (i.e., element, attribute, and function names), sometimes distinguished as $\mathcal{N}_\mathcal{E}$, $\mathcal{N}_\mathcal{A}$, and $\mathcal{N}_\mathcal{F}$,
- $\mathcal{V}$ is a set of nodes (from the graph point of view, vertices),

- $\mathcal{L}$ is a set of literals (integers, floats, strings),
- $\mathcal{E}$ is a (partial) mapping
$$\mathcal{E} \; : \; \mathcal{V} \times \mathcal{N} \times \mathbb{N} \to \mathcal{V} \cup \mathcal{L}$$
(subelement relationship; from the graph point of view, edges; ordered),
- $\mathcal{A}$ is a (partial) mapping
$$\mathcal{A} \; : \; \mathcal{V} \times \mathcal{N} \to 2^{\mathcal{V}} \cup 2^{\mathcal{L}}$$
(attribute values), and
- $\mathcal{I}$ is a (partial) mapping
$$\mathcal{I} \; : \; \mathcal{N} \times \mathcal{V} \times (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^* \to \mathcal{V} \cup \mathcal{L} \cup \mathcal{N}$$
(interpretation of built-in functions).

Furthermore, $\mathcal{E}$ is partitioned into the subelement part and the text contents part:

$$\mathcal{E}_{Sub}(v,n) := \begin{cases} \mathcal{E}(v,n) & \text{if } \mathcal{E}(v,n) \subseteq \mathcal{L} \;, \\ \perp & \text{if } \mathcal{E}(v,n) \subseteq \mathcal{V} \;, \end{cases}$$
$$\mathcal{E}_{text}(v,n) := \begin{cases} \mathcal{E}(v,n) & \text{if } \mathcal{E}(v,n) \subseteq \mathcal{V} \;, \\ \perp & \text{if } \mathcal{E}(v,n) \subseteq \mathcal{L} \;. \end{cases}$$

$\mathcal{A}$ is partitioned into the literal-valued part, and the reference part:

$$\mathcal{A}_{Lit}(v,n) := \begin{cases} \mathcal{A}(v,n) & \text{if } \mathcal{A}(v,n) \subseteq \mathcal{L} \;, \\ \perp & \text{if } \mathcal{A}(v,n) \subseteq \mathcal{V} \;, \end{cases}$$
$$\mathcal{A}_{Ref}(v,n) := \begin{cases} \mathcal{A}(v,n) & \text{if } \mathcal{A}(v,n) \subseteq \mathcal{V} \;, \\ \perp & \text{if } \mathcal{A}(v,n) \subseteq \mathcal{L} \;. \end{cases}$$
□

Note the following:

- *XML attribute nodes* do not belong to $\mathcal{V}$, but to $\mathcal{L}$.
- for reference attributes (`IDREF`), the "results" are not the `ID`-strings, but the target nodes themselves,
- `PCDATA` children are handled by the special name `text`[1].
- We use mnemonic ids (e.g., *john*) or artificial ids (e.g., f_42) for elements of $\mathcal{V}$.

The elements of the $\mathcal{V}$-component of X-structures do not carry useful information in themselves, they are only of interest as anonymous entities (similar to object ids) which have certain properties that are given by $\mathcal{E}, \mathcal{A}$, and $\mathcal{I}$.

Definition 3.2 (Navigation Graph)
The *Navigation Graph* of an X-Structure is the graph $(\mathcal{V}, \mathcal{E}_{Sub} \cup \mathcal{A}_{Ref})$ which is induced by all element nodes, the subelement relationship, and the attribute references.

Two navigation graphs $(\mathcal{V}_1, \mathcal{E}_{Sub1} \cup \mathcal{A}_{Ref_1})$ $(\mathcal{V}_2, \mathcal{E}_{Sub2} \cup \mathcal{A}_{Ref_2})$ are *equivalent* if there is a bijective mapping between $\mathcal{V}_1$ and $\mathcal{V}_2$. □

[1] We assume that nobody defines an element type `text`.

Preview. Equivalence of navigation graphs can, e.g., be used to compare XML documents, search for overlappings (equivalence of XML subtrees), or as *annotated equivalence* when combined with an equivalence notion on literals (e.g., translation between English and German documents). □

There is a canonical mapping from the set of XML instances to the set of X-structures (modulo equivalence):

Definition 3.3 (Canonical X-Structure)
Given an XML instance $\mathcal{D}$, the canonical X-structure $\mathcal{X}_\mathcal{D}$ is defined as follows:

- Starting with the root node, the document is traversed in document order, assigning the id $\mathsf{f\_i}$ to the ith element node (inducing a bijective mapping Φ from element nodes to ids). $\mathcal{V}$ is the set of used ids.
- $\mathcal{N}$ is the set of names used in the XML instance (or in the DTD, if available) together with the names of the predefined functions in XML and XPath.
- The interpretations of names are defined as

$$\mathcal{E}(v, name, i) = \Phi(node) \text{ where } node \text{ is the } i\text{th subelement of node } \Phi^{-1}(v) \text{ and } \mathsf{name}(node) = name \ (name{=}\mathsf{text} \text{ if } node \text{ is a text node.})$$

$$\mathcal{A}(v, name) = \begin{cases} \{\Phi(node) \mid node \text{ is a target node of the } name \text{ attribute of } \Phi^{-1}(v)\} \\ \qquad \text{if } name \text{ is a reference attribute of } \Phi^{-1}(v)\ , \\ \{\ell \mid \ell \text{ is a value of the } name \text{ attribute of } \Phi^{-1}(v)\} \\ \qquad \text{if } name \text{ is a non-reference attribute of } \Phi^{-1}(v)\ , \end{cases}$$

$$\mathcal{I}(\mathsf{name}, v) = \begin{cases} \mathsf{name}(\Phi^{-1}(v)) & \text{if } \Phi^{-1}(v) \text{ is an element node} \\ \mathsf{text} & \text{if } \Phi^{-1}(v) \text{ is a text node} \end{cases}$$

The definitions of built-in functions in XPath contribute to $\mathcal{I}$. □

Remark 3.1
Since the navigation tree does not contain attribute nodes (they are mapped to literals or resolved into references), there is no definition of the name function for attributes. □

Example 3.1 (X-Structure)
The XML instance given in Example 2.3 is represented by the following X-structure (using mnemonic ids):

$\mathcal{E}$(*mondial*,country,42) = *ch*,
$\mathcal{A}$(*ch*,car_code) = {"CH"}, $\mathcal{A}$(*ch*,capital) = {*bern*},
$\mathcal{A}$(*ch*,memberships) = {*efta*,*un*,...}
$\mathcal{E}$(*ch*,name,1) = *ch-name*, $\mathcal{E}$(*ch*,city,2) = *bern*, $\mathcal{E}$(*ch*,city,3) = ...,
$\mathcal{E}$(*ch-name*,text,1) = "Switzerland",
$\mathcal{A}$(*bern*,name) = {"Bern"}, $\mathcal{E}$(*bern*,population,1) = *bern-pop*,
$\mathcal{E}$(*bern-pop*,text,1) = 134393,

$$
\begin{array}{ll}
\mathcal{E}(\textit{mondial},\textsf{organization},179) = \textit{un}, & \\
\mathcal{A}(\textit{un},\textsf{id}) = \{\text{“org-un”}\}, & \mathcal{A}(\textit{un},\textsf{seat}) = \{\textit{newYork}\}, \\
\mathcal{E}(\textit{un},\textsf{name},1) = \textit{un-name}, & \mathcal{E}(\textit{un-name},\textsf{text},1) = \text{“United Nations”}, \\
\mathcal{E}(\textit{un},\textsf{abbrev},2) = \textit{un-abbrev}, & \mathcal{E}(\textit{un-abbrev},\textsf{text},1) = \text{“UN”}, \\
\mathcal{E}(\textit{un},\textsf{members},3) = \textit{un-mem-mem}, & \\
\mathcal{A}(\textit{un-mem-mem},\textsf{type}) = \{\text{“member”}\}, & \mathcal{A}(\textit{un-mem-mem},\textsf{country}) = \{\textit{al}, \textit{gr}, \ldots\}, \\
\mathcal{E}(\textit{un},\textsf{members},4) = \textit{un-mem-obs}, & \\
\mathcal{A}(\textit{un-mem-obs},\textsf{type}) = \{\text{“observer”}\}, & \mathcal{A}(\textit{un-mem-obs},\textsf{country}) = \{\textit{ch},\textit{vat},\ldots\}.
\end{array}
$$

Note that the X-Structure contains references instead of id strings for `IDREF` attributes.

There are two requirements on (fragments of) X-structures which mirror the tree-nature of XML instances. They are required when an X-structure (or a fragment of an X-Structure) is regarded as an XML instance (thus, when regarding an X-structure simply as a logical notion for itself, they are not relevant):
Each node v of an X-structure is a potential root element for an output XML instance which recursively consists of all v's subelements and attributes. The following definition characterizes when the subtree rooted in v is a "useful" XML instance:

Definition 3.4 (Descendants and roots in an X-structure)
For an $\mathcal{X}$-structure, let

$$
\begin{array}{rl}
\mathsf{desc}(r, v) :\Leftrightarrow & \mathcal{E}(r, n, i) = v \text{ for some } n \text{ and some } i, \text{ or} \\
& \mathcal{E}(r, n, i) = x \text{ for some } n \text{ and some } i, \text{ and } \mathsf{desc}(x, v) \text{ holds} \quad .
\end{array}
$$

An element r is a *root* in $\mathcal{X}$ if

- for all v s.t. $\mathsf{desc}(r, v)$, not $\mathsf{desc}(v, r)$ holds, i.e., the descendant relation in the subtree rooted in r is acyclic, and
- for all v s.t. $v' \in \mathcal{A}_{Ref}(v, n)$ s.t. $\mathsf{desc}(r, v)$, also $\mathsf{desc}(r, v')$ holds, i.e., the target of the reference also belongs to the subtree rooted in r.

For every node v which is a root in $\mathcal{X}$, the size of the tree rooted is defined as

$$
\mathsf{treesize}(v) := 1 + |\{w : \mathsf{desc}(v, w) \text{ holds }\}| \qquad \Box
$$

The above condition is stronger than "well-formed XML instance" since this would not require well-definedness of `IDREF` attributes, but weaker than "valid" since it does not consider a DTD.

Definition 3.5 (Basic Result Sets: Axes)
For every context node x in an X-structure $\mathcal{X}$,

$$
\mathcal{A}_{\mathcal{X}}(a, x) \subseteq ((\mathcal{V} \cup \mathcal{L}) \times \mathcal{N})
$$

denotes the list of nodes generated by axis a from x (do not confuse $\mathcal{A}_{\mathcal{X}}$ with $\mathcal{A}$ which denotes the interpretation of attributes in $\mathcal{X}$). We write $\mathcal{A}_{\mathcal{X}}(a, x)[i, j]$ for the sublist consisting of the ith to jth elements, and $\mathcal{A}_{\mathcal{X}}(a, x) \downarrow_1$ for the projection of the list to the first component of the list elements.

$$
\begin{array}{lcl}
\mathcal{A}_{\mathcal{X}}(child, x) & = & list_{i \in \mathbb{N}}((y, name) \mid \mathcal{E}(x, name, i) = y) \\
\mathcal{A}_{\mathcal{X}}(attribute, x) & = & set((y, name) \mid \mathcal{A}(x, name) = y)
\end{array}
$$

For the other axes, $\mathcal{A}_{\mathcal{X}}(a,x)$ are derived from $\mathcal{A}_{\mathcal{X}}(child,x)$ according to the XML specification:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{if there is a unique } p \text{ s.t. } x \in \mathcal{A}_{\mathcal{X}}(child,p)\downarrow_1:}\\
\mathcal{A}_{\mathcal{X}}(parent,x) &=& ((p,\mathcal{I}(\mathsf{name}(p)))\ \text{s.t. } x \in \mathcal{A}_{\mathcal{X}}(child,p)\downarrow_1\\
\mathcal{A}_{\mathcal{X}}(preceding\text{–}sibling,x) &=& reverse(\mathcal{A}_{\mathcal{X}}(children,p)[1,i\text{–}1])\\
&& \quad \text{s.t. } p = \mathcal{A}_{\mathcal{X}}(parent,x)[1] \text{ and } x = \mathcal{A}_{\mathcal{X}}(child,p)[i]\\
\mathcal{A}_{\mathcal{X}}(following\text{–}sibling,x) &=& \mathcal{A}_{\mathcal{X}}(children,p)[i{+}1,last()]\\
&& \quad \text{s.t. } p = \mathcal{A}_{\mathcal{X}}(parent,x)[1] \text{ and } x = \mathcal{A}_{\mathcal{X}}(child,p)[i]
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{if both } \mathcal{A}_{\mathcal{X}}(parent,x) \text{ and } \mathcal{A}_{\mathcal{X}}(ancestor,\mathcal{A}_{\mathcal{X}}(parent,x)[1]) \text{ are defined:}}\\
\mathcal{A}_{\mathcal{X}}(ancestor,x) &=& ((p,n)) \circ \mathcal{A}_{\mathcal{X}}(ancestor,p) \text{ s.t. } \mathcal{A}_{\mathcal{X}}(parent,x)[1] = (p,n)\\
\multicolumn{3}{l}{\text{if } x \text{ is a root in } \mathcal{X}:}\\
\mathcal{A}_{\mathcal{X}}(descendant,x) &=& concat_{c\in\mathcal{A}_{\mathcal{X}}(child,x)}((c) \circ \mathcal{A}_{\mathcal{X}}(descendant,c))
\end{array}
$$

□

Remark 3.2
Note that in contrast to the XML/XPath data model and the semantics given in [Wad99b],

- $\mathcal{A}$ does not return a list of elements, but a list of pairs (name, element). This allows for overlapping trees (in this case, the parent is ambiguous) and for the direct resolving of attributes. As a consequence, the name function is not needed for node tests. □

4 XPathLogic: A Logic for X-Structures

Since XPathLogic is based on the notions of first-order logic, a short review of first-order logic is given, introducing the notation in this work.

First-Order Logic Each first-order language contains a set of distinguished symbols, consisting of parentheses "(" and ")", constants true, false representing the truth values, boolean connectives $\neg$, $\wedge$, $\vee$, quantifiers $\forall$, $\exists$, and an infinite set Var of variables $x, y, x_1, x_2, \ldots$.

An individual first-order language is then given by its *signature* Σ. Σ is partitioned into a functional part $\mathcal{F}$ of function symbols and a relational part $\mathcal{P}$ of predicate symbols, each of the symbols with a given arity which is denoted by $\text{ord}(f)$ and $\text{ord}(p)$, respectively. 0-ary functions are also called *constants*.
The set Term_Σ of *terms* over Σ is defined inductively as

- each variable is a term,
- for $f \in \mathcal{F}$, $\text{ord}(f) = n$ and terms $t_1, \ldots, t_n$, also $f(t_1, \ldots, t_n)$ is a term.

The set of *atomic formulas* over Σ is given as

$$\text{Atoms}_\Sigma := \{s = t \mid s,t \in \text{Term}_\Sigma\} \cup \{p(t_1,\ldots,t_n) \mid p \in \mathcal{P},\ \text{ord}(p) = n,\ t_1,\ldots,t_n \in \text{Term}_\Sigma\}\ .$$

The main difference between XPath-Logic and first-order logic is that XPath-Logic has an additional type of atomic formulas: *reference expressions* which turn out to be similar to predicates.
The set of *first-order formulas* over Σ is defined as the least set with the following properties:

- all atomic formulas are formulas,
- true and false are formulas,
- for formulas A and B, a variable x, $\neg A$, $A \vee B$, $A \wedge B$, $\forall x : A$, and $\exists x : A$ are formulas.

The notions of bound and free variables are defined in the usual way, free(F) denoting the set of variables occurring free in a set F of formulas.

A *substitution* (over a signature Σ) is a mapping $\sigma : \mathrm{Var} \to \mathrm{Term}_\Sigma$ where $\sigma(x) \neq x$ for only finitely many $x \in \mathrm{Var}$. $\sigma : \sigma(x) = t$ is written as $[x \leftarrow t]$. Substitutions are extended to terms and formulas as usual.

The semantics of first-order logic is given by *first-order structures* over a given signature:

Definition 4.1 (First-Order Structure)
A *first-order structure* $\mathcal{I} = (I, \mathcal{U})$ over a signature Σ consists of a nonempty set $\mathcal{U}$ (*universe*) and an *interpretation* I of the signature symbols over $\mathcal{U}$ which maps

- every constant c to an element $I(c) \in \mathcal{U}$,
- every n-ary function symbol f to an n-ary function $I(f) : \mathcal{U}^n \to \mathcal{U}$,
- every n-ary predicate symbol p to an n-ary relation function $I(p) : \mathcal{U}^n \to \{\mathsf{true}, \mathsf{false}\}$.

For short, I consists of two mappings $I_F : \mathcal{F} \to \bigcup_{k=0}^{\infty} \mathcal{U}^{(\mathcal{U}^k)}$ and $I_P : \mathcal{P} \to \bigcup_{k=1}^{\infty} 2^{(\mathcal{U}^k)}$ $\square$

A *variable assignment* over a universe $\mathcal{U}$ is a mapping

$$\chi : \mathrm{Var} \to \mathcal{U} \ .$$

For a variable assignment χ, a variable x, and $d \in \mathcal{U}$, the *modified* variable assignment χ_x^d is identical with χ except that it assigns d to the variable x:

$$\chi_x^d : \mathrm{Var} \to \mathcal{U} : \begin{cases} y \mapsto \chi(y) & \text{if } y \neq x \ , \\ x \mapsto d & \text{otherwise.} \end{cases}$$

4.1 Syntax

Inspired by First-Order Logic as a logic for dealing with First-Order interpretations, XPath-Logic is defined for expressing properties of X-Structures. XPathLog is then the Horn fragment of XPath-Logic. XPathLog *LocationPaths* provide the "basic" components of the language.

Definition 4.2 (XPath-Logic: Syntax)
The set of basic formulas of an XPath-Logic language over an alphabet $\mathcal{N}$ of names is defined as follows:

- An *XPath-Logic reference expression* is an XPath *AbsoluteLocationPath* [XPa99, Chap. 2] using these names where the XPath syntax is extended as follows (we refer to the definition numbering in [XPa99]):
  - In XPath-Logic steps, $axis :: name$ may be replaced by $axis :: name \to V$, $axis :: V$, or $axis :: V \to W$ where V and W are variables:

```
[4] Step ::= AxisSpecifier NodeTest Predicate*
           | AxisSpecifier NodeTest "->" Var Predicate*
           | AxisSpecifier Var Predicate*
           | AxisSpecifier Var "->" Var Predicate*
```

  - XPath-Logic *LocationPaths* may begin with nodes of the universe which are given as constants or bound to variables:

```
[2b] ConstantLocationPath ::= constant "/" RelativeLocationPath
                             | variable "/" RelativeLocationPath
```

– If *predExpr* is an XPath-Logic predicate expression, then also $\exists x : predExpr$ is an XPath-Logic filter expression:

```
[9] PredicateExpr ::= Expr
                    | exists Var ":" "(" Expr ")"
```

– Additionally, the comparison and operations of node sets are allowed:

```
[18] UnionExpr ::= PathExpr
                 | UnionExpr ("|"|"union") UnionExpr
                 | UnionExpr "intersect" UnionExpr
                 | UnionExpr "minus" UnionExpr


[23] EqualityExpr ::= RelationalExpr
                    | EqualityExpr "=" EqualityExpr
                    | EqualityExpr "!=" EqualityExpr
                    | UnionExpr "==" UnionExpr  % use the LaTeX \equiv
                    | UnionExpr "subseteq" UnionExpr
                    | UnionExpr "subsetneq" UnionExpr
```

– Note that by rules [4], [5], and [6] of [XPa99], expressions of the form `a/@b/c` are allowed. The XPath semantics given in [Wad99b] evaluates any navigation starting from attribute values to false. In our semantics, we allow for navigation following `IDREF` attributes.

- An *XPath-Logic predicate* is
  – an XPath-Logic reference expression (evaluates to true if the result set of the reference expression is non-empty), or
  – a predicate over XPath-Logic reference expressions, □

4.2 Semantics

4.2.1 Data Model

The semantics of XPath-Logic reference expressions is defined wrt. an X-structure $\mathcal{X}$ as an *annotated result list*, i.e., every formula fml is mapped to a list of pairs where each pair consists of

(i) a result element, and

(ii) a list of bindings for some variables. In our approach, not only the result lists are ordered, but also the associated variable bindings are ordered.

Definition 4.3 (Semantics)
In the sequel,

$$\text{Var\_Bindings} := \bigcup_{n \in \mathbb{N}_0} 2^{((\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^n)^{(\text{Var}^n)}}$$

is the set of sets of variable assignments. We use χ for denoting an individual variable binding, and $\xi \in$ Var_Bindings for denoting a set of variable bindings.
The set $\{true\}$ is the lowest element in Var_Bindings (representing the set with an empty variable assignment); in contrast, $\emptyset$ means that there is no variable binding which satisfies a given requirement.

$$\text{AnnotatedResults} := ((\mathcal{V} \cup \mathcal{L}) \times \text{Var\_Bindings})$$

is the set of annotated results. We use θ for denoting an annotated result list.
The semantics[2] $\langle\!\langle \varphi \rangle\!\rangle_{\mathcal{X}}$ of an XPath-Logic reference expression φ is defined wrt. an X-structure $\mathcal{X}$ as an *annotated result list*

$$\langle\!\langle\, . \,\rangle\!\rangle_{\mathcal{X}} : \text{Fml} \rightarrow ((\mathcal{V} \cup \mathcal{L}) \times \text{Var\_Bindings})^{\mathbb{N}}$$

The semantics of a predicate is either $(true, \xi)$ for some $\xi \in$ Var_Bindings, or $(false, \emptyset)$ (the special case of the above where there is no result element). □

Remark 4.1
The semantics can be characterized equivalently, but less formally as follows:

(i) a result list, and

(ii) with every element of the result list, a list of variable bindings is associated.

The result list (i) is closely related to the one defined for XPath for the expression obtained from φ when deleting all variables - but take care of join variables. □

Example 4.1 (Semantics)
First, the semantics is illustrated by an example, showing the difference between the result set and the additional bindings. Let $\mathcal{X}$ be the XML structure given in Example 3.1, let

$$\begin{aligned} expr := {} & \mathsf{//organization{\rightarrow}O[member/@country[@car\_code{\rightarrow}C\ and\ name/text(){\rightarrow}N]]} \\ & \mathsf{/abbrev/text(){\rightarrow}A.} \end{aligned}$$

$$\langle\!\langle\, expr \,\rangle\!\rangle_{\mathcal{X}} =$$

$$\begin{array}{l} \mathsf{list((\text{“UN”}, \{O/\mathit{un},\ A/\text{“UN”},\ C/\text{“AL”},\ N/\text{“Albania”},} \\ \qquad\qquad\quad \mathsf{O/\mathit{un},\ A/\text{“UN”},\ C/\text{“GR”},\ N/\text{“Greece”},} \\ \qquad\qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad \mathsf{\}),} \\ \quad \mathsf{(\text{“EU”}, \{O/\mathit{eu},\ A/\text{“EU”},\ C/\text{“D”},\ N/\text{“Germany”},} \\ \qquad\qquad\quad \mathsf{O/\mathit{eu},\ A/\text{“EU”},\ C/\text{“F”},\ N/\text{“France”},} \\ \qquad\qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad \mathsf{\}),} \\ \quad\quad \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{)} \end{array}$$

Thus, searching for the 7th element of the above result set with

$$\langle\!\langle\, //organization \rightarrow O[\ldots][7]/abbrev/text() \rightarrow A \,\rangle\!\rangle_{\mathcal{X}}$$

yields the name of the 7th organization, together with a set of variable bindings for the other variables. This is much different from defining a semantics yielding simply a list of variable bindings $\mathsf{O}/\ldots$, $\mathsf{A}/\ldots$, $\mathsf{C}/\ldots$, $\mathsf{N}/\ldots$ *which would presumably yield the variable bindings for the 7th member of the UN.*

[2] we do not use the usual [[...]] notation since it can be confused with the XPath filter brackets.

Remark 4.2 (Ordering?)
Recall that the result sets of XPath expressions are *unordered* [XPa99]:

```
"The primary syntactic construct in XPath is the expression.
An expression matches the production Expr. An expression is
evaluated to yield an object, which has one of the following
four basic types:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)"
```

Only internally to the computation, when single steps are considered, there is a *current node list* which is used for applying the position selection operator.

Although, most implementations return the elements of the result set in document ordering. In this section, we follow the unordered definition. Nevertheless, similar to the implementations, our semantics is *list-based* – using the fact that from the object-oriented point of view, the class *list* is a specialization of the class *(multi)set*. We use a list constructor

$$list_{domain}(elements)$$

where *domain* specifies a set of items and *element* gives for each item a term which is appended to the list. We do neither fix an enumeration of the set *domain*, nor, how append works. Nevertheless, the straightforward intuition how to operate on lists provides a good understanding, and presumably concides with implementations. □

Definition 4.4 (Operators on Annotated Result Lists)
First, operations are defined which are needed for evaluating XPath-Logic reference expressions.

Join: The usual relational join is extended to a polymorphic join operator on semantics:

- When joining sets of variable bindings,

$$\begin{aligned} \bowtie : \quad & \text{Var\_Bindings} \times \text{Var\_Bindings} \to \text{Var\_Bindings} \\ & (\xi_1, \xi_2) \mapsto \xi_1 \bowtie_{rel} \xi_2 \end{aligned}$$

  where $\bowtie_{rel}$ is the usual relational join (on the variable names which occur in both arguments).

- The join of an element of an annotated result list with a list of variable bindings is defined as

$$\begin{aligned} \bowtie : \quad & \text{AnnotatedResults} \times \text{Var\_Bindings} \to \text{AnnotatedResults} \\ & ((x, \xi_1), \xi_2) \mapsto (x, \xi_1 \bowtie \xi_2) \end{aligned}$$

  joining the sets of variable bindings (used for adjoining filters to paths in Definition 4.5(6)).

- The join of an annotated result list with an element of an annotated result list is defined as

$$\bowtie\ :\ \ \text{AnnotatedResults}^{\mathbb{N}} \times \text{AnnotatedResults} \to \text{AnnotatedResults}^{\mathbb{N}}$$
$$(\theta,(x,\xi)) \mapsto \begin{cases} \text{if } x \in \text{Res}(\theta)\text{:} & \text{replace } (x, \text{Bdgs}(\theta,x)) \text{ by } (x, \xi \bowtie \text{Bdgs}(\theta,x)) \text{ in } \theta\ . \\ \text{else:} & \text{append } (x,\xi) \text{ to } \theta\ . \end{cases}$$

i.e., the list of variable bindings is added to the appropriate element of the annotated result set.
The "else"-case is not used in our definitions.

Operations on lists of variable bindings: For two sets of variable bindings,

$$\begin{aligned} \cup\ :\ & \text{Var\_Bindings} \times \text{Var\_Bindings} \to \text{Var\_Bindings} \\ & (\xi_1,\xi_2) \mapsto \text{restrict}_{\text{free}((\xi_1)\cap\text{free}(\xi_2))}(\xi_1) \cup \text{restrict}_{\text{free}((\xi_1)\cap\text{free}(\xi_2))}(\xi_2) \\ \cap\ :\ & \text{Var\_Bindings} \times \text{Var\_Bindings} \to \text{Var\_Bindings} \\ & (\xi_1,\xi_2) \mapsto \text{restrict}_{\text{free}((\xi_1)\cap\text{free}(\xi_2))}(\xi_1) \cap \text{restrict}_{\text{free}((\xi_1)\cap\text{free}(\xi_2))}(\xi_2) \end{aligned}$$

Similar to relational set comparisons,

$$\begin{aligned} \xi_1 \equiv \xi_2 \ &:\Leftrightarrow\ \xi_1 \cap \xi_2 = \xi_1 = \xi_2 \\ \xi_1 \subset \xi_2 \ &:\Leftrightarrow\ \xi_1 \cap \xi_2 = \xi_1 \end{aligned}$$

Operations on annotated result lists:

$$\begin{aligned} \cup\ :\ & \text{AnnotatedResults}^{\mathbb{N}} \times \text{AnnotatedResults}^{\mathbb{N}} \to \text{AnnotatedResults}^{\mathbb{N}} \\ & (\theta_1,\theta_2) \mapsto \mathit{list}_{x\in\text{Res}(\theta_1)\cup\text{Res}(\theta_2)}(x, \text{Bdgs}(\theta_1,x) \cup \text{Bdgs}(\theta_2,x)) \\ \cap\ :\ & \text{AnnotatedResults}^{\mathbb{N}} \times \text{AnnotatedResults}^{\mathbb{N}} \to \text{AnnotatedResults}^{\mathbb{N}} \\ & (\theta_1,\theta_2) \mapsto \mathit{list}_{x\in\text{Res}(\theta_1)\cup\text{Res}(\theta_2)}(x, \text{Bdgs}(\theta_1,x) \cap \text{Bdgs}(\theta_2,x)) \end{aligned}$$

$$\begin{aligned} \theta_1 \subseteq \theta_2 \ :\Leftrightarrow\quad & \text{there is an injective mapping } \varphi : \text{Res}(\theta_1) \to \text{Res}(\theta_2) \\ & \text{s.t. } \text{Bdgs}(\theta_1,x) \subseteq \text{Bdgs}(\theta_2,x) \\ \theta_1 = \theta_2 \ :\Leftrightarrow\quad & \text{there is a bijective mapping } \varphi : \text{Res}(\theta_1) \to \text{Res}(\theta_2) \\ & \text{s.t. } \text{Bdgs}(\theta_1,x) = \text{Bdgs}(\theta_2,x) \end{aligned}$$

□

4.2.2 Semantics of Expressions

The basic (non-annotated) result lists are provided by $\mathcal{A}_\mathcal{X}(axis, v)$ for every node v in $\mathcal{X}$ (cf. Definition 3.5). Recall that for $\mathcal{A}_\mathcal{X}(attribute, x)$, the result set contains literals in case of non-reference attributes, and element nodes in case of reference attributes. The set $\mathcal{A}_\mathcal{X}(attribute, x)$ can be regarded as a list by an arbitrary enumeration.

Definition 4.5 (Semantics of XPath-Logic expressions)
The semantics is defined by operators $\mathcal{S}$ and $\mathcal{Q}$ derived from the formal semantics given in [Wad99b] (the second of the semantics given there).

- $\mathcal{S}_\mathcal{X}\ :\ (\text{Axes} \times \text{Reference\_Expressions} \times \mathcal{V}) \to \text{AnnotatedResults}^{\mathbb{N}}$
  evaluates reference expressions wrt. an axis and a context node and returns a list of pairs as described above. Arithmetic expressions are also reference expressions.
- $\mathcal{Q}_\mathcal{X}\ :\ \text{Filter\_Expressions} \times \mathcal{V} \to \text{Var\_Bindings}$
  evaluates filter expressions wrt. a context node and returns a set of variable bindings.

- $\mathcal{S}$ is polymorphic in the sense that it may be "called" with two additional integer arguments (information about the size of the current context list and the position of the current element).
- Additionally, $\mathcal{I}$ provides an interpretation of XPath built-in functions and predicates according the the specification [XPa99].

Expressions are evaluated by $\mathcal{S}$:

1. entry points: rooted path
$$\mathcal{S}^{any}_{\mathcal{X}}(/p) = \mathcal{S}^{child}_{\mathcal{X}}(p, root)$$
2. entry points: constants $x \in \mathcal{V}$:
$$\mathcal{S}^{any}_{\mathcal{X}}(x/p) = \mathcal{S}^{child}_{\mathcal{X}}(p, x)$$
(this is mainly of interest when multiple documents are used; but also in the core XPathLog language, names/identifiers may be associated to nodes, see Section 6.3)
3. entry points: variables $v \in$ Var:
$$\mathcal{S}^{any}_{\mathcal{X}}(v/p) = list_{x \in \mathcal{A}_{\mathcal{X}}(descendants, root)\downarrow_1}(\mathcal{S}^{child}_{\mathcal{X}}(p, x) \bowtie (x, \{v/x\}))$$
4. Single step:
$$\mathcal{S}^{a}_{\mathcal{X}}(axis :: pattern, x) = \mathcal{S}^{axis}_{\mathcal{X}}(pattern, x)$$
5. node test, no filter:
$$\begin{array}{lcl}
\mathcal{S}^{a}_{\mathcal{X}}(name, x) & = & list_{pair \in \mathcal{A}_{\mathcal{X}}(a,x),\ pair\downarrow_2 = name}(pair\downarrow_1, \{true\}) \\
\mathcal{S}^{a}_{\mathcal{X}}(name \to C, x) & = & list_{pair \in \mathcal{A}_{\mathcal{X}}(a,x),\ pair\downarrow_2 = name}(pair\downarrow_1, \{C/pair\downarrow_1\}) \\
\mathcal{S}^{a}_{\mathcal{X}}(N, x) & = & list_{pair \in \mathcal{A}_{\mathcal{X}}(a,x)}(pair\downarrow_1, \{N/pair\downarrow_2\}) \\
\mathcal{S}^{a}_{\mathcal{X}}(N \to C, x) & = & list_{pair \in \mathcal{A}_{\mathcal{X}}(a,x)}(pair\downarrow_1, \{N/pair\downarrow_2, C/pair\downarrow_1\})
\end{array}$$
6. node test with filter (let $\theta := \mathcal{S}^{a}_{\mathcal{X}}(nodetest, x)$):
$$\begin{array}{l}
\mathcal{S}^{a}_{\mathcal{X}}(nodetest[filter], x) \quad = \\
\qquad list_{y \in \mathrm{Res}(\theta),\ \mathcal{Q}_{\mathcal{X}}(filter, y, k, n, \mathrm{Bdgs}(\theta, y)) \neq \emptyset}(y,\ \mathcal{Q}_{\mathcal{X}}(filter, y, k, n, \mathrm{Bdgs}(\theta, y)))
\end{array}$$
$$\begin{array}{l}
\mathcal{S}^{a}_{\mathcal{X}}(nodetest \to C[filter], x) \quad = \\
\qquad \underset{\substack{y \in \mathrm{Res}(\mathcal{S}^{a}_{\mathcal{X}}(nodetest, x)),\\ \mathcal{Q}_{\mathcal{X}}(filter, y, k, n, \mathrm{Bdgs}(\theta, y)) \neq \emptyset}}{list}(y,\ \mathrm{Bdgs}(\mathcal{S}^{a}_{\mathcal{X}}(nodetest \to C, x), y) \bowtie \mathcal{Q}_{\mathcal{X}}(filter, y, k, n, \mathrm{Bdgs}(\theta, y)))
\end{array}$$
where with
$$L_1 := list_{y \in \mathrm{Res}(\mathcal{S}^{a}_{\mathcal{X}}(name, x)),\ \mathcal{Q}_{\mathcal{X}}(filter, y, k, n, \mathrm{Bdgs}(\theta, y)) \neq \emptyset}(y) \ ,$$
$n := size(L_1)$, and for every y, let j the index of x_1 in L_1, $k := j$ if a is a forward axis, and $k := n+1-j$ if a is a backward axis (cf. [Wad99b]). k and n are only used if the filter contains a position() or last() predicate.

   Note that we use here the case that, if there are no variable bindings which result from the filter, $\{true\}$ is returned from $\mathcal{Q}_{\mathcal{X}}(filter, y, k, n)$, whereas $\mathcal{Q}_{\mathcal{X}}(filter, y, k, n) = \emptyset$ denotes that the filter is not satisfied.

7. Path:

$$\mathcal{S}^a_{\mathcal{X}}(p_1/p_2, x) = list_{y \in \mathrm{Res}(\mathcal{S}^a_{\mathcal{X}}(p_1,x))}(\mathcal{S}^a_{\mathcal{X}}(p_2, y) \bowtie (y, \mathrm{Bdgs}(\mathcal{S}^a_{\mathcal{X}}(p_1, x), y)))$$

8. Function applications:

$$\mathcal{S}^a_{\mathcal{X}}(p_1!f(args), x) = list_{y \in \mathrm{Res}(\mathcal{S}^a_{\mathcal{X}}(p_1,x))}(\mathcal{S}_{\mathcal{X}}(f(args), y) \bowtie (y, \mathrm{Bdgs}(\mathcal{S}^a_{\mathcal{X}}(p_1, x), y)))$$

9. text children:

$$\mathcal{S}^a_{\mathcal{X}}(text(), x) = list_{y \in \mathcal{A}_{\mathcal{X}}(a,x),\ \mathsf{nodetype}(y) = \mathsf{text}}(y, \{true\})$$

10. Position selection:

$$\mathcal{S}^a_{\mathcal{X}}(p[i], x) = i\text{th element of } \mathcal{S}^a_{\mathcal{X}}(p, x)$$

Filters, Predicates etc.

11. Reference expressions (existential semantics) in filters:

$$\mathcal{Q}_{\mathcal{X}}(refExpr, x, k, n, Bdgs) = \bigcup_{y \in \mathrm{Res}(\mathcal{S}^a_{\mathcal{X}}(refExpr,x))} \mathrm{Bdgs}(\mathcal{S}^a_{\mathcal{X}}(refExpr, x), y)$$

12. Predicate expressions:

$$\mathcal{Q}_{\mathcal{X}}(pred(A_1, \ldots, A_n), x, k, n, Bdgs) = \bigcup_{\substack{x_1 \in \mathrm{Res}(\mathcal{S}^a_{\mathcal{X}}(A_1,x,k,n)), \ldots, \\ x_n \in \mathrm{Res}(\mathcal{S}^a_{\mathcal{X}}(A_n,x,k,n)), \\ \mathcal{I}(pred(x_1,\ldots,x_n)) = true}} \mathrm{Bdgs}(\mathcal{S}^a_{\mathcal{X}}(A_1, x_1, k, n)) \bowtie \ldots \bowtie \mathrm{Bdgs}(\mathcal{S}^a_{\mathcal{X}}(A_1, x_n, k, n))$$

Recall that $\mathcal{I}$ is an interpretation of predicates, including built-in predicates of XPath.

13. Boolean Connectives:

$$\begin{aligned} \mathcal{Q}_{\mathcal{X}}(A \text{ and } B, x, k, n, Bdgs) &= \mathcal{Q}_{\mathcal{X}}(A, x, k, n, Bdgs) \bowtie \mathcal{Q}_{\mathcal{X}}(B, x, k, n, Bdgs) \\ \mathcal{Q}_{\mathcal{X}}(A \text{ or } B, x, k, n, Bdgs) &= \mathcal{Q}_{\mathcal{X}}(A, x, k, n, Bdgs) \cup \mathcal{Q}_{\mathcal{X}}(B, x, k, n, Bdgs) \end{aligned}$$

note that in the or-case the resulting variable bindings may differ (e.g. in the case r(A,C) and (p(A,C) or q(A)) where C is a (join) variable which is defined in the first literal). In this case, subsumed bindings can be removed.

14. Negated expressions which do *not contain any free variable*:

$$\mathcal{Q}_{\mathcal{X}}(\text{not } A, x, k, n, Bdgs) = \begin{cases} \{true\} & \text{if } \mathcal{Q}_{\mathcal{X}}(A, x, k, n, Bdgs) = \emptyset, \\ \emptyset & \text{otherwise, i.e., if } \mathcal{Q}_{\mathcal{X}}(A, x, k, n, Bdgs) = \{true\}. \end{cases}$$

Negated expressions which contain free variables are handled separately, dealing with safety issues, in Definition 4.6.

15. Existential quantification

$$\mathcal{Q}_{\mathcal{X}}(\exists X : A, x) = \mathrm{Bdgs}(\mathcal{Q}_{\mathcal{X}}(A, x, k, n, Bdgs)) \text{ restricted to all variables } Y \neq X$$

By default, duplicates are not removed.

Evaluation of Functions etc.

16. constants (in free predicates or in comparisons in filters); let *value* be a string or a number:
$$\mathcal{S}^a_{\mathcal{X}}(value, x) = (value, \{true\})$$

17. functions:
$$\mathcal{S}^a_{\mathcal{X}}(f(args), y) = (\mathcal{I}(f, y, args), \{true\})$$
Recall that $\mathcal{I}$ is an interpretation of functions, including built-in functions of XPath.

18. context-related properties:
$$\mathcal{S}^a_{\mathcal{X}}(position(), x, k, n) = (k, \{true\})$$
$$\mathcal{S}^a_{\mathcal{X}}(last(), x, k, n) = (n, \{true\})$$

19. arithmetics:
$$\mathcal{S}^a_{\mathcal{X}}(expr_1 \ \mathsf{op}\ expr_2, x) =$$
$$list_{\substack{x \in \mathrm{Res}(\mathcal{S}^a_{\mathcal{X}}(expr_1)),\\ y \in \mathrm{Res}(\mathcal{S}^a_{\mathcal{X}}(expr_2)),\\ \text{nested loop}}} (x \ \mathsf{op}\ y, \mathrm{Bdgs}(\mathcal{S}^a_{\mathcal{X}}(expr_1), x) \bowtie \mathrm{Bdgs}(\mathcal{S}^a_{\mathcal{X}}(expr_2), y))$$

Handling Result Sets.

20. Alternative/Union
$$\mathcal{S}^a_{\mathcal{X}}(p_1|p_2, x) = \mathcal{S}^a_{\mathcal{X}}(p_1 \ \mathsf{union}\ p_2, x) = \mathcal{S}^a_{\mathcal{X}}(p_1, x) \cup \mathcal{S}^a_{\mathcal{X}}(p_2, x)$$

21. Intersection
$$\mathcal{S}^a_{\mathcal{X}}(p_1 \ \mathsf{intersect}\ p_2, x) = \mathcal{S}^a_{\mathcal{X}}(p_1, x) \cap \mathcal{S}^a_{\mathcal{X}}(p_2, x)$$
□

Theorem 4.1 (Correctness)
For variable-free expressions without navigating along reference attributes, the semantics coincides with the one given in [Wad99b]: For every XPath expression *expr*,
$$\mathrm{Res}(\mathcal{S}^{any}_{\mathcal{X}}(expr)) = [[expr]]$$
where [[*expr*]] is as defined in [Wad99b]. □

Proof. Structural induction. The definition is an extension of the one in [Wad99b] to annotated result sets. □

For negated expressions in filters, the semantics of Logic Programming languages is used: negated expressions which contain free variables are intended to *exclude* some bindings from a given set of potential results. Thus, these variables are required to be *safe*, i.e., to get their bindings from somewhere else. Thus, negated literals do not create new variable bindings. Thus, they are evaluated *after* positive literals. Then, the safe variables are known, together with the bindings obtained from evaluating positive literals and paths.

Definition 4.6 (Negation)
The semantics of negated literals in filters is defined wrt. a set *SV* of safe variables, and a set *Bdg* of variable bindings.

- All variables which occur free in the negated expression must be safe, i.e., every tuple of variable bindings has to provide a value for them.

For the enhanced definition of the semantics of filters, the set of variable bindings is propagated to the filter evaluation:

- Single step with filter:

$$\begin{aligned}&\mathcal{S}^a_{\mathcal{X}}(nodetest[filter], x) = \\ &\quad list_{y \in \mathrm{Res}(\theta),\ \mathcal{Q}_{\mathcal{X}}(filter,y,k,n,\mathrm{Bdgs}(\theta)) \neq \emptyset}(y,\ \mathcal{Q}_{\mathcal{X}}(filter, y, k, n, \mathrm{Bdgs}(\theta)))\end{aligned}$$

  where $\theta := \mathcal{S}^a_{\mathcal{X}}(nodetest, x)$

  (analogously for $nodetest \to C[filter], x$).

- Boolean Connectives:

$$\begin{aligned}\mathcal{Q}_{\mathcal{X}}(A \text{ and } B, x, k, n, Bdgs) &= (x, \mathrm{Bdgs}(\mathcal{Q}_{\mathcal{X}}(A, x, k, n, Bdgs)) \bowtie \mathrm{Bdgs}(\mathcal{Q}_{\mathcal{X}}(B, x, k, n, Bdgs))) \\ \mathcal{Q}_{\mathcal{X}}(A \text{ or } B, x, k, n, Bdgs) &= (x, \mathrm{Bdgs}(\mathcal{Q}_{\mathcal{X}}(A, x, k, n, Bdgs)) \cup \mathrm{Bdgs}(\mathcal{Q}_{\mathcal{X}}(B, x, k, n, Bdgs)))\end{aligned}$$

- Negated Expressions: For defining $\mathcal{Q}_{\mathcal{X}}(\text{ not } A, x, k, n, Bdgs)$, The set SV of safe variables is the set of those variables which occur in every tuple in $Bdgs$. Only variables in SV occur free in A, thus A binds only them. For two tuples ξ_1, ξ_2, of variable bindings, let $\xi_1 \leq \xi_2$ if ξ_1 is subsumed by ξ_2 (i.e., all variable bindings in ξ_1 occur also in ξ_2). Intuitively, in this case, if ξ_1 is "abandoned", ξ_2 should also be abandoned.

$$\mathcal{Q}_{\mathcal{X}}(\text{not } A, x, k, n, Bdgs) = Bdgs - \{\xi \in Bdgs \mid \text{ there is a } \xi' \in \mathcal{Q}_{\mathcal{X}}(A, x, k, n) \text{ s.t. } \xi \leq \xi'\}$$

$\square$

4.3 Formulas

XPath-Logic Formulas are built over predicates and reference expressions, using $\wedge$, $\vee$, $\neg$, $\exists$, and $\forall$.

Definition 4.7 (Semantics of XPath-Logic Formulas)
Formulas are interpreted according to the usual first-order semantics

$$\models\ \subseteq\ (\text{Structures} \times \text{Var\_Assignments} \times \text{Formulas})$$

22. Reference Expressions: The semantics $\langle\!\langle\, refExpr \,\rangle\!\rangle$ corresponds to the interpretation of a *predicate symbol* in first-order logic. From the annotated result sets, only the resulting Bindings are taken:

$$(\mathcal{X}, \chi) \models refExpr \ :\Leftrightarrow\ \chi \in \text{Var\_Bindings}(\mathcal{S}^{any}_{\mathcal{X}}(refExpr))\ .$$

23. Predicates: For "free" predicates (i.e., those not occurring in a filter), the semantics of filter predicates can be used (since they are in fact the same as filter predicates where all arguments are non-relative paths, i.e., absolute, or beginning with a constant):

$$(\mathcal{X}, \chi) \models pred(A_1, \ldots, A_n) \ :\Leftrightarrow\ \chi' \in \mathcal{Q}_{\mathcal{X}}(pred(A_1, \ldots, A_n), null, -1, -1)\ .$$

24. Boolean Connectives

$$
\begin{array}{lll}
(\mathcal{X},\chi) \models \mathsf{true} \ , & & \\
(\mathcal{X},\chi) \models fml_1 \wedge fml_2 & :\Leftrightarrow & (\mathcal{X},\chi) \models fml_1 \text{ and } (\mathcal{X},\chi) \models fml_2 \\
(\mathcal{X},\chi) \models fml_1 \vee fml_2 & :\Leftrightarrow & (\mathcal{X},\chi) \models fml_1 \text{ or } (\mathcal{X},\chi) \models fml_2 \\
(\mathcal{X},\chi) \models \neg fml & :\Leftrightarrow & \text{not } (\mathcal{X},\chi) \models fml \\
(\mathcal{X},\chi) \models \exists x : fml & :\Leftrightarrow & \text{there is a } d \in \mathcal{U} \text{ with } (\mathcal{X},\chi_x^d) \models fml \ . \\
(\mathcal{X},\chi) \models \forall x : fml & :\Leftrightarrow & \text{for all } d \in \mathcal{U},\ (\mathcal{X},\chi_x^d) \models fml \ .
\end{array}
$$
□

We write $A \rightsquigarrow B$ for the implication which is usual denoted by "$\rightarrow$", since "$\rightarrow$" is used for variable binding in reference expressions.

4.4 Examples and Applications

Using XPath-Logic formulas, conditions on XML documents (and even sets of documents) can be specified, e.g.

- the sum of percentages of ethnic groups in a country is less than 100%:

  $\forall$ C: (//country$\rightarrow$C $\rightsquigarrow$ C[sum{N ; ethnicgroups/@percentage$\rightarrow$N} $\leq$ 100])

- the membership of countries in organizations is represented bidirectionally:

  $\forall$ C,O: (//country$\rightarrow$C[memberships$\rightarrow$O] $\leftrightarrow$
  $\exists$ T: //organization$\rightarrow$O/members[@type$\rightarrow$T and @country$\rightarrow$C])

DTDs and XML Schema. Instead of specifying the document structure by a DTD or an XML Schema instance, a set of closed XPath-Logic expressions can be given, e.g.,

$\forall$ C: (//country$\rightarrow$C $\rightarrow$ C[@car_code and @population and @area and name ...])

5 XPathLog: The Horn Fragment of XPath-Logic

Similar to the case of Datalog which is the quantifier-free Horn fragment of predicate Logic, XPathLog is a Logic Programming language based on XPath-Logic. The language is evaluated wrt. suitable Herbrand structures.

Definition 5.1 (XPathLog Atoms)

- an *XPathLog atom* is either a predicate expression, or a quantifier-free XPath-Logic reference expression[3],
- an XPathLog atom is *definite* if it uses only the child and sibling axes, the descendant axis is allowed only in leftmost position; and the atom does not contain negation, disjunction, function applications, and *proximity position predicates*

  These atoms are allowed in rule heads (see Section 6.3); the excluded features would cause ambiguities what update is intended.
- an *XPathLog literal* is an atom or a negated atom,
- an *XPathLog query* is a list $?\!-\!L_1, \ldots, L_n$ of literals (in general, containing free variables).

[3] recall that in XPath-Logic, quantifiers are allowed in filters.

- an *XPathLog rule* is a formula of the form

$$A_1, \ldots, A_n \leftarrow L_1, \ldots, L_n$$

  where L_i are literals and A_i are definite atoms. $L_1, \ldots, L_n$ is the *body* of the rule, evaluated as a conjunction.

  Note that in contrast to usual Logic Programming, we allow for lists of atoms in the rule head which are interpreted as conjunctions. □

In the remainder of this section, the semantics of queries is investigated. Since the rule bodies are queries, the query semantics is then also used for defining the semantics of XPathLog programs.

5.1 Queries in XPathLog

Similar to Prolog, evaluation of a query `?- L`$_1$`, ..., L`$_n$ results in a set of tuples of variable bindings. As usual, in the Herbrand-style semantics the variables are not bound to elements of an arbitrary universe, but to *ground terms*, i.e., elements of a suitable *Herbrand universe.* The Herbrand-style semantics is defined over an alphabet which is consists of the following sets (not necessarily disjoint):

Literals: $\mathcal{L}$ (literal values, i.e., numbers and strings);

First-order partition:

- $\mathcal{P}$ and $\mathcal{F}$ (first-order partition, predicate symbols and function symbols)

XML partition:

- $\mathcal{N}_\mathcal{A}$, $\mathcal{N}_\mathcal{E}$ (attribute and element names, can e.g. be given by a DTD or XML Schema description),
- $\mathcal{V}$ (node identifiers).

The Herbrand universe is the set generated by $\mathcal{L} \cup \mathcal{N} \cup \mathcal{V}$ and $\mathcal{F}$ (note that with this, predicates involving names and node identifiers are allowed).
For a given alphabet, the *XML Herbrand base* $\mathcal{HB}$ is the set of all *ground atoms* of the form

- (for the XML tree structure:)

$$n[@a \to v] \qquad \text{and} \qquad n[axis@(i) :: e \to w]$$

  where $n \in \mathcal{N}$ is an identifier, $axis$ is an XML axis, $a \in \mathcal{A}$ is an attribute name, $e \in \mathcal{E}$ is an element name (or the constant `text`), i is an integer, and $v \in \mathcal{N} \cup \mathcal{L}$ is an identifier or a literal value, and $w \in \mathcal{N}$ is an identifier, or a literal in case that $e =$ `text` (for representing `PCDATA` contents). The n position is called the "*host*" position, the a/e position is calles the "*property*" position, and the v/w position is called the "*value*" position.

- (for (optional) predicates:)

$$p(u_1, \ldots, u_n)$$

  where $p \in \mathcal{P}$ is a predicate symbol, and $u_i \in \mathcal{N} \cup \mathcal{L}$.

The meaning of the non-predicate atoms is as follows:

- $n[@a \to v]$ denotes that node n has an attribute a s.t.
  - a is defined as `CDATA`, `NMTOKEN`, or `ID` and v is the value of a, or
  - a is defined as `NMTOKENS` and v is one of the values of a, or
  - a is defined as `IDREF`, id is the value of a, and there is a node v whose id is id, or
  - a is defined as `IDREFS`, id is one of the values of a, and there is a node v whose id is id.
- $n[child@(i) :: e \to w]$ denotes that node w is the ith subelement of node n, and that it is of the element type e (note that this differs from the XPath position expression n/child::e[f][i] which addresses the ith element which satisfies n/child::e[f]).

Note that we use the syntax n[e→v] instead of XPath's x[n=v] since the equality is directed one-to-many: there can be many v s.t. x[@a→v] holds (e.g., for multivalued attributes).

Now, usual logic programming frameworks would define the notion of an XML Herbrand Structure based on these atoms which is then used to define a T_P operator. In the XML case, this is not appropriate since the insertion of subelements would require large changes in the structure. Instead, we follow the DOM [DOM98] idea:

Definition 5.2 (DOM Herbrand Structure)
A *DOM Herbrand structure* $\mathcal{H}D$ over a given Herbrand universe consists of

- a set preds($\mathcal{HD}$) of predicate atoms from $\mathcal{HB}$, and
- a (partial) mapping which associates with every $x \in \mathcal{V}$ two lists of ground pairs
  - $A_{\mathcal{H}D}(\mathsf{child}, x) \in (\mathcal{V} \times \mathcal{N}_{\mathcal{E}})^{\mathbb{N}}$ and
  - $A_{\mathcal{H}D}(\mathsf{attribute}, x) \in (\mathcal{V} \times \mathcal{N}_{\mathcal{A}})^{\mathbb{N}}$ □

Again, there is a natural 1:1-correspondence between XML instances and DOM Herbrand structures (cf. Definitions 3.3 and 3.5).

The DOM Herbrand structure contains only the basic facts about the XML tree. Recall that $A_{\mathcal{X}}(\mathsf{child}, x)$ has been defined for the derived axes based on the child axis in Definition 3.5; this definition also applied to DOM Herbrand structures.

Definition 5.3 (Formulas in DOM Herbrand Structures)
The satisfaction relation $\models$ between XPath-Logic formulas and DOM Herbrand models is defined based on

- $\mathcal{A}_{\mathcal{H}}(axis, x)$ (cf. Definition 3.5), and
- $\mathcal{S}$ and $\mathcal{Q}$ (cf. Definition 4.5).

Since $\mathcal{S}$ is completely based on $\mathcal{A}$, only the evaluation of predicates has to be adapted from Definition 4.5:

12. Predicate expressions:

$$\mathcal{Q}_{\mathcal{HD}}(pred(A_1, \ldots, A_n), x, k, n) = \bigcup_{\substack{x_1 \in \mathrm{Res}(\mathcal{S}^a_{\mathcal{X}}(A_1, x, k, n)), \ldots, \\ x_n \in \mathrm{Res}(\mathcal{S}^a_{\mathcal{X}}(A_n, x, k, n)), \\ pred(x_1, \ldots, x_n) \in \mathrm{preds}(\mathcal{HD})}} \mathrm{Bdgs}(\mathcal{S}^a_{\mathcal{X}}(A_1, x_1, k, n)) \bowtie \ldots \bowtie \mathrm{Bdgs}(\mathcal{S}^a_{\mathcal{X}}(A_1, x_n, k, n))$$

Then, $\models$ is defined as in Definition 4.7 for $\mathcal{X}$-structures. □

Note that the semantics of formulas is not directly based on the Herbrand structure (as "usual" Herbrand semantics are), but on the intermediate interpretations $\mathcal{A}_{\mathcal{HD}}$ of the axes in the DOM Herbrand model.

DOM Herbrand structures are now used to define a Herbrand-like structure consisting of ground atoms which contains the complete information about the node lists returned by the axes:

Definition 5.4 (Atomic Representation of DOM Herbrand Structures)
For a DOM Herbrand Structure $\mathcal{HD}$, and $\mathcal{A}_{\mathcal{HD}}(axis, x)$ defined as in Definition 3.5, the *flattened* structure $\mathcal{Fl}(\mathcal{HD})$ is defined as

$$\begin{aligned}\mathcal{Fl}(\mathcal{HD}) = \mathcal{HD} \;&\cup \{n[axis@(i) :: e \rightarrow w] \;\text{ s.t. } \mathcal{A}_{\mathcal{HD}}(axis, x)[i] = (w, name)\}\\ &\cup \{\text{all predicate atoms in } \mathcal{HD}\}\\ &\cup \{\text{all XPath built-in predicates and functions}\}\end{aligned}$$

□

Preview. For an implementation, it is preferable to store the DOM Herbrand structure and to handle $\mathcal{Fl}$ intensionally. □

Example 5.1 (Flat XML Structure)
The XML instance given in Example 2.3 is represented by the following flattened structure:

mondial[child@(42)::country→*ch*].
ch[@car_code→"CH"]. *ch*[@capital→*bern*].
ch[@memberships→*efta*]. *ch*[@memberships→*un*].
ch[child@(1)::name→*ch-name*]. *ch*[child@(2)::city→*bern*]. *ch*[child@(3)::city→...].
ch-name[child@(1)::text →"Switzerland"].
bern[@name→"Bern"]. *bern*[child@(1)::population→*bern-pop*].
bern-pop[child@(1)::text→134393].

mondial[child@(179)::organization→*un*].
un[@id→"org-un"]. *un*[@seat→*newYork*].
un[child@(1)::name→*un-name*]. *un-name*[child@(1)::text→"United Nations"].
un[child@(2)::name→*un-abbrev*]. *un-abbrev*[child@(1)::text→"UN"].
un[child@(3)::members→*un-mem-mem*].
un-mem-mem[@country→*alb*]. *un-mem-mem*[@country→*gr*]. ...
un[child@(4)::members→*un-mem-obs*].
un-mem-obs[@country→*ch*]. *un-mem-obs*[@country→*vat*].
(plus atoms for derived axes)

Definition 5.5
Given a DOM Herbrand Structure $\mathcal{HD}$, the answer to a query

?- $L_1, \ldots, L_n$.

is the set

$$\text{answers}_{\mathcal{HD}}(L_1, \ldots, L_n) \;:=\; \mathcal{Q}_{\mathcal{HD}}(L_1 \text{ and } \ldots \text{and } L_n, root, 0, 0, \emptyset)$$

of tuples of variable bindings (cf. Definitions 4.5 and 3.5(13)). Recall that in the Herbrand-style context, the variables are bound to ground terms. □

For the case that the expression does not use *proximity position predicates* (cf. XPath spec 2.4), a pure Herbrand-style semantics based on flat structures is defined in Section 6.2.

6 XPathLog Programs

In Logic Programming, rules are used for a declarative specification: if the body of a clause evaluates to true for some assignment of its variables, the truth of the head atom for the same variable assignment can be inferred. Depending on the intention, this semantics can be used for (i) check if something is derivable from a given set of facts, or (ii) extending a given set of facts by additional, derived knowledge. The second aspect is followed by *rule-based* approaches, which use rules for the specification of updates to a database or, generally, a situation.

REFERENZEN: Manchester-Active-DB Rock&Roll, GOLOG, ...

Independent from the above, for Logic Programs, two evaluation strategies can be distinguished:

- Top-Down: refute or validate a claim, or compute an answer variable binding for a query.
- Bottom-up: collect all facts which can be derived from a given set of facts.

Note that bottom-up evaluation can also be applied for validating a claim or computing answers: the claim and the answers must be contained in the derived knowledge. Nevertheless, if only a single query is to be answered, the target-driven top-down evaluation which checks only the relevant parts of the input data is in general more efficient. On the other hand, from the database point of view, it is more common to derive all facts first (i.e., to update the database), and then to answer questions.

In this work, we investigate the bottom-up strategy, regarding XPathLog as an *update language for XML databases.*

The body of an XPathLog rule is a set of XPathLog expressions. The evaluation of the body wrt. a given structure yields variable bindings which are propagated to the rule head where facts are added to the model.

6.1 Bottom-up Evaluation: Positive Programs

Positive (i.e., the rules contain only positive literals; in our case also filters may only contain positive expressions) XPathLog programs are evaluated bottom-up by a T_P-like operator, providing a minimal model semantics. We first give the definition of T_P as defined for Prolog and Datalog:

Definition 6.1 (Deductive Fixpoint Semantics: The T_P-Operator)
For a Datalog program P and an H-structure $\mathcal{H}$,

$$\begin{aligned}
T_P(\mathcal{H}) &:= \mathcal{H} \cup \{h \mid (h \leftarrow b_1, \ldots, b_n) \text{ is a ground instance of some rule of } P \\
&\qquad\qquad \text{and } b_i \in \mathcal{H} \text{ for all } i = 1, \ldots, n\} \ , \\
T_P^0(\mathcal{H}) &:= \mathcal{H} \ , \\
T_P^{i+1}(\mathcal{H}) &:= T_P(T_P^i(\mathcal{H})) \ , \\
T_P^\omega(\mathcal{H}) &:= \begin{cases} \lim_{i\to\infty} T_P^i(\mathcal{H}) & \text{if the sequence } T_P^0(\mathcal{H}), T_P^1(\mathcal{H}), \ldots \text{ converges,} \\ \perp & \text{otherwise.} \end{cases}
\end{aligned}$$

The application of a Datalog program P to a given database $\mathcal{D}$ is defined as

$$P(\mathcal{D}) = T_P^\omega(\mathcal{D})$$

□

Preview. This definition is adjusted in Definition 6.5 for XPathLog programs after explaining the semantics of insertions and updates. □

6.2 Atomization

In this section, we present an alternative, but *nearly* (it is equivalent for the cases where expression does not contain negation, disjunction, quantifiers, and *proximity position predicates*; cf. Remark 6.2) equivalent semantics of queries which provides the base for the semantics of reference expressions in rule heads.

The semantics wrt. the DOM Herbrand structure defined in Definition 5.3 is not purely Herbrand-like since it reduces to the evaluation of the X-Structure induced by the DOM Herbrand structure.

Instead, a semantics is defined which resolves reference expressions in their constituting atomic steps which are directly represented in the flat Herbrand structure (cf. Definition 5.4).

6.2.1 Positive Formulas

The function

$$\mathsf{atomize} : \text{XPathLogExpr} \rightarrow 2^{\text{XPathLogAtom}}$$

resolves a definite XPathLog atom into atoms of the form $\mathsf{n}[axis{::}\mathsf{e}{\rightarrow}\mathsf{w}]$. atomize is defined by structural induction corresponding to the induction steps when defining $\mathcal{S}_{\mathcal{X}}$:

Definition 6.2 (Atomization of Formulas)
The mapping

$$\mathsf{atomize} : \text{XPathLogExpr} \rightarrow 2^{\text{XPathLogAtom}}$$

resolves a definite XPath-Logic atom into a set of atomic expressions. In the following, *ref* stands for a reference (or a variable over references), and *name* for a name (or a variable over names).

- the entry case:

$$\mathsf{atomize}(/remainder) = \mathsf{atomize}(\mathsf{root}/remainder)$$

- Paths are resolved into steps and filters are isolated:

$$\begin{array}{l}\mathsf{atomize}(ref/axis :: name \rightarrow var[filter]) = \{ref[axis :: name \rightarrow var]\} \cup \mathsf{atomize}(var[filter])\ ,\\ \mathsf{atomize}(ref/axis :: name \rightarrow var[filter]\ /remainder) =\\ \qquad \mathsf{atomize}(ref[axis :: name \rightarrow var]) \cup \mathsf{atomize}(var[filter]) \cup \mathsf{atomize}(var/remainder)\ ,\\ \mathsf{atomize}(ref/axis :: name[filter]/remainder) =\\ \qquad \mathsf{atomize}(ref[axis :: name \rightarrow \_X]) \cup \mathsf{atomize}(var[filter]) \cup \mathsf{atomize}(\_X/remainder)\ .\\ \qquad \text{where } \_\mathsf{X} \text{ is a new variable}\end{array}$$

- text nodes:

$$\begin{array}{l}\mathsf{atomize}(ref/axis :: text() \rightarrow var) = \{ref[axis :: \mathsf{text}() \rightarrow var]\}\\ \mathsf{atomize}(ref/axis :: text()) = \{ref[axis :: \mathsf{text}() \rightarrow \_X]\}\\ \qquad \text{where } \_\mathsf{X} \text{ is a new variable}\end{array}$$

- function applications:

$$\begin{array}{l}\mathsf{atomize}(ref!f(args)) = \{ref[f(args) \rightarrow \_X]\}\\ \qquad \text{where } \_\mathsf{X} \text{ is a new variable}\\ \mathsf{atomize}(ref!f(args) \rightarrow var) = \{ref[f(args) \rightarrow var]\}\end{array}$$

- Conjunctions in filters are separated:

$$\mathsf{atomize}(\mathit{ref}[\mathit{pred}_1 \wedge \ldots \wedge \mathit{pred}_n]) = \mathsf{atomize}(\mathit{ref}[\mathit{pred}_1], \ldots, \mathit{ref}[\mathit{pred}_n])$$

- Predicates in filters (including the case $\mathit{ref}[\mathit{expr}]$; recall that quantifiers, negation, and disjunction are not allowed, in definite XPath-Logic atoms):

$$\begin{aligned}\mathsf{atomize}(\mathit{ref}[\mathit{pred}(\mathit{expr}_1, \ldots, \mathit{expr}_n)]) = {} & \mathsf{atomize}(\mathit{equality}(\mathit{ref}, \mathit{expr}_1, \_X_1)) \cup \ldots \\ & \mathsf{atomize}(\mathit{equality}(\mathit{ref}, \mathit{expr}_n, \_X_n)) \cup \\ & \{\mathit{pred}(\_X_1, \ldots, \_X_n)\}\end{aligned}$$

  where $\mathit{equality}(\mathit{ref}, \mathit{expr}, X)$ is defined as follows (if expr_i is a constant, it is not replaced by a variable):

  – $\mathit{expr} \to X$ if expr is of the form $//\mathit{remainder}$, or

  – $\mathit{ref}!\mathit{function}() \to X$ if expr is of the form $\mathit{function}()$, or

  – $\mathit{ref}/\mathit{expr} \to X$ if expr is of the form $\mathit{axis} :: \mathit{name}\ \mathit{remainder}$. □

For every XPathLog expression expr, $\mathsf{atomize}(\mathit{expr})$ contains only atoms of the form $\mathsf{node}[\mathsf{axis}::\mathsf{name} = \mathsf{result}]$ and atomic predicates over variables and constants.

Example 6.1 (Atomization)

```
?- //organization→O[name/text()→ON;
                    @seat = members/@country[name/text()→CN]/@capital].
```

is atomized into

```
?- root[descendant::organization→O],   O[name→_ON], _ON[text() →ON],
   O[@seat→_S], O[members→_M], _M[@country→_C], _C[@country→_Cap], _S = _Cap,
   _C[child::name→_CN], _CN[text() →CN].
```

Definition 6.3 (Full-Herbrand-Semantics of XPathLogic reference expressions)
The $\models_{at}$ relation using atomized queries is defined as follows: For an DOM Herbrand structure $\mathcal{HD}$, a variable assignment χ, and a ground atom a which is a predicate atom or of the form

$$n[axis :: e \to w] \qquad \text{or} \qquad n[@a \to w] \ ,$$

$$(\mathcal{HD}, \chi) \models_{at} \mathit{atom} \ :\Leftrightarrow\ \chi(\mathit{atom}) \in \mathcal{F\!l}(\mathcal{HD}) \ .$$

For compound formulas, $\models_{at}$ is defined as usual by structural induction. □

Lemma 6.1 (Correctness of atomize)
For every DOM Herbrand structure $\mathcal{HD}$ and every definite XPath-Logic atom expr,

- for every variable assignment χ of free(expr) s.t. $(\mathcal{HD}, \chi) \models \mathit{expr}$, there exists a variable assignment $\chi' \supseteq \chi$ of free($\mathsf{atomize}(\mathit{expr})$) s.t. $(\mathcal{F\!l}(\mathcal{HD}), \chi') \models_{at} \mathsf{atomize}(\mathit{expr})$, and

- for every variable assignment χ' of free($\mathsf{atomize}(\mathit{expr})$) s.t. $(\mathcal{F\!l}(\mathcal{HD}), \chi') \models_{at} \mathsf{atomize}(\mathit{expr})$, $(\mathcal{HD}, \chi'|_{\mathrm{free}(\mathit{expr})}) \models \mathit{expr}$. □

Theorem 6.2 (Equivalence)
The above semantics is equivalent to the one presented in Definition 5.3 for all definite XPathLog atoms: For every DOM Herbrand structure $\mathcal{HD}$,

$$\mathrm{answers}_{\mathcal{HD}}(\mathit{query}) = \mathrm{answers}_{\mathcal{HD}}(\mathsf{atomize}(\mathit{query}))$$

(recall that don't care variables are not considered in the answer.) □

6.2.2 Negation and Disjunction

Negation and Disjunction require nested atomization which mirrors the evaluation of complex subexpression in the definition of $\mathcal{S}$. On the other hand, these constructs are not allowed in rule heads, so the section may be skipped by the reader.

Negation: For negation over complex expressions, the evaluation strategies described for PathLog [FLU94] can be used: The body of the negated subquery is atomized and evaluated. Then the result set is projected on the variable which are relevant for merging with the outer expression. Then it is joined as a virtual, intensional predicate with the outer atoms. Here, the atomization is given by

$$\begin{aligned}\mathsf{atomize}(\mathit{ref}[\neg \mathit{pred}(\mathit{expr}_1,\ldots,\mathit{expr}_n)]) = \neg V \in \Pi_V(&\mathsf{atomize}(\mathit{equality}(\mathit{ref},\mathit{expr}_1,\_X_1)) \cup \ldots\\ &\mathsf{atomize}(\mathit{equality}(\mathit{ref},\mathit{expr}_n,\_X_n)) \cup\\ &\{\mathit{pred}(\_X_1,\ldots,\_X_n)\})\end{aligned}$$

where V is the set of free variables in $\mathit{pred}(\mathit{expr}_1,\ldots,\mathit{expr}_n)$.

Example 6.2

The following query yields all organizations which are not seated in a city in some member country:

?- //organization[name/text()→N and not @seat = members/@country//city].

The answer includes those where no seat is known, since the filter predicate says "not those where (i) the seat attribute is defined, and (ii) references a city in a country which is a member". Thus, a subquery must be evaluated separately. The atomization of the query is

?- root[descendant::organization→O], O[name→_ON], _ON/text() = ON,
not O ∈ {O: O[@seat→_S], O[members→_M], _M[@country→_C], _C[city→_Cty], _S = _Cty}

Disjunction: To be well-defined, all variables which get their binding inside the disjunction and are used outside, must occur in both parts of the disjunction.

$$\begin{aligned}&\mathsf{atomize}(\mathit{ref}[\mathit{pred}_1 \text{ or } \mathit{pred}_2]) =\\ &\qquad V \in_{subs} (\Pi_{V_1}(\mathsf{atomize}(\mathit{ref}[\mathit{pred}_1])) \cup \Pi_{V_2}(\mathsf{atomize}(\mathit{ref}[\mathit{pred}_2])))\end{aligned}$$

where $V = \mathrm{free}(\mathit{pred}_1) \cap \mathrm{free}(\mathit{pred}_2)$, $V_i = \mathrm{free}(\mathit{pred}_i)$ and $\chi \in_{subs} \xi$ denotes that χ qualifies if it is subsumed by some $\chi' \in \xi$.

6.3 Left Hand Side

Since the right hand side of rules has already been handled in Section 5.1, the semantics of the left hand side is now investigated based on the atomization of expressions.

Using *logical expressions* for specifying an update is perhaps the most important difference to approaches like XSLT, XML-QL, or Quilt where the structure to be *generated* is always specified by XML patterns (this implies that these languages do not allow for updating existing nodes – e.g., adding children or attributes –, but only for generating complete nodes). In contrast, in XPathLog, existing nodes are communicated via variables to the head, where they are *modified* when appearing at host position of atoms.

The head of an XPathLog rule is a set of definite XPathLog atoms. When used in the head, the / and // operators and the [. . .] construct specify which properties should be added or updated (thus, [...] does not act as a filter, but as a *constructor*). Additionally, when manipulating subelements, copying (for data transformation) or linking (for data integration) can be specified.

Note that the language does *not* allow to delete or replace existing elements or attributes – modifications are always monotonic in the sense that existing "things" remain (although, children may be inserted between already existing children which makes the evaluation of proximity position predicates non-monotonic; similar to the evaluation of aggregration operators in related approaches).

Modification of Elements. When using the child or attribute axis for updates, the host of the expression gives the element to be updated or extended; when a sibling axis is used, effectively the parent of the host is extended with a new subelement.

Generation or Extension of Attributes. A ground instantiated atom of the form $n[@a{\to}v]$ specifies that the attribute @a of the node n should be set or extended with v. If v is not a literal value but a node, a reference to v is stored.

Example 6.3 (Attributes)
We add the data code to Switzerland, and make it a member of the European Union:

```
C[@datacode→"ch"], C[@memberships→O] :-
    //country→C[@car_code="CH"], //organization→O[abbrev/text()→"EU"].
```

results in

```
<country datacode="ch" car_code="CH" industry="machinery chemicals watches"
        memberships="org-efta org-un org-eu ..."> ... </country>
```

We discuss insertion of new subelements below, after showing how to create elements.

Creation of Elements. Elements can either be created as *free* elements by atoms of the form //name[...] (meaning "some element of type name" – in the rule head, this is interpreted to create an element which is not a subelement of any other element), or as subelements.

Example 6.4
We create a new (free) country element with some properties:

```
//country[@car_code→"BAV" and @capital→X and city→X and city→Y] :-
    //city→X[name/text()="Munich"],   //city→Y[name/text()="Nurnberg"].
```

Note that the two city elements are linked *as subelements. This operation has no equivalent in the "classical" XML model: these elements are now children of* two *country elements.* Copying *elements is described below. Thus, changing the elements effects both trees. By* linking *elements, it is possible to introduce cycles in the document hierarchy.*

Insertion of Subelements and Attributes. Already existing elements can be assigned as subelements to existing elements by using filter syntax in the rule head: A ground instantiated atom $n[child :: s \to m]$ makes m a subelement of type s of n. In this case, in the Herbrand representation, m is *linked* as n/s somewhere (no position specified). In contrast, $n[child :: s{\twoheadrightarrow}m]$ will deep-copy m.

Example 6.5
The following two rules are equivalent to the above ones:

//country[@car_code→"BAV"].
C[@capital→X and city→X and city→Y] :-
  //city→X[name/text()→"Munich"], //city→Y[name/text()→"Nurnberg"],
  //country→C[@car_code="BAV"].

Here, the first rule creates a free element, whereas the second rule uses the variable binding of C for inserting subelements and attributes.

In the above case, the position of the new subelement is not specified. If the atom is of the form n[child@(i)::$s \rightarrow m$] or n[following-sibling@(j)::$s \rightarrow m$], this means that the new element to be inserted should be made the ith subelement of n or jth sibling of n.

Once-for-each-Binding. In contrast to classial Logic Programming where it does not matter if a fact is "inserted" into the database several times (e.g., once in every round, conforming with the formal definition of the T_P operator), here subelements must be created exactly once for each instantiation of a rule. We consider this with defining a revised T_P-operator in Definition 6.5.

Generation of Elements by Path Expressions. Additionally, subelements can be created by *path expressions* in the rule head which create nested elements which satisfy the given path expression. The atomization introduces local variables which occur *only in the head* of the rule, i.e., the result is not a valid Logic Programming rule. Here, we follow again the semantics of PathLog [FLU94] which is implemented in [LHL+98] for object creation. After the atomization, the resulting atoms are processed in an ordering such that the local variables are bound to the nodes/objects which are generated. Thus, the rules are in fact safe.

Example 6.6
Bavaria gets a (PCDATA) subelement name:

C/name[/text()→"Bavaria"] :- //country→C[@car_code="BAV"].

Here, the atomized version of the rule is

C[name→_N], _N[/text()→"Bavaria"] :- root[descendant::country→C], C[@car_code="BAV"].

The body produces the variable binding C/bavaria. *When the head is evaluated, first, the fact* bavaria[child::name→x_1] *is inserted, adding an (empty)* name *subelement* x_1 *to* bavaria *and binding the local variable _N to* x_1. *Then, the second atom is evaluated, generating the text contents to* x_1.

Using Navigation Variables for Restructuring For data restructuring and integration, the intuitiveness and declarativeness of a language gains much from variables ranging not only over data, but also over schema concepts (classically, relation and columns, as, e.g., in SchemaSQL [LSS96]). Such features have already been used for HTML-based Web data integration with F-Logic [KLW95, LHL+98].

Extending the XPath wildcard concept, XPathLog also allows to have variables at name position. Thus, it allows for schema querying, and also for generating new structures dependent on the data contents of the original one. Here, the we define the semantics of XPathLog to cast strings into names when a variable is bound to string contents in the body, and occurs at name position in the head:

Example 6.7
From the elements

<water type="river" name="Mississippi"> ... </water>
<water type="sea" name="North Sea"> ... </water> ,

the rule //T[@name→N] :- //water[@type→T and @name→N]
creates <river name="Mississippi"/> and <sea name="North Sea"/> .
Attributes and contents are then transformed by separate rules which use @name *for identification. Properties are copied by using variables at element name and attribute name position:*

X[@A→V] :- //water[@type→T and @name→N and @A→V], //T→X[@name→N].
X[S→V] :- //water[@type→T and @name→N and S→V], //T→X[@name→N].

6.4 Semantics of Positive XPathLog Programs

For implementing the once-for-each-binding approach, the T_P operator has to be extended with bookkeeping about the ground instances of inserted rule heads. Additionally, the insertion of subelements adds some nonmonotonicity: adding an atom n[child@(i)::e→v] (or n[sibling@(i)::e→v]) adds a new subelement at the ith position (or as ith sibling), making the original ith child/sibling the i+1th etc. In case of multiple extensions to the same element, the indexes are evaluated wrt. the original structure. If several insertions effect the same position, they may be executed in arbitrary ordering.

Remark 6.1
Note that this does not *exchange* the i-th child/sibling (which would possibly produce dangling references). □

Definition 6.4 (Extension of DOM Herbrand Structures)
Given a DOM Herbrand structure $\mathcal{HD}$ and a set $\mathcal{I} \subset \mathcal{HB}$ of atoms (to be inserted), the new DOM Herbrand structure $\mathcal{HD}' = \mathcal{HD} \prec \mathcal{I}$ is obtained from $\mathcal{HD}$ as follows:

- initialize
$$\begin{aligned} &\mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, x)) := \mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x))\ , \\ &\mathcal{A}_{\mathcal{HD}'}(\mathsf{attribute}, x)) := \mathcal{A}_{\mathcal{HD}}(\mathsf{attribute}, x))\ , \\ &\mathrm{preds}(\mathcal{HD}') := \mathrm{preds}(\mathcal{HD}) \cup \{p \mid p \in \mathcal{I} \text{ is a predicate atom}\} \end{aligned}$$
for all node identifiers x.

- for all elements of $\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)$,
$$\alpha(\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)[i]) := \mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, x)[i]$$
(α maps the indexing from the old list to the new one),

- for all atoms p[child@(i)::e→y] $\in \mathcal{I}$, insert y into $\mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, p)$ immediately after $\alpha(\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, p)[i])$.

- for all atoms x[following-sibling@(i)::e→y] $\in \mathcal{I}$ s.t. p[child@(j)::e→x] $\in \mathcal{H}$, insert y into $\mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, p)$ immediately after $\alpha(\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)[j{+}i])$.

- for all atoms x[preceding-sibling@(i)::e→y] $\in \mathcal{I}$ s.t. p[child@(j)::e→x] $\in \mathcal{H}$, insert y into $\mathcal{A}_{\mathcal{HD}'}(\mathsf{child}, p)$ immediately after $\alpha(\mathcal{A}_{\mathcal{HD}}(\mathsf{child}, x)[j{-}i])$.

- for all atoms p[@a→y] $\in \mathcal{I}$, append y to $\mathcal{A}_{\mathcal{HD}'}(\mathsf{attribute}, p)$. □

Definition 6.5 ($\mathcal{T}\!X_P$-Operator for XPath-Logic Programs)
The $\mathcal{T}\!X$-operator works on pairs $(\mathcal{HD}, Dic)$ where $\mathcal{HD}$ is DOM Herbrand structure, and Dic is a dictionary which associates to every rule a set ξ of bindings which have been instantiated in the current iteration.

$$\begin{aligned}(\mathcal{HD}, Dic) + (\{(r_1,\chi_1),\ldots,(r_n,\chi_n)\}) &:= (\mathcal{HD} \prec \{\chi_i(head(r_i)) \mid 1 \leq i \leq n\},\\ &\qquad Dic.insert(\{(r_1,\chi_1),\ldots,(r_n,\chi_n)\}) \ ,\\ (\mathcal{HD}, Dic)\downarrow_1 &:= \mathcal{HD} \ .\end{aligned}$$

For an XPathLog program P and a DOM Herbrand structure $\mathcal{HD}$,

$$\begin{aligned}\mathcal{T}\!X_P(\mathcal{HD},B)) &:= (\mathcal{HD},B) + \{(\chi(h),(r,\chi)) \mid r = (h \leftarrow b) \in P \text{ and } \mathcal{HD} \models \chi(b), \text{ and } (r,\chi) \notin B\} \ ,\\ \mathcal{T}\!X_P^0(\mathcal{HD}) &:= (\mathcal{HD},\emptyset) \ ,\\ \mathcal{T}\!X_P^{i+1}(\mathcal{HD},B) &:= \mathcal{T}\!X_P(TX_P^i(\mathcal{HD})) \ ,\\ \mathcal{T}\!X_P^{\omega}(\mathcal{HD},B) &:= \begin{cases} \lim_{i\to\infty} \mathcal{T}\!X_P^i(\mathcal{HD}) & \text{if the sequence } \mathcal{T}\!X_P^0(\mathcal{HD}), \mathcal{T}\!X_P^1(\mathcal{HD}), \ldots \text{ converges,}\\ \perp & \text{otherwise.}\end{cases}\end{aligned}$$

□

The input to an XPathLog program is an XML document, respectively, the corresponding canonical XML Herbrand structure. (recall that only atoms for the attribute axis and the child axis are generated which then unambiguously represent the tree structure.)

Definition 6.6 (Semantics of Positive XPathLog Programs)
The application of a positive XPathLog program P wrt. a given XML document $\mathcal{D}$ is defined as

$$P(\mathcal{D}) = (\mathcal{T}\!X_P^{\omega}(\mathcal{HD},\emptyset))\downarrow_1 \ .$$

where $\mathcal{HD}$ is the canonical DOM Herbrand structure to $\mathcal{D}$. □

Remark 6.2
Note that for pure Datalog programs P, evaluation wrt. $\mathcal{T}\!X_P$ does not change the semantics, i.e.,

$$\mathcal{T}\!X_P^{\omega}(\emptyset) = T_P^{\omega}(\emptyset) \ .$$

□

An XPathLog program is a declarative specification how to create a set of XML documents from one or more input documents. The semantics of XPathLog programs is defined by bottom-up evaluation based on a T_P operator similar to Datalog. Thus, the semantics coincides with the usual understanding of a stepwise process. Additionally, an intuitive meaning of negation and other nonmonotonic features can easily be expressed by stratification.

6.5 Semantics of General XPathLog Programs

For logic programs which use negation (or similar nonmonotonic features, such as aggregation), there is no *minimal model semantics.* Instead, their semantics is defined wrt. perfect models, well-founded models, or stable models.

For practical use - especially when considering bottom-up evaluation – the notion of *perfect models* and *stratification* [Prz88] provides a solution to the problems raised by negation:

> "... negation always refers to an already known relation. More specifically, first some relations should be defined (perhaps recursively) in terms of themselves *without* the

use of negation. Next, some new relations can be defined in terms of themselves without the use of negation and in terms of the previous ones possibly with the use of negation. This process can be iterated."

A Datalog program P is stratified, if it can be partitioned into *strata* $P_1, \ldots, P_n$ such that for $i = 1, \ldots, n$,

- If a relation symbol R occurs *positively* in the body of a rule in P_i, then all rules containing R in their heads are contained in $\bigcup_{k \leq i} P_j$.
- If a relation symbol R occurs *negatively* in the body of a rule in P_i, then all rules containing R in their heads are contained in $\bigcup_{k<i} P_j$.

Then, P is *stratified via* $P_1 \cup \ldots \cup P_n$, and every P_i is a *stratum* of P.
So far, the definition for Datalog. Note that not all Datalog programs are stratifiable – it is simply a syntactical criterion which is sometimes satisfied.
As long as variables are not allowed at the property position, a reasonable notion of stratification can be defined based on the names occurring at property position. Nevertheless, (i) there are many programs which are not syntactically stratifiable, and (ii) variables at the property position prove to be very useful for data integration (cf. Example 6.7).

Since the intention of XPathLog programs is in general to implement a stepwise process (which is also mirrored by the bottom-up evaluation strategy), often there is a natural, *user-defined* stratification. User defined stratification is supported in the LoPiX system [LoP].

7 Metadata: Classes and Signature Information

Classes (in general organized with a class hierarchy and equipped with some inheritance concept) provide an important means for *data modeling*. Additionally, for data integration, reasoning in the meta level, i.e., signatures (possibly augmented by ontologies which provide meta information about signatures) is a crucial requirement.

The pure XML data model does not directly include a notion of classes and class membership. For XML, signature information is usually given either by DTDs or by XML Schema instances.

DTDs. DTDs are a heritage from the SGML area, tailored to describe the document structure. Here, the notion of element *types* is present for specifying the structure of documents, but not as a *modeling concept*. Some typical schema issues from the database area are not covered:

- Datatypes: the only literal types are `CDATA/PCDATA` and `NMTOKEN(S)`.
- Cardinalities: instead of min/max cardinalities as, e.g., in the ER model or in UML, only optional/required and scalar/multiple (by iteration) can be specified.

  A significant weakness is that it is clumsy to specify that an element must contain some subelements in *arbitrary* order:

Example 7.1

A country *element must have one* area *subelement, one* population *subelement and one* gdp *subelement:*

```
<!ELEMENT country ((area,population,gdp)|(area,gdp,population)|
                   (population,gdp,area)|(population,area,gdp)|
                   (gdp,area,population)|(gdp,population,area))
```

7.1 DTDs vs. XML Schema

In practice, since the DTDs are the "original" way to describe structural information on XML documents, most XML tools support DTDs, and thus, most XML instances come with DTDs.

An important argument for DTDs is that standard validating parsers use them; although also XML Schema aware parsers/validators have recently been presented (see `http://www.w3.org/XML/Schema`). On the other hand, DTDs give only incomplete information on what usually is understood as schema information in databases: especially, the support for cardinalities is only restricted.

With usual XML programming languages, there is only implicit access to the DTD - it does not fit into the XML data model (commercial tools such as Tamino and eXcelon provide access to the schema description of stored documents). Nevertheless, as stated above this implicit access is crucial for validating parsers, and for parsing documents unambiguously (e.g., distinguishing `NMTOKENS` attributes from `CDATA` ones).

Remark 7.1
The LoPiX XML parser also accesses the DTD for resolving attributes; especially, `IDREF` attributes are automatically resolved. □

In contrast, XML Schema documents are valid XML documents according to the XML Schema DTD which specifies the structure of XML Schema instances. Thus, applications which are aware of the special semantics of these tags can access them using standard languages such as XML-QL, or Quilt/XQuery, similar to the *Data Dictionary* in relational database systems.

7.2 Signatures in XPath-Logic

XPath-Logic seamlessly integrates signature information via *signature atoms*:

Definition 7.1 (XPath-Logic Signature Atoms)

- *XPath-Logic signature atoms* are of the form

  $$elementtype[subelement{\Rightarrow}elementtype_2] \quad \text{or} \quad elementtype[\texttt{@}attribute{\Rightarrow}type]$$

  where $elementtype \in \mathcal{N}_E$ is an elementtype, where $subelement \in \mathcal{N}_E \cup \mathcal{N}_A$ is an elementtype or attribute name, and $elementtype_2$ is an elementtype or a literal type (number, string).

  $$elementtype[subelement{\Rightarrow}elementtype_2]$$

  denotes that elements of *elementtype* may have subelements with tag *subelement* which are of $elementtype_2$ (see remark below). Analogously,

  $$elementtype[\texttt{@}attribute{\Rightarrow}type]$$

  denotes that elements of *elementtype* may have an attribute *attribute* which have value(s) of type *type* (see remark below).

- Multiple Specifications: If two signature specifications

$elementtype[subelement{\Rightarrow}elementtype_2]$ and
$elementtype[subelement{\Rightarrow}elementtype_3]$

are given, this means that *subelement* subelements of elements of type *elementtype* belong (conjunctively) to both elementtypes/classes $elementtype_2$ and $elementtype_3$ (which can either be overlapping classes, or one a subclass of the other). □

Remark 7.2
The above signatures mirror the differences between the XPath-Logic data model, the basic XML tree model, and the data model used in XML Schema:

- Subelements: When using s DTD, $subelement = elementtype_2$ since elements and element types are not cleanly destinguished. When using an XML Schema metadata description elements of original type $elementtype_2$ may occur als subelements with tag $subelement \neq elementtype_2$. The same may happen when executing updates and linking an element of original type $elementtype_2$ as $subelement \neq elementtype_2$-subelements e.g., for

  result[metropolis →C] :- cia/city→C[population > 1000000].

  (regarding population as an annotated literal) the result is specified by the signature atom

  result[metropolis ⇒city].

- Attributes: for non-reference attributes, the result type is always literal or a subclass of it (number, string).
  For reference attributes (which are resolved automatically when parsing an XML document, the result type is always *object* or a subclass of it (i.e., elementtypes or user-defined classes). □

8 Related Work and Conclusion

Other approaches to semi-structured data, especially focusing on semi-structured data as databases (in contrast to documents) are OEM [GMPQ+97], YATL [CDSS99], and F-Logic [KLW95, LHL+98], using "proprietary" semi-structured data models of the respective languages (in pre-XML times). With these, also Logic Programming style languages have been used for manipulating and integrating semi-structured data(bases).

Prolog-style (i.e., with cut and fail semantics), and Constraint Logic Programming approaches for programming Web access and generation of HTML documents have been presented in [HC96, Tar99]. Here, in contrast to the intuitive bottom-up semantics, the operational semantics of Prolog evaluation is exploited in combination with imperative programming languages.

Several XML querying and transformation languages have already been mentioned in the introduction. XQL is a restricted pure querying language based on XPath. XPath also provides the basis for the transformation language XSLT, which is – similar to XPathLog – rule-based, but following a functional idea. XML-QL [DFF+99] is another querying/transformation language based on matching of XML-style patterns. QUILT [CRF00] is a recent proposal for a comprehensive XML query language which also directly produces XML output. The above languages can be classified as declarative and in some sense rule-based (in XSL(T), following the functional paradigm). Only XSL(T) supports the notions of "stratification" and views by its nodeset concept. None of these languages allows for updating a database. There are also several XML database-like systems, e.g., TAMINO [Sof] and eXcelon [eXc] which provide comfortable user-interfaces for querying and design of XML databases; the latter extends XSLT with an update language XUL.

Conclusion. To our knowledge, XPathLog is the first declarative, native XML language which allows for view definition and updates. XPathLog is completely XPath-based, i.e., both the rule bodies and the rule heads use an extended XPath syntax, thereby defining an update semantics for XPath expressions. The close relationship with XPath ensures that its declarative semantics is well understood from the XML perspective. Especially the nature of rule based bottom-up programming is easily understandable for XSLT practitioners, providing even more functionality. The Logic Programming background provides a strong theoretical foundation of the language concept.

XPathLog can be used both for querying XML documents, and for restructuring and integration of XML sources.

Implementation. Large parts of XPathLog have already been implemented in the LoPiX system [LoP], based on the evaluation component of the F-Logic system FLORID [LHL+98]. Currently, the combination with a native XPath/DOM implementation is investigated.

References

[CDSS99] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators need Data Conversion. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1999.

[CRF00] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *WebDB 2000*, pp. 53–62, 2000.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. In *8th. WWW Conference.* W3C, 1999. World Wide Web Consortium Technical Report, NOTE-xml-ql-19980819, `www.w3.org/TR/NOTE-xml-ql`.

[DOM98] Document Object Model (DOM). `http://www.w3.org/DOM/`, 1998.

[eXc] eXcelon Corporation. XML Application Development using eXcelon. `http://www.odi.com/`.

[FLU94] J. Frohn, G. Lausen, and H. Uphoff. Access to Objects by Path Expressions and Rules. In *Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 273–284, 1994.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2), 1997.

[HC96] M. Hermenegildo and D. Cabeza. Internet and WWW Programming using Computational Logic Systems [PiLLoW]. In *Workshop on Logic Programming and the Internet*, 1996. `www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html`.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[LHL+98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. *Information Systems*, 23(8):589–612, 1998.

[LoP] The LoPiX System. `http://www.informatik.uni-freiburg.de/~may/lopix/`.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL – A Language for Interoperability in Relational Multi-Database Systems. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 239–250, 1996.

[Mon] The Mondial Database. `http://www.informatik.uni-freiburg.de/~may/Mondial/`.

[Prz88] T. C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 191–216. Morgan Kaufmann, 1988.

[Rob99] J. Robie. XQL (XML Query Language). `http://www.metalab.unc.edu/xql/xql-proposal.html`, 1999.

[Sof] Software AG. Tamino – An Internet Database System. `http://www.tamino.com/`.

[Tar99] P. Tarau. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java. In *PAAM*, 1999. `www.binnetcorp.com/Jinni`.

[Wad99a] P. Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies*, 1999. `http://www.cs.bell-labs.com/who/wadler/topics/xml.html`.

[Wad99b] P. Wadler. Two semantics for XPath. 1999. `http://www.cs.bell-labs.com/who/wadler/topics/xml.html`.

[XPa99] XML Path Language (XPath). `http://www.w3.org/TR/xpath`, 1999.

[XSL99] XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`, 1999.