TLP **4** (3): 239–287, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S147106840300187X Printed in the United Kingdom

XPath-logic and XPathLog: A logic-programming style XML data manipulation language

WOLFGANG MAY
Institut für Informatik, Universität Göttingen, Göttingen, Germany
(*e-mail:* `may@informatik.uni-goettingen.de`)

Abstract

We define XPathLog as a Datalog-style extension of XPath. XPathLog provides a clear, declarative language for querying and manipulating XML whose perspectives are especially in XML data integration. In our characterization, the formal semantics is defined wrt. an *edge-labeled graph-based* model, which covers the XML data model. We give a complete, logic-based characterization of XML data and the main language concept for XML, XPath. XPath-Logic extends the XPath language with variable bindings and embeds it into first-order logic. XPathLog is then the Horn fragment of XPath-Logic, providing a Datalog-style, rule-based language for querying and manipulating XML data. The model-theoretic semantics of XPath-Logic serves as the base of XPathLog as a logic-programming language, whereas also an equivalent answer-set semantics for evaluating XPathLog queries is given. 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.

KEYWORDS: XML, XPath, Logic Programming, Information Integration.

1 Introduction

Logic-based languages have proven useful in many areas since they allow for small, declarative, and extendible programs. For the database area, Datalog has been investigated for querying and rule-based data manipulation. Extending the Datalog idea, more complex logic-based frameworks like F-Logic (Kifer and Lausen, 1989; Kifer *et al.*, 1995), or the languages of the Tsimmis project (Garcia-Molina *et al.*, 1997; Abiteboul *et al.*, 1997) have been successfully applied for knowledge representation and data integration. The experiences with a powerful language like F-Logic were the motivation to have a similar "native" language for the XML world that is much simpler than F-Logic, and that is based on the standard XPath language. As a result, we present XPathLog as an XPath-based Datalog-style language for querying and manipulating XML data. By extending XPath with variable bindings and providing a constructive semantics for XPath in rule reads, a declarative XML data manipulation language is obtained. Since both XPath and rule-based programming by using variable bindings are well-known, intuitive concepts, the "effect" of the language is easy to understand on an intuitive basis. Additionally, the well-known logic programming semantics provide concise *global* semantics of such programs which coincide with the intuitive ideas. Queries and rules for manipulating and restructuring the internal XML database can be expressed much easier than e.g. in XQuery (XQuery, 2001) (where update functionality is still in a prototypical state).

Semistructured Data and XML. 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. The XML data model applies both to *documents* and to *databases*: The SGML language was originally defined for *documents* in the publishing area. On the other hand, the interest in research on *semistructured* data in the 1990s[1] (e.g. F-Logic (Kifer and Lausen, 1989; Kifer *et al.*, 1995), GraphLog (Consens and Mendelzon, 1990), UnQL (Buneman *et al.*, 1996; Buneman *et al.*, 2000), Tsimmis (Garcia-Molina *et al.*, 1997; Abiteboul *et al.*, 1997) with the *OEM* data model and the *MSL*, *WSL*, and *Lorel* languages, Strudel/StruQL (Fernandez *et al.*, 1997; Fernandez *et al.*, 1998), and *YAT/YATL* (Cluet *et al.*, 1999)) was motivated by the *database* community, searching for a data model for *data integration* and a data format for *electronic data interchange*. Here, also the combination of document-oriented aspects with database aspects was an important motivation to go beyond classical data models which then resulted in the design of XML.

The XML data model is a hierarchical model which defines an ordered tree with attributes that can easily be interpreted as a document. The natural relationships in documents are either (i) substructures, or (ii) references to other parts of the document (where the term *reference* here means simply a cross-reference in a document). The nested elements define a document structure whose leaves are the text contents. Elements (i.e. the structuring components) are annotated by attributes which do *not* belong to the document contents. Inside the tree, cross-references (`IDREF` attributes) are allowed.

In contrast, for databases, a hierarchical structure is in general not intuitive. Here, several kinds of relationships have to be represented, between substructures and pure references. When using XML for a database-like application, these relationships have to be represented by reference attributes. On the other hand, order is often not relevant in databases.

Mainstream XML Languages. Specialized languages have been defined for XML querying, e.g. XQL (Robie, 1999), XML-QL (Deutsch *et al.*, 1999), then XPath (XPath, 1999) developed from the experiences with XQL and XSL Patterns (and XPointer) as an addressing language. XQuery (XQuery, 2001) extends XPath with SQL-like constructs. XSLT (XSLT, 1999) is an XPath-based language for transforming XML data. A proposal for extending XQuery with update constructs (XUpdate) has been published in Tatarinov *et al.* (2001); a more detailed proposal is described in Lehti (2001).

[1] We list the approaches in the temporal order of their presentation.

Other Approaches to XML. XML-GL (Ceri *et al.*, 1999; Comai *et al.*, 2001) continued the idea of GraphLog for XML. Elog (Baumgartner *et al.*, 2001a) is based on Datalog and classical first-order logic, flattening XML into predicates. It is used as an internal language in Lixto (Baumgartner *et al.*, 2001b). Bry and Schaffert (2002) present the Xcerpt language which regards XML trees as terms, similar to UnQL.

1.1 Comparison of design concepts

Amongst the existing languages for handling semistructured data and XML, there are several facets for distinguishing them in terms of the concepts they use. A more detailed comparison with individual languages can be found in section 6.

Data model. Semistructured data can be regarded as a general graph (OEM, UnQL, Strudel, GraphLog, and F-Logic) or as a tree (YATL and XML). Moreover, node-labeled graphs/trees (XML) or edge-labeled graphs (as in Strudel, UnQL, GraphLog, and F-Logic) can be distinguished; for OEM both representations can be found. It is easy to represent a node-labeled instance in a labeled model, whereas the other way requires node replication. Also, ordered (e.g. XML) and unordered (e.g. in OEM, F-Logic, and UnQL) tree/graph models are distinguished.

Access mechanism. Generally, there are two approaches for selecting items in a semistructured data tree or graph:

- by matching patterns and templates (GraphLog, MSL/WSL, UnQL, YATL, and later for XML-QL and XML-GL). In UnQL and Xcerpt, (bi)simulation between semistructured data trees is used. If a simulation of a match pattern with variables by the underlying database is found, the appropriate variable bindings are returned and used for generating an answer tree.
- navigational access, like in object-oriented database languages (OQL), as done in Lorel, StruQL, and F-Logic, and later in XPath and also in our XPathLog approach. UnQL provides both patterns and a navigational syntax.

Functionality. There are different approaches to either *generating* an answer by instantiating a generating pattern in the rule head according to the variable bindings (UnQL, StruQL, and later XML-QL, XQuery and XML-GL), or *manipulating* the underlying structure by adding information to the underlying database (GraphLog, F-Logic and XUpdate).

Note that this distinction did not exist when considering classical rule-based languages, e.g. Datalog for predicate logic. The facts derived in the rule head were added to the database – either extensionally, or intensionally as view definitions – without directly interfering with the already stored facts. Other rules of the program could easily use both the original data and the derived data. For XML, a semantics where rules generate separate structures is easy to define. In contrast, a semantics where the rule heads interfere with the database contents has to take into account that the evaluation of rules may violate the tree structure.

Underlying framework. Some of the languages are based on a kind of model-theoretic semantics: UnQL and Xcerpt directly operate on tree-term structures, employing and defining mechanisms like tree matching, term unification and (bi)simulation unification. Elog is based on Datalog and classical first-order logic, flattening XML data into predicates. F-Logic defines F-Structures that extend classical first-order logic, and then applies logic programming mechanisms to such models. For the other languages (Tsimmis, Strudel, XML-QL, XQuery, XUpdate), the semantics is directly defined in terms of data structures.

Rule-based vs. logic programming. All languages discussed above are *rule-based* and *declarative*, generating variable bindings by a matching/selection part in the “rule body” and then using these bindings in the “rule head” for generating output or updating the database. This rule-based nature is more or less explicit: F-Logic, MSL/WSL (Tsimmis), Elog, and Xcerpt use the “:-” Prolog syntax, whereas UnQL, Lorel, StruQL, XML-QL and XQuery/XUpdate cover their rule-like structure in an SQL-like clause syntax. These clausal languages allow for a straightforward extension with update constructs (as it has been done for Lorel and proposed with XUpdate for XQuery). GraphLog and XML-GL use a graphical representation.

The first, “pure” group separates strictly between the selection part in the rule body and generation/update part in the rule head, whereas UnQL, StruQL, XML-QL and XQuery allow for nested selection-generation parts in the rule bodies.

The global semantics of these languages is influenced by their functionality, distinguishing between query/transformation and query/update languages: UnQL, Xcerpt, XML-QL, and XQuery *generate* (output) structures in their head which are not feeded back into the input or internal database.

Only MSL/WSL, Elog, and F-Logic allow to for *additions to the database* or *view definitions* (depending whether bottom-up or top-down semantics is considered), and to use the derived facts in the selection/matching part of other rules. StruQL, XML-QL, and XQuery overcome this restriction by nesting selection-generation parts in the rule bodies. The `traverse` construct of UnQL (applying a subquery by structural recursion to arbitrary depth) also comes near to local view definitions. Note that these languages require regular path expressions to compute the transitive closure of a binary relation (see Fernandez *et al.* (1997)). We consider the ability to compute a transitive closure as an important feature for a language for handling semistructured data (especially, for a “logic-programming” language, since that is what makes the distinction between Datalog and the relational algebra/calculus).

The difference between *rule-based transformation languages* and *logic programming* languages is mirrored by the fact that the semantics of UnQL, Xcerpt, XML-QL, and XQuery is completely given by he semantics of their rules (qualifying them as *rule-based* languages). In contrast, the *global* semantics of F-Logic and Elog also requires the notions of the T_P operator and of minimal or well-founded models (qualifying them as *logic programming* languages). As a consequence, they require both a *model-theoretic semantics*, and an *answer semantics* for queries.

Design principles for XPathLog. XPathLog follows a logic-based approach which has been motivated by the experiences with F-Logic: XML instances are mapped to a semantical structure for interpreting *XPath-Logic* formulas. XPath-Logic is based on (i) first-order logic, and (ii) XPath expressions extended with variable bindings. The

Horn fragment of XPath-Logic, called *XPathLog*, provides a declarative, Datalog-style logic-programming language for manipulation of XML documents.

Regarding the above design principles, XPathLog is positioned as follows:

- XPathLog is completely XPath-based (i.e. navigational access). Matching and generation/update part are strictly distinguished.
  An extended XPath syntax is used for querying (rule bodies) *and* generating/manipulating data (rule heads). The rule body serves for generating variable bindings which are then used in the head for *extending* the current XML database, thereby defining an update semantics for XPath expressions.
- XPathLog uses an *edge-labeled graph* model, which is advantageous when defining several tree views of the internal database. The data model is partly ordered like in XML: subelements are ordered, attributes are unordered.
- XPathLog is a logic-programming language according to the above characterization. It works on an abstract semantical model which represents an *XML database* supporting multiple overlapping XML trees. These *X-Structures* together with the logic, called *XPath-Logic*, provide for a logical characterization of XML data. XPathLog combines the intuitive "local" semantics of addressing XML data by XPath with the appeal of the "global" semantics of logic programming. As an update language, it is based on a bottom-up semantics.
- In contrast to XML-QL, XQuery, and XSLT, the language does not use additional constructs whose semantics has to be defined separately: the only semantic prerequisite is the bottom-up evaluation strategy of Datalog or any other logic programming language.

In this paper, we describe the data model, its logical foundation, the internal semantics of queries, rules, and programs of XPathLog as a true logic programming language for XML. Some aspects have been published in May (2002); May and Behrends (2001) and with the LoPiX prototype in May (2001c). Here, we focus on the theoretical issues of modeling XML and the semantics of a language for queries and basic, elementary updates. A full report on XPathLog can be found in May (2001a).

A possible application area for XPathLog is, for instance, the integration of XML data from several sources, as done in the case study by May (2001b). Here, the power of the combination of XPath expressions with additional variable bindings allows for short and concise declarative and flexible rules. Both, queries and rules for manipulating and restructuring the internal XML database can be expressed much easier than, for example, in XQuery (where update functionality is still in a prototypical state).

Structure of the paper. Section 2 defines X-Structures as semantical structures which represent XML documents and presents XPath-Logic. The answer semantics of XPathLog as an XML query language is investigated in section 3. Section 4 defines the semantics of rule heads for generating and modifying XML data, and the semantics and evaluation of XPathLog programs. The implementation in the LoPiX (**Lo**gic **P**rogramming **i**n **X**ML) system and a case study are described in section 5.

An analysis, a general discussion of related work, and the conclusion can be found in section 6. Additional proofs can be found in Appendix Appendix A.

2 XPath-Logic: The model-theoretic framework

XPath-Logic and its Horn fragment, XPathLog, extend XPath (XPath, 1999) with Datalog-style variable bindings. XPath-Logic provides the model-theoretic framework for defining a global, logic-programming style semantics for XPathLog.

XPath (XPath, 1999) 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 *root*/*axisStep*/.../*axisStep* where *root* specifies a starting point of the expression (the root of a document, or a variable that is bound to a node in an XML instance). Every *axisStep* is of the form *axis*::*nodetest* [*qualifier*]*. The *axes* define navigation directions in an XML tree: Given an element *e*, the child axis contains all its children and the attribute axis contains all its attributes. Analogously, parent, ancestor, descendant, preceding-sibling and following-sibling axes are defined. They enumerate the respective nodes by traversing the document tree starting from *e*.

First, along the chosen axis, all elements which satisfy the *nodetest* (which specifies the nodetype or an elementtype which nodes should be considered) are selected; the resulting list is called the *context*. Then, the given *qualifier(s)* (also called *filters*) are applied to each of the nodes (as the *context node*) for finer selection. Inside qualifiers, *relative location paths* are allowed that implicitly start at the context node. Starting with this (local) result set, the next step expression is applied (for details, see (XPath, 1999) or (XQFS, 2001)). The most frequently used axes are abbreviated as *path*/nodetest for *path*/child::*nodetest*, *path*/@*nodetest* for *path*/attribute::*nodetest*, and *path*//*nodetest* for *path*/descendant-or-self::*/child::*nodetest*.

Example 1 (*XML, XPath, Result Sets*)
Consider the of the MONDIAL database (May, 2001e) for illustrations; the DTD is given as follows:

```
<!ELEMENT mondial (country+, organization+, . . . )>
<!ELEMENT country (name, population, encompassed+, border*, city+, . . . )>
    <!ATTLIST country  car_code ID #REQUIRED  capital IDREFS #REQUIRED
                       memberships IDREFS #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT encompassed EMPTY>
    <!ATTLIST encompassed  continent CDATA #REQUIRED
                           percentage CDATA #REQUIRED>
<!ELEMENT border EMPTY>
    <!ATTLIST border  country IDREF #REQUIRED   length CDATA #REQUIRED>
<!ELEMENT city (name, population*)> <!ATTLIST city country IDREF #REQUIRED>
<!ELEMENT population (#PCDATA)> <!ATTLIST population year CDATA #IMPLIED>
<!ELEMENT organization (name, abbrev, members+)>
    <!ATTLIST organization  id ID #REQUIRED   headq IDREF #IMPLIED>
<!ELEMENT abbrev (#PCDATA)>
```

```
<!ELEMENT members EMPTY>
  <!ATTLIST members  type CDATA #REQUIRED  country IDREFS #REQUIRED>
```

An excerpt of the instance is given below (and depicted as a graph can be found in Figure 1 when X-Structures are considered).

```
<country car_code="B" capital="cty-Brussels" memberships="org-eu org-nato . . . ">
  <name>Belgium</name>  <population>10170241</population>
  <encompassed continent="Europe" percentage="100"/>
  <border country="NL" length="450"/>  <border country="D" length="167"/>
      ⋮
  <city id="cty-Brussels" country="B">  <name>Brussels</name>
    <population year="95">951580</population>
  </city>
    ⋮
</country>
<country car_code="D" capital="cty-Berlin" memberships="org-eu org-nato . . . ">
  ⋮
</country>
<organization id="org-eu" headq="cty-Brussels">
  <name>European Union</name>  <abbrev>EU</abbrev>
  <members type="member" country="GR F E A D I B L . . . "/>
  <members type="membership applicant" country="AL CZ . . . "/>
</organization>
<organization id="org-nato" headq="cty-Brussels" . . . >
  ⋮
</organization>
```

The XPath expression

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

returns all names of cities such that the city belongs (i.e. is a subelement) to a country where a name subelement exists, the city's population is higher than 100,000, and its zipcode is known.

XPath is only an addressing mechanism, not a full query language. It provides the base for most XML query languages, which extend it with their special constructs (e.g. functional style in XSLT, and SQL/OQL style (e.g. joins) in XQuery). In the case of XPath-Logic and XPathLog, the extension feature are Prolog/Datalog style variable bindings, joins, and rules.

Remark 1 (*Relationship to W3C Documents*)
We restrict the considerations to the core concepts of XPath as an addressing and navigation formalism for XML data, i.e. stepwise navigation along the axes and step qualifiers/filters. For the XPath syntax and non-formal semantics, we always refer to the *W3C XPath 2.0 Working Draft* (XPath, 1999). Note that the syntax and semantics of the core concepts of XPath is the same as in XQL (Robie, 1999), XPointer, early drafts of XPath, XPath 1.0 (XPath, 1999), although both the presentation and the naming have been changed several times.

A formal semantics of XPath has been given as a *denotational* semantics in Wadler (1999) that already covers these central notions of XPath. Later, it has been re-formulated first in the *W3C XML Query Algebra* (XMQ-A, 2001) and then in the *W3C Query Formal Semantics* (XQFS, 2001), where a description in terms of type inference rules and value inference rules is given. Note that since the early implementations (e.g. xt (Clark, 1998)), the actual semantics of XSL Patterns/XPath as its "behavior" has not changed. For comparing our approach with the formal semantics of XPath, we refer to Wadler (1999), which gives a short and concise definition of the central concepts that is best suited as a reference.

2.1 Syntax of XPath-Logic

Inspired by the derivation of F-Logic from first-order logic as a logic for dealing with structures containing complex objects, XPath-Logic is defined for expressing properties of XML structures. 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 a special kind of predicates with a built-in semantics. The "basic" components of the language are XPath-Logic *PathExpressions* which are syntactically derived from XPath's *PathExpressions* by extending Path with Prolog/Datalog style variable bindings.

Definition 1 (*XPath-Logic: Syntax*)
The set of basic formulas of an XPath-Logic language is defined as follows:

- every language contains an infinite set Var of variables.
- a specific XPath-Logic language is given by its *signature* Σ of element names, attribute names, function names, constant symbols, and predicate names.
- *XPath-Logic reference expressions* over the above names extend the XPath *path expressions*: The syntax of *AxisSteps*, *axis*::*name*[*stepQualifier*]*, may be extended to bind the selected nodes to variables by "-> Var":

```
Step ::= Axis "::" NodeTest StepQualifiers
       | Axis "::" NodeTest StepQualifiers "->" Var StepQualifiers
       | Axis "::" Var StepQualifiers
       | Axis "::" Var StepQualifiers "->" Var StepQualifiers
```

  For an XPath-Logic reference expression, the *underlying XPath expression* is obtained by removing the inserted variable binding constructs.
- An *XPath-Logic predicate* is a predicate over reference expressions.
- *terms* and *atomic formulas* are defined analogously to first-order logic.
- XPath-Logic *compound formulas* are built over predicates and reference expressions, using $\wedge$, $\vee$, $\neg$, $\exists$, and $\forall$.
- XPath-Logic allows to have formulas in step qualifiers.

Note that XPath-Logic does not use the explicit dereference operator $\Rightarrow$ from XPath 2.0; instead implicit dereferencing of attributes in paths is supported.[2]
The goal of this paper is to introduce the Horn fragment from XPath-Logic, called XPathLog as a Datalog-style XML query and manipulation language. The following

[2] XPath-Logic has been designed before XPath 2.0 replaced XPath's id(.) function by the dereferencing operator. Furthermore, we use a data model that directly incorporates references.

example gives some XPathLog queries that review the basic XPath constructs, and illustrate the use of the additional variable binding syntax.

Example 2 (XPathLog: Introductory Queries)
The following examples are evaluated against the MONDIAL database.

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() = "Belgium"]//city/name/text().
true
```

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

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

```
?- //country[name/text() = "Belgium"]//city/name/text()→N.
N/"Brussels"
N/"Antwerp"
⋮
```

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

```
?- //country[name/text() = "Belgium"]//city→C.
C/brussels
C/antwerp
⋮
```

Additional Variables: XPathLog allows to bind all nodes which are traversed by an expression: The following expression returns all tuples (N_1, C, N_2) such that the city with name N_2 belongs to the country with name N_1 and car code C:

```
?- //country[name/text()→N1 and @car_code→C]//city/name/text()→N2.
N2/"Brussels" C/"B" N1/"Belgium"
N2/"Antwerp" C/"B" N1/"Belgium"
    ⋮
N2/"Berlin" C/"D" N1/"Germany"
    ⋮
```

Dereferencing IDREF Attributes: For every organization, give the name of city where the headquarter is located and all names and types of members:

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

One element of the result set is, for example,

```
N/"..."   A/"EU"   SN/"Brussels"   MT/"member"   MN/"Belgium"
```

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

?- //city/$\boxed{\text{SubElName}}$.
SubElName/name
SubElName/population
⋮

Navigation Variables: Search for all things that have the name “Monaco”. More explicitly, give the element type of all elements that have a name subelement with the text contents “Monaco”:

?- //$\boxed{\text{Type}}$→X[name/text()→“Monaco”].

Type/country	X/*country-monaco*
Type/city	X/*city-monaco*

Closed XPath-Logic formulas can e.g. be used for expressing integrity constraints.

Example 3 (*Integrity Constraints*)
There are some application-specific integrity constraints on the Mondial database:

Range restrictions: The text contents of population elements and the value of area attributes must be a non-negative number:

∀ X: ((//population/text()→X or //@area→X) ⇝ X ⩾ 0).

The sum of percentages of ethnic groups in a country is at most 100%:

∀ C: (//country→C ⇝ sum{N [C]; C/ethnicgroups/@percentage→N} ⩽ 100).

Bidirectional relationships: The membership of countries in organizations is represented bidirectionally:

∀ C,O: (//country→C[@memberships→O] ↔
∃ T: //organization→O/members[@type→T and @country→C]).

Other conditions: The country attribute of border subelements of country elements must reference a country which is encompassed by the same continent:

∀ C,C2: (//country→C/border[@country→C2] ⇝
(//country→C2 and ∃ Cont: (C/encompassed/@continent→Cont and
C2/encompassed/@continent→Cont))).

2.2 *XML instances as semantical structures*

Next, we need a basis for a model-theoretic semantics of XPath-Logic. The *information* that is carried by an XML instance is *abstractly* defined in the *XML Information Set* specification (XMLInf, 1999). It can be represented in different ways (e.g. as the *human-readable* ASCII-based notation, or by using the DOM (DOM-W3C, 1998) that provides an abstract datatype for *implementations*. There are approaches that regard XML trees as database items where the languages operate on (UnQL, Xcerpt). In our approach, the atomic items are the edges of a *graph* (than can be an XML tree, but that can also represent overlapping tree views on an internal graph-like database), called *XTreeGraph*. In contrast to the DOM model and the XML Query Data Model (XMQ-D, 2001) which use a node-labeled tree (i.e. the element and attribute names are associated with the nodes), the XTreeGraph is an *edge-labeled* model. Using an edge-labeled model proves useful for data manipulation and integration (see (May and Behrends, 2001)). Recall that XML-QL (Deutsch *et al.*,

1999) also uses an *edge-labeled* graph, which especially defines the same handling of text contents as ours; influenced by the experiences with the STRUDEL/STRUQL (Fernandez *et al.*, 1998) project for data integration.

Formally, the XTreeGraph is represented by an *X-Structure* (that interprets a signature consisting of element and attribute names, similar to a first order structure). The advantage of that approach is that it allows for *manipulating* an internal database by adding edges to the graph. Thus, XPathLog is not only a query language, but also a manipulation language. Its rule heads do not necessarily *construct* new XML trees/terms, but can update the X-Structure. As a prerequisite for mapping XML instances to X-Structures, some notation for handling lists is needed:

Notation 1 (*Lists*)
Throughout this work, the following usual notation is used:

- For two sets A and B, the set of mappings from A to B is denoted by B^A.
- A list over a domain D is a mapping from $\mathbb{N}$ to D. Thus, the set of lists over D is denoted by $D^{\mathbb{N}}$.
- the empty list is denoted by ε; a unary list containing only the element x is denoted by (x); list concatenation as an operator is denoted by $\circ$.
- $\mathsf{set}(expr_1(x_1,\ldots,x_n) \mid expr_2(x_1,\ldots,x_n))$ stands for

$$\{expr_1(x_1,\ldots,x_n) \mid expr_2(x_1,\ldots,x_n)\}$$

  (i.e. the set of all $expr_1(x_1,\ldots,x_n)$ such that the condition $expr_2(x_1,\ldots,x_n)$ holds). In the following, sets are sometimes used as lists exploiting the fact that a set can be seen as a list by an arbitrary enumeration.
- In a similar way, a list can be constructed by enumerating its elements. For a list $\ell = (i_1, i_2, \ldots)$, $\mathsf{list}_{i\in\ell}(expr_1(i) \mid expr_2(i))$ is the list of all $expr_1(i_j)$ where $expr_2(i_j)$ holds. Similar to list, $\mathsf{concat}_{i\in I}(expr_1(i) \mid expr_2(i))$ does the same if $expr_1(i)$ is already a list.
- For a finite list $\ell = (x_1,\ldots,x_n)$, $\mathsf{reverse}(\ell) = (x_n,\ldots,x_1)$.
- For a list ℓ, $\ell[i,j]$ denotes the sublist that consists of the ith to jth elements,
- For a list ℓ of pairs i.e. $\ell = ((x_1,y_1),(x_2,y_2),\ldots)$, $\ell\downarrow_1$ denotes the projection of the list to the first component of the list elements, i.e. $\ell\downarrow_1 := (x_1,x_2,\ldots)$.

X-Structures. When representing XML instances as X-Structures, (i) their elements/subelement structure, and (ii) the elements' attributes have to be represented. The universe consists of the *element nodes* of the XML instance and the *literals* used as attribute values and text contents. *Element nodes* have properties, defined by (i) subelements (which are ordered) and (ii) attributes (which are unordered). Multivalued attributes (`NMTOKENS` and `IDREFS`) are silently split, and reference attributes are silently resolved. Additionally, X-Structures support named constants and predicates as known from first-order logic.
Each XML instance is represented as a structure with a universe $\mathcal{U}$ over a signature $\Sigma = (\Sigma_N, \Sigma_F, \Sigma_C, \Sigma_P)$ which consists of

- Σ_N: element names and attribute names,
- Σ_F: names of XML-built-in functions,

- Σ_C: constant symbols, denoting elements in the XML instance (e.g. germany, interpreted as the element addressed by /mondial/country[name= "Germany"]).
- Σ_P: predicates (with arity).
- Additionally, a basic set of literal constants is assumed.

An X-Structure contains only the basic facts about the XML tree, i.e. the child and attribute relationships (similar to the DOM). Note that our approach which associates the order with the children of elements, differs from, for instance, the DOM and XML-QL approaches where a *global* order of all elements is assumed.

Definition 2 (*X-Structure*)
An X-Structure over a given signature Σ is a tuple $\mathscr{X} = (\mathscr{V}, \mathscr{L}, \mathscr{N}, \mathscr{I}, \mathscr{E}, \mathscr{A})$where the universe $\mathscr{U}$ consists of three sets $\mathscr{V}$, $\mathscr{L}$, and $\mathscr{N}$: $\mathscr{V}$ is a set of nodes (from the graph point of view, vertices), identified by internal names, $\mathscr{L}$ is a set of literals (integers, floats, strings), $\mathscr{N}$ is the set of names (e.g. as occurring in node tests). Names may be further distinguished into $\mathscr{N}_{\mathscr{E}}$, containing the element names (and a special element text() for handling text children), and $\mathscr{N}_{\mathscr{A}}$ containing the attribute names.

- $\mathscr{I}$ is a (partial) mapping, which interprets the signature: $\mathscr{I}_{\mathscr{E}}: \Sigma_N \rightarrow \mathscr{N}_{\mathscr{E}}$ and $\mathscr{I}_{\mathscr{A}}: \Sigma_N \rightarrow \mathscr{N}_{\mathscr{A}}$ interpret the names in Σ_N by element and attribute names. $\mathscr{I}_C : \Sigma_C \rightarrow \mathscr{V}$ interprets the constant symbols in Σ_C by nodes in $\mathscr{V}$, and $\mathscr{I}_F : \mathscr{V} \times \Sigma_F \times (\mathscr{V} \cup \mathscr{L} \cup \mathscr{N})^* \rightarrow \mathscr{V} \cup \mathscr{L} \cup \mathscr{N}$ represents the interpretation of built-in functions (as defined in XPQOF (2001)). Finally, $\mathscr{I}_P : \Sigma_P \times (\mathscr{V} \cup \mathscr{L} \cup \mathscr{N})^* \rightarrow \{true, false\}$ represents the interpretation of predicates.
- $\mathscr{E}$ is a (partial) mapping $\mathscr{E} : \mathscr{V} \times \mathbb{N} \times \mathscr{N}_{\mathscr{E}} \rightarrow \mathscr{V} \cup \mathscr{L}$ (subelement relationship and text contents; from the graph point of view, an ordered set of edges).
- $\mathscr{A}$ is a (partial) mapping $\mathscr{A} : \mathscr{V} \times \mathscr{N}_{\mathscr{A}} \rightarrow 2^{\mathscr{V}} \cup 2^{\mathscr{L}}$ (attribute values). *XML attribute nodes* do not belong to $\mathscr{V}$, but their literal values belong to $\mathscr{L}$. For reference attributes (IDREF), the "results" are not the ID-strings, but the target nodes in $\mathscr{V}$ themselves.

Note that $\mathscr{E}$ and $\mathscr{A}$ are not direct interpretations of Σ, but mappings that "interprete" $\mathscr{N}$. Σ is mapped to $\mathscr{N}$ before being interpreted by $\mathscr{I}$, making attribute and element names full citizens of the language (e.g. as in F-Logic).

There is a canonical mapping from the set of XML instances to the set of X-Structures. The canonical X-Structure to an XML instance is a single XML tree (cf. Figure 1), covering the DOM model.

Example 4
Figure 1 shows the X-Structure of the running example given in Example 1.

The elements of $\mathscr{V}$ (representing the element nodes) do not carry information in themselves, they are only of interest as anonymous entities (similar to object ids) which have certain properties that are given by $\mathscr{E}$, $\mathscr{A}$, and $\mathscr{I}$. In the following, mnemonic ids (e.g. *germany*) are used for elements of $\mathscr{V}$. Also, Σ_N is identified with $\mathscr{N}_{\mathscr{E}}$ and $\mathscr{N}_{\mathscr{A}}$, omitting $\mathscr{I}_{\mathscr{E}}$ and $\mathscr{I}_{\mathscr{A}}$.

In full generality, an X-Structure can also contain subelement edges and reference edges which are not conforming with the XML tree model, but which are crucial

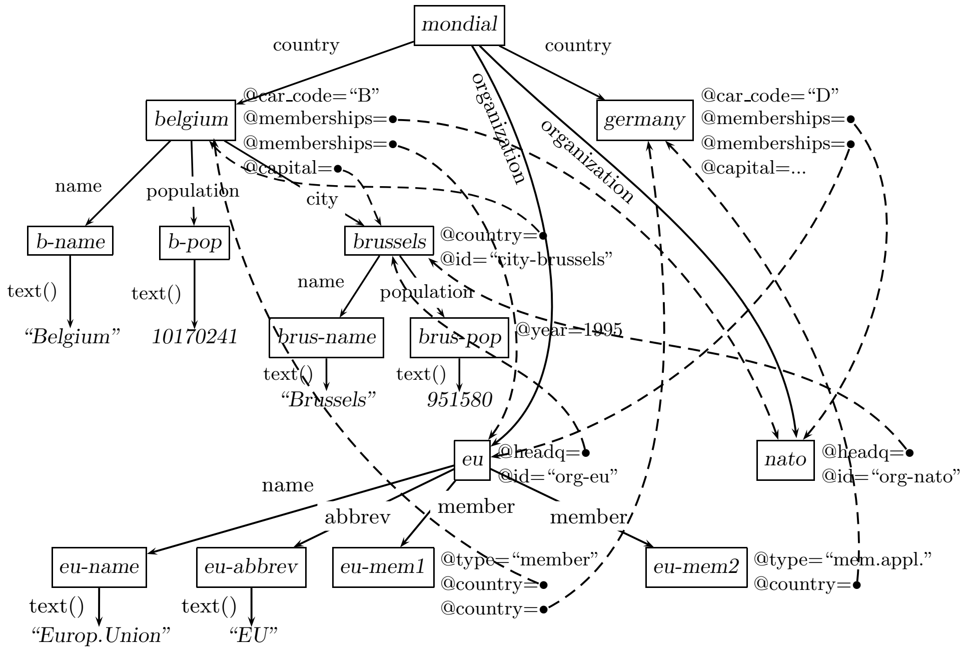


Fig. 1. Example X-Structure.

for data integration: An element may be a subelement of several other elements (as we show in section 4.2 and Figure 3), even with different names of the subelement relationship ("overlapping trees" – thus, the term *XTreeGraph* (May and Behrends, 2001) for the abstract data model).

In the following, X-Structures serve for defining a semantics for XPath-Logic, using the same terms as for XPath.

Definition 3 (*Basic Result Sets: Axes*)
For every node x in an X-Structure $\mathscr{X}$ and every *axis* a as defined in XPath,

$$\mathscr{A}_{\mathscr{X}}(a,x) \in ((\mathscr{V} \cup \mathscr{L}) \times \mathscr{N})^{\mathbb{N}}$$

is the list of pairs ($value, name$) generated by axis a with x as context node (do not confuse $\mathscr{A}_{\mathscr{X}}$ with $\mathscr{A}$ which denotes the interpretation of attributes in $\mathscr{X}$).

$$\mathscr{A}_{\mathscr{X}}(\mathsf{child}, x) \quad := \quad \mathsf{list}_{i \in \mathbb{N}}((y, name) \mid \mathscr{E}(x, i, name) = y)$$
$$\mathscr{A}_{\mathscr{X}}(\mathsf{attribute}, x) \quad := \quad \mathsf{list}((y, name) \mid y \in \mathscr{A}(x, name)) \quad \text{by some enumeration.}$$

For the other axes, $\mathscr{A}_{\mathscr{X}}(a,x)$ is derived according to the XPath specification:

$$\mathscr{A}_{\mathscr{X}}(\mathsf{parent}, x) \quad := \quad \mathsf{set}((p, \mathscr{I}(p, \mathsf{name}, ())) \mid \ x \in \mathscr{A}_{\mathscr{X}}(\mathsf{child}, p) \downarrow_1)$$
$$\mathscr{A}_{\mathscr{X}}(\mathsf{preceding\text{-}sibling}, x) \quad :=$$
$$\qquad \mathsf{concat}_{p \in \mathscr{A}_{\mathscr{X}}(\mathsf{parent},x)\downarrow_1}(\mathsf{reverse}(\mathscr{A}_{\mathscr{X}}(\mathsf{child}, p)[1, i\text{–}1]) \mid x = \mathscr{A}_{\mathscr{X}}(\mathsf{child}, p)[i])$$
$$\mathscr{A}_{\mathscr{X}}(\mathsf{following\text{-}sibling}, x) \quad :=$$
$$\qquad \mathsf{concat}_{p \in \mathscr{A}_{\mathscr{X}}(\mathsf{parent},x)\downarrow_1}(\mathscr{A}_{\mathscr{X}}(\mathsf{child}, p)[i{+}1, \mathsf{last}()] \mid x = \mathscr{A}_{\mathscr{X}}(\mathsf{child}, p)[i])$$
$$\mathscr{A}_{\mathscr{X}}(\mathsf{ancestor}, x) \quad := \quad \mathsf{concat}_{(p,n) \in \mathscr{A}_{\mathscr{X}}(\mathsf{parent},x)}(((p, n)) \ \circ \ \mathscr{A}_{\mathscr{X}}(\mathsf{ancestor}, p))$$
$$\mathscr{A}_{\mathscr{X}}(\mathsf{descendant}, x) \quad := \quad \mathsf{concat}_{(c,n) \in \mathscr{A}_{\mathscr{X}}(\mathsf{child},x)}((c, n) \circ \mathscr{A}_{\mathscr{X}}(\mathsf{descendant}, c))$$

Remark 2
Recall that in the node-labeled XML/XPath data model, the semantics of expressions is always a list or a set of labeled nodes. In contrast, $\mathcal{A}_{\mathcal{X}}$ does not return a list of (labeled) nodes, but a list of pairs $(node/literal, name)$, according to the edge-labeled data model underlying our approach.

2.3 Semantics

The semantics of XPath-Logic is defined similar to that of first-order logic by induction over the structure of expressions and formulas. The main task here is to define the semantics of reference expressions, handling navigation, order, and filtering. A reference expression simultaneously acts as a term (it has a result (list) and can be compared to terms) and as a predicate (when used in a step qualifier).

The basic result lists are provided by $\mathcal{A}_{\mathcal{X}}(axis, v)$ for every node v of $\mathcal{X}$; recall that $\mathcal{A}_{\mathcal{X}}(\mathsf{attribute}, x)$ contains literals in case of non-reference attributes, and element nodes in case of reference attributes.

2.3.1 Semantics of expressions

As for first-order logic, a *variable assignment* $\beta : \mathsf{Var} \to \mathcal{U}$ maps variables to elements of the universe $\mathcal{U}$ (nodes, literals, and names) of the underlying X-Structure. For a variable assignment β, a variable x, and $d \in \mathcal{U}$, the *modified* variable assignment β_x^d is identical with β except that it assigns d to the variable x:

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

For β as above, and a variable x, $\beta \setminus \{x\}$ denotes β without the mapping for x.

Expressions are decomposed into their *axis steps*. Every step consists of choosing an axis, preselecting nodes by a *node test*, and filtering the result by (i) "normal" predicates and (ii) XPath *context functions* (e.g. $\mathsf{position}()$ and $\mathsf{last}()$) which use the order of the intermediate result list for selecting a certain element by its index.

Definition 4 (*Semantics of XPath-Logic expressions*)
The semantics is defined by operators $\mathcal{S}$ and $\mathcal{Q}$ which are derived from the formal semantics given in Wadler (1999).

- $\mathcal{S}_{\mathcal{X}}$: Reference_Expressions $\to (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^{\mathbb{N}}$, and
  (Axes $\times$ Reference_Expressions $\times\, \mathcal{V} \times$ Var_Assignments) $\to (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^{\mathbb{N}}$
  evaluates reference expressions wrt. an axis, a context node, and a variable assignment and returns a result list. In the second case, we use *any* to denote that the actual value of the node does not matter, and we use $\mathcal{S}^{any}$ to denote that the actual value of *axis* does not matter.
- $\mathcal{Q}_{\mathcal{X}}$: (Predicate_Expressions $\times\, \mathcal{V} \times$ Var_Assignments) $\to$ Boolean
  evaluates step qualifiers wrt. a context node and a variable assignment.

Reference expressions are evaluated by $\mathscr{S}$:

1. For closed expressions, $\mathscr{S}_{\mathscr{X}}(refExpr) = \mathscr{S}^{any}_{\mathscr{X}}(refExpr, any, \emptyset)$.
2. Reference expressions are translated into path expressions wrt. a start node:
   - rooted paths: $\mathscr{S}^{any}_{\mathscr{X}}(/p, any, \beta) = \mathscr{S}^{any}_{\mathscr{X}}(p, root, \beta)$ where *root* is as follows:
     - the unique root node if only one XML document is currently stored,
     - the root node that has been used in the outer expression, if $/p$ occurs in an expression of the form $path[/p]$.
   - rooted paths in other documents:
     $\mathscr{S}^{any}_{\mathscr{X}}(\text{document(“http://... ”)}/p, any, \beta) = \mathscr{S}^{any}_{\mathscr{X}}(p, root, \beta)$
     where *root* is the root node of the document stored at http://....
   - entry points specified by a constant c: $\mathscr{S}^{any}_{\mathscr{X}}(c/p, any, \beta) = \mathscr{S}^{any}_{\mathscr{X}}(p, \mathscr{I}_C(c), \beta)$ (this is mainly of interest when multiple documents are used and constants are associated with their roots or some nodes, see section 4.2).
   - entry points specified by variables $v \in \mathsf{Var}$: $\mathscr{S}^{any}_{\mathscr{X}}(v/p, any, \beta) = \mathscr{S}^{any}_{\mathscr{X}}(p, \beta(v), \beta)$
3. Axis steps: $\mathscr{S}^{any}_{\mathscr{X}}(axis :: pattern, x, \beta) = \mathscr{S}^{axis}_{\mathscr{X}}(pattern, x, \beta)$
   where *pattern* is of the form *nodetest remainder* where *remainder* is a sequence of step qualifiers and variable bindings. These are evaluated left to right, always applying the rightmost “operation” (step qualifier or variable) to the result of the left part:
4. Node test:
   $$\begin{aligned} \mathscr{S}^{a}_{\mathscr{X}}(name, x, \beta) &= \mathsf{list}_{(v,n)\in\mathscr{A}_{\mathscr{X}}(a,x)}(v \mid n = name) \\ \mathscr{S}^{a}_{\mathscr{X}}(\mathsf{node()}, x, \beta) &= \mathsf{list}_{(v,n)\in\mathscr{A}_{\mathscr{X}}(a,x)}(v \mid v \in \mathscr{V}) \\ \mathscr{S}^{a}_{\mathscr{X}}(\mathsf{text()}, x, \beta) &= \mathsf{list}_{(v,n)\in\mathscr{A}_{\mathscr{X}}(a,x)}(v \mid v \in \mathscr{L}) \\ \mathscr{S}^{a}_{\mathscr{X}}(N, x, \beta) &= \mathsf{list}_{(v,n)\in\mathscr{A}_{\mathscr{X}}(a,x)}(v \mid n = \beta(N)) \end{aligned}$$
5. Step with variable binding:
   $$\mathscr{S}^{a}_{\mathscr{X}}(pattern \to V, x, \beta) = \begin{cases} (\beta(V)) & \text{if } (\beta(V)) \in \mathscr{S}^{a}_{\mathscr{X}}(pattern, x, \beta) \\ \varepsilon & \text{otherwise.} \end{cases}$$
6. Step qualifiers:
   $$\mathscr{S}^{a}_{\mathscr{X}}(pattern[stepQ], x, \beta) = \mathsf{list}_{y\in\mathscr{S}^{a}_{\mathscr{X}}(pattern,x,\beta)}(y \mid \mathscr{Q}_{\mathscr{X}}(stepQ, y, \beta^{k,n}_{Pos,Size}))$$
   where $L_1 := \mathscr{S}^{a}_{\mathscr{X}}(pattern, x, \beta)$ and $n := size(L_1)$, and for every y, let j the index of y in L_1, $k := j$ if a is a forward axis, and $k := n + 1 - j$ if a is a backward axis (cf. (Wadler, 1999)). *Pos* and *Size* are only used if the step qualifier contains a context function.
7. Path: $\mathscr{S}^{a}_{\mathscr{X}}(p_1/p_2, x, \beta) = \mathsf{concat}_{y\in\mathscr{S}^{a}_{\mathscr{X}}(p_1,x,\beta)}(\mathscr{S}^{any}_{\mathscr{X}}(p_2, y, \beta))$

Step Qualifiers are evaluated by $\mathscr{Q}$:

8. Reference expressions have an existential semantics in step qualifiers:
   $$\mathscr{Q}_{\mathscr{X}}(refExpr, y, \beta) \;:\Leftrightarrow\; \mathscr{S}^{any}_{\mathscr{X}}(refExpr, y, \beta) \neq \emptyset$$
9. Predicates (including comparison predicates): The semantics of predicates in XPath is *element-oriented*: $p(refExpr, term)$ evaluates to *true* if at least one pair

taken from the result sets of *refExpr* and *term* satisfies the predicate p (either defined in $\mathscr{I}_P$, or a built-in predicate of XPath (XPQOF, 2001)):

$$\begin{aligned} \mathcal{Q}_{\mathcal{X}}(pred(expr_1,\dots,expr_n),y,\beta) \;:\Leftrightarrow \\ &\text{there are } x_1 \in \mathscr{S}^{any}_{\mathcal{X}}(expr_1,y,\beta),\dots,x_n \in \mathscr{S}^{any}_{\mathcal{X}}(expr_n,y,\beta) \\ &\text{such that } (x_1,\dots,x_n) \in \mathscr{I}_P(pred) \end{aligned}$$

10. Boolean Connectives and Quantification are defined as usual.

Evaluation of terms.

11. Constants $c \in \Sigma_C$: $\mathscr{S}^a_{\mathcal{X}}(c,x,\beta) = \mathscr{I}_C(c)$. For literals, $\mathscr{S}^a_{\mathcal{X}}(lit,x,\beta) = lit$.
12. Variables: $\mathscr{S}^a_{\mathcal{X}}(var,x,\beta) = \beta(var)$.
13. Functions and arithmetics are also defined element-wise:
$\mathscr{S}^a_{\mathcal{X}}(f(expr_1,\dots,expr_n),x,\beta) =$
$\{\mathscr{I}_F(\beta(x),f,x_1,\dots,x_n) \mid x_1 \in \mathscr{S}^{any}_{\mathcal{X}}(expr_1,x,\beta),\dots,x_n \in \mathscr{S}^{any}_{\mathcal{X}}(expr_n,x,\beta)\}$
14. Context-related functions use the extension of variable bindings by pseudo-variables $Size$ and Pos in rule (6):

$$\mathscr{S}^{any}_{\mathcal{X}}(\mathsf{position()},x,\beta) = \beta(Pos) \qquad \text{and} \qquad \mathscr{S}^{any}_{\mathcal{X}}(\mathsf{last()},x,\beta) = \beta(Size)\ .$$

The following theorem states the equivalence of our semantics with that given in Wadler (1999), which is in turn equivalent to the one defined by the W3C for XPath in XQFS (2001).

Theorem 1 (*Correctness of $\mathscr{S}$ and $\mathcal{Q}$ wrt. XPath*)
For XPath reference expressions without splitting `NMTOKENS` attributes, the semantics coincides with the one given in Wadler (1999) (which already covers all core constructs of XPath as an addressing formalism): For every XPath expression $expr$,

$$\mathscr{S}_{\mathcal{X}}(expr) = \mathscr{S}[[expr]](x)$$

(for arbitrary x) where $\mathscr{S}[[expr]]$ is as defined in (Wadler, 1999).

Note that $\mathscr{S}[[expr]]$ defines only a result *set* that is implicitly ordered wrt. document order. Our semantics coincides with the document order as long as no dereferencing is used.

The proof uses the following Lemma which contains the structural induction (for proofs, see Appendix A).

Lemma 2 (*Correctness of $\mathscr{S}$ and $\mathcal{Q}$ wrt. XPath: Structural Induction*)
XPath-Logic reference expressions correspond to XPath as follows:

1. For absolute expressions (i.e. $expr = /expr'$, and no free variables):
$$\mathscr{S}^{any}_{\mathcal{X}}(expr,any,\emptyset) = \mathscr{S}[[expr]](x) \text{ for arbitrary } x.$$
2. For expressions, for all β: $\mathscr{S}^{any}_{\mathcal{X}}(expr,v,\beta) = \mathscr{S}[[expr]](v)$.
3. For step qualifiers, for all β: $\mathcal{Q}_{\mathcal{X}}(stepQ,v,\beta^{k,n}_{Pos,Size}) \Leftrightarrow \mathcal{Q}[[stepQ]](v,k,n)$.
4. For arithmetic expressions and built-in functions, for all β:
$$\mathscr{S}^{any}_{\mathcal{X}}(expr,v,\beta^{k,n}_{Pos,Size}) = \mathscr{E}[[expr]](v,k,n).$$

where $\mathcal{Q}[[expr]]$, $\mathcal{S}[[expr]]$, and $\mathcal{E}[[expr]]$ are as defined in Wadler (1999). Since XPath expressions are variable-free, β is empty except handling the pseudo variables $Size$ and Pos (which are often also empty).

The above behavior deviates from XPath for special kinds of attributes: When navigating along reference attributes, the result is not in document order, but in the same order as the referencing elements were. Additionally, NMTOKENS that are considered as atomic in XPath, are split in XPath-Logic.

2.3.2 *Semantics of formulas*

Definition 5 (*Semantics of XPath-Logic Formulas*)
Formulas are interpreted according to the usual first-order semantics

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

15. Reference Expressions: The semantics of reference expressions corresponds to a *predicate* in first-order logic, defining a purely existential semantics:
$$(\mathcal{X}, \beta) \models refExpr \quad :\Leftrightarrow \quad (\mathcal{S}_{\mathcal{X}}(refExpr, \beta)) \neq \emptyset$$
16. predicates and boolean connectives: same as in first-order logic.

The above definitions associate a *truth value semantics* with XPath-Logic formulas. The $\models$ relationcan be used for expressing integrity constraints on XML documents (see Example 3) and even sets of documents, and for reasoning on X-Structures. In contrast, when defining XPathLog as a *data manipulation language* in the next section, a completely different formalization of the semantics is given: there, as for Datalog queries, the *answer substitutions* for a formula containing free variables have to be computed.

2.4 *Annotated literals*

The XML data model distinguishes between elements and their text contents. Nevertheless, in several situations, elements containing text contents are expected to act as numbers or strings:

Example 5 (*Annotated Literals*)
Consider again the MONDIAL XML instance. The XPath queries

//country[population > 5000000]/name/text()   and
//country[population/text() > 5000000]/name/text()

are equivalent and return “Belgium” in their result set. In the first query, the *element* <population>10170241</population> is implicitly casted into its literal value.

What happens here is not evident in XML/DTD environments. A corresponding XML Schema instance would show that the *complexType* population is derived from a simple type for integers. The idea here is that an element with a text contents adds structure to a simple type by allowing subelements and attributes. Thus, text elements with attributes behave as *annotated literals*:

- comparisons, arithmetics, and (optionally) output use the literal value,
- navigation expressions use the element node, and
- in variable bindings, the variable is bound to the element, but it acts as described above when the variable is used, for instance, in a comparison.

3 XPathLog: The Horn fragment of XPath-Logic

Similar to the case of Datalog which is the function-free Horn fragment of first-order predicate logic, XPathLog is a logic programming language based on XPath-Logic. The evaluation of a query ?- $\mathsf{L}_1, \ldots, \mathsf{L}_n$ results in a set of variable bindings (of the free variables of the query) to elements of the universe. The semantics of XPathLog programs (i.e. the semantics of the evaluation of a set of XPathLog rules as a logic program) is then defined in section 4 by combining the answer semantics with the model-theoretic semantics defined in the preceding section.

Definition 6 (*XPathLog*)
Atoms are the basic components of XPathLog rules:

- an *XPathLog atom* is either an XPath-Logic *reference expression* which does not contain quantifiers or disjunction in step qualifiers, or a predicate expression over such expressions.
- an XPathLog atom is *definite* if it uses only the child, sibling, and attribute axes and the atom does not contain negation, disjunction, function applications, and context functions. These atoms are allowed in rule heads (see section 4.2). The excluded features would cause ambiguities what update is intended, e.g. “insert x as a descendant” does not specify where the element should actually be inserted.

Similar to Datalog, an *XPathLog literal* is an atom or a negated atom and an *XPathLog query* is a list ?- $\mathsf{L}_1, \ldots, \mathsf{L}_n$ of literals (in general, containing free variables). An *XPathLog rule* is a formula of the form $A_1, \ldots, A_k \leftarrow L_1, \ldots, L_n$ 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. $A_1, \ldots, A_k$ is the *head* of the rule, which may contain free variables that must also occur free in the body. In contrast to usual logic programming, we allow for lists of atoms in the rule head which are interpreted as conjunctions.

3.1 Queries in XPathLog

The semantics $\mathcal{SB}$ of XPathLog queries associates a result set and a set of *answer substitutions* with every XPathLog query by extending the above definition of $\mathcal{S}$. The semantics provides the formal base for the implementation of an *algebraic evaluation* of XPathLog queries in LoPiX (cf. section 5).

3.1.1 Answers data model

Whereas in Datalog, the answer to a query ?- $\mathsf{L}_1, \ldots, \mathsf{L}_n$ is a set of variable bindings, the semantics of XPath-Logic reference expressions is defined wrt. an X-Structure

$\mathscr{X}$ as an *annotated result list*, i.e. the semantics of an expression is

(i) a result list (corresponding to the result list of the underlying XPath expression, i.e. without the additional variable bindings), and
(ii) with every element of the result list, a list of variable bindings is associated.

The result list (i) is the same as defined by $\mathscr{S}$ in Definition 4, equivalent to the one defined for XPath expressions in Wadler (1999), and by the W3C for XPath (XQFS, 2001). Whereas from the XPath point of view for "addressing" nodes, only the result list is relevant, XPathLog queries are mapped to a set of variable bindings based on the associated bindings lists.

Example 6 (*Semantics*)
First, the semantics is illustrated by an example. Let $\mathscr{X}$ be the XML structure given in Example 4, and

```
expr := //organization→O
                    [member/@country[@car_code→C and name/text()→N]
                    ]/abbrev/text()→A.
```

The underlying XPath expression is

```
//organization[member/@country[@car_code and name/text()]]/abbrev/text() .
```

with the result list ("UN","EU",...). With each of the results, a list of bindings for the variables O, C, N, and A is associated, yielding the annotated result list

$$
\begin{aligned}
\mathscr{SB}_{\mathscr{X}}(expr) = \mathsf{list}((&\text{“UN”}, \{(\mathsf{O}/un,\ \mathsf{A}/\text{“UN”},\ \mathsf{C}/\text{“AL”},\ \mathsf{N}/\text{“Albania”}),\\
&\qquad(\mathsf{O}/un,\ \mathsf{A}/\text{“UN”},\ \mathsf{C}/\text{“GR”},\ \mathsf{N}/\text{“Greece”}),\\
&\qquad\qquad\vdots \qquad\qquad \}),\\
(&\text{“EU”}, \{(\mathsf{O}/eu,\ \mathsf{A}/\text{“EU”},\ \mathsf{C}/\text{“D”},\ \mathsf{N}/\text{“Germany”}),\\
&\qquad(\mathsf{O}/eu,\ \mathsf{A}/\text{“EU”},\ \mathsf{C}/\text{“F”},\ \mathsf{N}/\text{“France”}),\\
&\qquad\qquad\vdots \qquad\qquad \}),\\
&\vdots \qquad\qquad\qquad\qquad)
\end{aligned}
$$

Definition 7 (*Semantics*)
The domain of *sets of* variable bindings for $V_1, \ldots, V_n$ (i.e. the domain of the second component of our semantics – i.e. the possible answer sets for a query whose free variables are $V_1, \ldots, V_n$) is

$$\text{Var\_Bindings}_{V_1,\ldots,V_n} \ := \ \left(2^{((\mathscr{V}\cup\mathscr{L}\cup\mathscr{N})^n)}\right)^{\{V_1,\ldots,V_n\}}.$$

Thus, in the general case for a general set Var of variables where n is unknown,

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

is the set of sets of variable assignments. For an empty set of variables, $\{true\}$ is the only element in Var_Bindings; in contrast, $\emptyset$ means that there is no variable binding which satisfies a given requirement. We use β for denoting an individual variable binding, and $\xi \in$ Var_Bindings for denoting a set of variable bindings.

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

is the set of annotated results (i.e. an annotated result is a pair (v, ξ) where v is a node or a literal and ξ is a set of variable bindings (for the set of variables occurring free in a certain formula)).

Definition 8 (*Operators on Annotated Result Lists*)
From an annotated result list θ, the result list is obtained as Res(θ):

$$\begin{array}{rl} \text{Res} : & \text{AnnotatedResults}^{\mathbb{N}} \to (\mathscr{V} \cup \mathscr{L} \cup \mathscr{N})^{\mathbb{N}} \\ & ((x_1, \xi_1), \ldots, (x_n, \xi_n)) \mapsto (x_1, \ldots, x_n) \end{array}$$

For an annotated result list θ and a given $x \in \text{Res}(\theta)$ contained in the result list, the set of variable bindings associated with x is obtained by Bdgs(θ, x):

$$\begin{array}{l} \text{Bdgs} : \ \text{AnnotatedResults}^{\mathbb{N}} \times (\mathscr{V} \cup \mathscr{L} \cup \mathscr{N}) \to \text{Var\_Bindings} \\ \quad (((x_1, \xi_1), \ldots, (x, \xi), \ldots, (x_n, \xi_n)), x) \mapsto \xi \quad (\text{let Bdgs}(\theta, x) = \emptyset \text{ if } x \notin \text{Res}(\theta)) \end{array}$$

Note that the joins ($\bowtie$) used in this section are always purely relational joins that are applied to the bindings component.

Example 7 (*Semantics (Cont'd)*)
Continuing Example 6, Res($\mathscr{S}\mathscr{B}_{\mathscr{X}}(expr)$) = ("UN","EU",. . .) is the result list of the underlying XPath expression, and

$$\begin{array}{rl} \text{Bdgs}(\mathscr{S}\mathscr{B}_{\mathscr{X}}(expr), \text{“EU”}) = \{ & (\mathsf{O}/\mathit{eu}, \ \mathsf{A}/\text{“EU”}, \ \mathsf{C}/\text{“D”}, \ \mathsf{N}/\text{“Germany”}), \\ & (\mathsf{O}/\mathit{eu}, \ \mathsf{A}/\text{“EU”}, \ \mathsf{C}/\text{“F”}, \ \mathsf{N}/\text{“France”}), \\ & \qquad\qquad \vdots \qquad\qquad \} \end{array}$$

yields the variable bindings that are associated with the result value "EU".

3.1.2 Safety

The semantics definition evaluates formulas and expressions wrt. a given set of variable bindings which, for example, results from evaluating other subexpressions of the same query. This approach allows for a more efficient evaluation of joins (*sideways information passing strategy*), and is especially needed for evaluating *negated* expressions (by defining negation as a relational "minus" operator). Negated expressions which contain free variables are intended to *exclude* some bindings from a given set of potential results. Thus, for variables occurring in the scope of a negation, the input answer set to the negation must already provide potential bindings. This leads to a *safety* requirement similar to Datalog.

Definition 9 (*Safe Queries*)
First, safety of variables is decided for each individual ocurrence. A variable occurrence V is *safe* wrt. the query if at least one of the following holds:

- if the occurrence is in a literal L, and it is not inside the scope of a negation and not in a comparison predicate other than equality (e.g. $X < 3$ is unsafe).
- if the occurrence is in a literal L_i inside a step qualifier *pattern*[L_1 and … and L_n] and V has a safe occurrence in *pattern* or in some L_j such that $j < i$.
- if the occurrence is in a literal L_i of the query ?–L_1 and … and L_n, and V has a safe occurrence in some L_j such that $j < i$.

A query ?- L$_1$, . . . , L$_n$ is *safe* if all variable occurrences in the query are safe.

3.1.3 Semantics of expressions

In the following, the semantics of safe queries is defined. The basic (non-annotated) result lists are again provided by $\mathscr{A}_{\mathscr{X}}(axis, v)$ for every node v of $\mathscr{X}$.

Definition 10 (*Answer Semantics of XPath-Logic Expressions*)
The semantics is defined by operators $\mathscr{SB}$ and $\mathscr{QB}$ derived from $\mathscr{S}$ and $\mathscr{Q}$ as defined in Definition 4; the $\mathscr{B}$ stands for the extension with variable bindings:

- $\mathscr{SB}_{\mathscr{X}}$: (Reference_Expressions) $\rightarrow$ AnnotatedResults$^{\mathbb{N}}$, and
  (Axes $\times \mathscr{V} \times$ Reference_Expressions $\times$ Var_Bindings)
  $\rightarrow$ AnnotatedResults$^{\mathbb{N}}$

  evaluates reference expressions wrt. an axis, an (optional) context node and a given set of variable bindings and returns an annotated result list.
- $\mathscr{QB}_{\mathscr{X}}$: (Predicate_Expressions $\times \mathscr{V} \times$ Var_Bindings) $\rightarrow$ Var_Bindings

  evaluates step qualifiers wrt. a context node to sets of variable bindings.

Expressions are evaluated by $\mathscr{SB}$:

1. If no input bindings are given, $\mathscr{SB}_{\mathscr{X}}(refExpr) = \mathscr{SB}^{any}_{\mathscr{X}}(refExpr, any, \emptyset)$
2. Reference expressions are translated into path expressions wrt. a start node:
   - entry points: rooted path: $\mathscr{SB}^{any}_{\mathscr{X}}(/p, any, Bdgs) = \mathscr{SB}^{any}_{\mathscr{X}}(p, root, Bdgs)$
     where *root* is the current root as in Definition 4 (2) for the same case.
   - entry points: constants $c \in \Sigma_C$: $\mathscr{SB}^{any}_{\mathscr{X}}(c/p, any, Bdgs) = \mathscr{SB}^{any}_{\mathscr{X}}(p, c, Bdgs)$
   - rooted paths in other documents:
     $\mathscr{SB}^{any}_{\mathscr{X}}(\mathsf{document(“http://\ldots”)}/p, any, Bdgs) = \mathscr{S}^{any}_{\mathscr{X}}(p, root, \beta)$
     where *root* is the root node of the document stored at http://. . . .
   - entry points: variables $V \in \mathsf{Var}$:

     $$\mathscr{SB}^{any}_{\mathscr{X}}(V/p, any, Bdgs) = \mathsf{concat}_{x \in \mathscr{V}_{\text{active}}}\left(\mathscr{SB}^{any}_{\mathscr{X}}(p, x, Bdgs \bowtie \{V/x\})\right)$$

     where $\mathscr{V}_{\text{active}}$ is the set of element nodes in the current database.
     Remark: Here, the input bindings are used for optimization: if every $\beta \in Bdgs$ provides already bindings for the variable V, the *sideways information passing strategy* directly effects the join $\{V/x\} \bowtie Bdgs$, restricting the possible values for V which in fact results in

     $$\mathscr{SB}^{any}_{\mathscr{X}}(V/p, any, Bdgs) = \mathsf{concat}_{\beta \in Bdgs, x = \beta(V)}\left(\mathscr{SB}^{any}_{\mathscr{X}}(p, x, Bdgs \bowtie \{V/x\})\right)$$

     Thus, the propagation of bindings is not only necessary for handling negation but also provides a relevant optimization for positive literals.
     Note that in the recursive call $\mathscr{SB}^{any}_{\mathscr{X}}(p, x, Bdgs \bowtie \{V/x\})$, the propagated bindings are already augmented with the bindings for V.
3. Axis step: $\mathscr{SB}^{any}_{\mathscr{X}}(axis :: pattern, x, Bdgs) = \mathscr{SB}^{axis}_{\mathscr{X}}(pattern, x, Bdgs)$
   where *pattern* is of the form *nodetest remainder* where *remainder* is a sequence of step qualifiers and variable bindings. These are evaluated from left to right, always applying the rightmost “operation” (qualifier or variable) to the result of the left part:

4. Node test:

$$\begin{array}{lcl} \mathcal{SB}^a_{\mathcal{X}}(name, x, Bdgs) & = & \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x),\ n=name}(v, \{true\} \bowtie Bdgs) \\ \mathcal{SB}^a_{\mathcal{X}}(\mathsf{node()}, x, Bdgs) & = & \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x),\ v\in\mathcal{V}}(v, \{true\} \bowtie Bdgs) \\ \mathcal{SB}^a_{\mathcal{X}}(\mathsf{text()}, x, Bdgs) & = & \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x),\ v\in\mathcal{L}}(v, \{true\} \bowtie Bdgs) \\ \mathcal{SB}^a_{\mathcal{X}}(N, x, Bdgs) & = & \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x)}(v, \{N/v\} \bowtie Bdgs) \end{array}$$

5. Step with variable binding:

$$\mathcal{SB}^a_{\mathcal{X}}(pattern \to V, x, Bdgs) = \mathsf{list}_{(y,\xi)\in\mathcal{SB}^a_{\mathcal{X}}(pattern,x,Bdgs)}(y, \xi \bowtie \{V/y\})$$

6. Step qualifiers:

$$\begin{array}{l} \mathcal{SB}^a_{\mathcal{X}}(pattern[stepQ], x, Bdgs) = \\ \quad \mathsf{list}_{(y,\xi)\in\mathcal{SB}^a_{\mathcal{X}}(pattern,x,Bdgs),\ \mathcal{QB}_{\mathcal{X}}(stepQ,y,\xi')\neq\emptyset}(y,\ \mathcal{QB}_{\mathcal{X}}(stepQ, y, \xi') \setminus \{Pos, Size\}) \end{array}$$

If the step qualifier does not contain *context functions*, then $\xi' := \xi$, otherwise let $L := \mathcal{SB}^a_{\mathcal{X}}(pattern, x, Bdgs)$, and then for every (y, ξ) in L, ξ' is obtained as follows, extending ξ with bindings of the pseudo variables $Size$ and Pos:

- start with $\xi' = \emptyset$,
- for every $\beta \in \xi$, the list $L' = \mathsf{list}_{(y,\xi)\in L \text{ s.t. } \beta\in\xi}(y)$ contains all nodes which are selected *for the variable assignment* β.
- let $size := size(L')$, and for every y, let j the index of x_1 in L', $pos := j$ if a is a forward axis, and $pos := size{+}1{-}j$ if a is a backward axis.
- add $\beta^{Size,Pos}_{size,pos}$ to ξ'.

7. Path: $\mathcal{SB}^a_{\mathcal{X}}(p_1/p_2, x, Bdgs) = \mathsf{concat}_{(y,\xi)\in\mathcal{SB}^{any}_{\mathcal{X}}(p_1,x,Bdgs)}\mathcal{SB}^a_{\mathcal{X}}(p_2, y, \xi)$

Step Qualifiers are evaluated by $\mathcal{QB}$:

8. Reference expressions (existential semantics) in step qualifiers:

$$\mathcal{QB}_{\mathcal{X}}(refExpr, x, Bdgs) = \bigcup_{(y,\xi)\in\mathcal{SB}^{any}_{\mathcal{X}}(refExpr,x,Bdgs)} \xi$$

9. The built-in equality predicate "=" is not only a comparison if both sides are bound, but also serves as an assignment if the left-hand side is a variable $V \in \mathsf{Var}$ which is *not* bound in $Bdgs$:

$$\mathcal{QB}_{\mathcal{X}}(V = expr, x, Bdgs) = \bigcup_{(y,\xi)\in\mathcal{SB}^{any}_{\mathcal{X}}(expr,x,Bdgs)} \xi \bowtie \{V/y\}$$

All other built-in comparisons require all variables to be bound:

$$\mathcal{QB}_{\mathcal{X}}(expr_1\ \mathsf{op}\ expr_2, x, Bdgs) = \bigcup_{(x_i,\xi_i)\in\mathcal{SB}_{\mathcal{X}}(expr_i,x,Bdgs),\ x_1\ \mathsf{op}\ x_2} \xi_1 \bowtie \xi_2.$$

10. Predicates except built-in comparisons:

$$\mathcal{QB}_{\mathcal{X}}(pred(expr_1, \ldots, expr_n), x, Bdgs) = \bigcup_{\substack{(x_i,\xi_i)\in\mathcal{SB}_{\mathcal{X}}(expr_i,x,Bdgs) \\ (x_1,\ldots,x_n)\in\mathcal{I}(pred)}} \xi_1 \bowtie \ldots \bowtie \xi_n$$

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

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

12. For negated expressions which contain free variables, negation is interpreted as the "minus" operator (as known e.g. from the relational algebra) wrt. the given input bindings. Thus, all variables which occur free in A must be safe, i.e. every input variable binding has to provide a value for them.
For two variable bindings β_1, β_2, we write $\beta_1 \leqslant \beta_2$ if all variable bindings in β_1 occur also in β_2. Intuitively, in this case, if β_1 is "abandoned", β_2 should also be abandoned.

$$\begin{aligned} &\mathcal{QB}_{\mathcal{X}}(\text{not } expr, x, Bdgs) = \\ &\qquad Bdgs - \{\beta \in Bdgs \mid \text{ there is a } \beta' \in \mathcal{QB}_{\mathcal{X}}(expr, x, Bdgs) \text{ s.t. } \beta \leqslant \beta'\} \end{aligned}$$

13. Conjunction:

$$\begin{aligned} &\mathcal{QB}_{\mathcal{X}}(expr_1 \text{ and } expr_2, x, Bdgs) = \\ &\qquad \mathcal{QB}_{\mathcal{X}}(expr_1, x, Bdgs) \bowtie \mathcal{QB}_{\mathcal{X}}(expr_2, x, \mathcal{QB}_{\mathcal{X}}(expr_1, x, Bdgs)) \end{aligned}$$

Here, in case of negated conjuncts in the step qualifier, the safety of variables has to be considered. The above definition assumes that by a left-to-right evaluation of conjuncts, the evaluation is safe.

Evaluation of terms

14. Constants: for literals, $\mathcal{SB}^{any}_{\mathcal{X}}(lit, x, Bdgs) = (lit, Bdgs)$. For constants $c \in \Sigma_C$, $\mathcal{SB}^{any}_{\mathcal{X}}(c, x, Bdgs) = (\mathcal{I}_C(c), Bdgs)$.
15. Variables: the variable occurrence must be safe, then: $\mathcal{SB}^{any}_{\mathcal{X}}(var, x, Bdgs) = \text{list}_{\beta \in Bdgs}(\beta(var), \beta)$.
16. Function terms and arithmetics:

$$\begin{aligned} &\mathcal{SB}^{any}_{\mathcal{X}}(f(arg_1, \ldots, arg_n)), x, Bdgs) = \\ &\qquad \text{list}_{(x_i, \xi_i) \in \mathcal{SB}^{any}_{\mathcal{X}}(arg_1, x, Bdgs), \ldots,}(f(x_1, \ldots, x_n), \xi_1 \bowtie \ldots \bowtie \xi_n) \end{aligned}$$

where $f(x_1, \ldots, x_n)$ results from the built-in evaluation of f.

17. Context-related functions use the extension of variable bindings by pseudo-variables $Size$ and Pos in rule (6):

$$\mathcal{SB}^{any}_{\mathcal{X}}(\text{position}(), x, Bdgs) = \text{list}_{\beta \in Bdgs}(\beta(Pos), \{\beta' \in Bdgs \mid \beta(Pos) = \beta'(Pos)\})$$
$$\mathcal{SB}^{any}_{\mathcal{X}}(\text{last}(), x, Bdgs) = \text{list}_{\beta \in Bdgs}(\beta(Pos), \{\beta' \in Bdgs \mid \beta(Size) = \beta'(Size)\})$$

The above semantics is an algebraic characterization of the logical semantics of XPath-Logic expressions which has been defined in section 4:

Theorem 3 (*Correctness of $\mathcal{SB}$ and $\mathcal{QB}$*)
For every (in general, containing free variables) XPathLog expression *expr*,

$$\text{Res}(\mathcal{SB}_{\mathcal{X}}(expr)) = \bigcup_{\beta \in (\mathcal{V} \cup \mathcal{L} \cup \mathcal{N})^{\text{free}(expr)}} \mathcal{S}_{\mathcal{X}}(expr, \beta).$$

More detailed, for all $x \in \mathscr{V} \cup \mathscr{L} \cup \mathscr{N}$,

$$(x \in \mathrm{Res}(\mathscr{SB}_{\mathscr{X}}(expr)) \text{ and } \beta \in \mathrm{Bdgs}(\mathscr{SB}_{\mathscr{X}}(expr), x)) \Leftrightarrow x \in \mathscr{S}_{\mathscr{X}}(expr, \beta).$$

Again, the theorem uses a lemma which encapsulates the structural induction.

Lemma 4 (*Correctness of $\mathscr{SB}$ and $\mathscr{QB}$: Structural Induction*)
The correctness of the answers semantics of XPathLog expressions mirrors the generation of answer sets by the evaluation: The input set *Bdgs* may contain bindings for the free variables of an expression. If for some variable *var*, no binding is given, the result extends *Bdgs* with bindings of *var*. If bindings are given for *var*, this specifies a constraint on the answers to be returned (expressed by joins).

- For every absolute expression *expr*, (i.e. $expr = /expr'$) and every set *Bdgs* of variable bindings,

$$(x \in \mathrm{Res}(\mathscr{SB}_{\mathscr{X}}(expr, Bdgs)) \text{ and } \beta \in \mathrm{Bdgs}(\mathscr{SB}_{\mathscr{X}}(expr, Bdgs), x)) \Leftrightarrow$$
$$(x \in \mathscr{S}_{\mathscr{X}}(expr, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(expr)).$$

- For every expression *expr*, node v, and every set *Bdgs* of variable bindings,

$$(x \in \mathrm{Res}(\mathscr{SB}_{\mathscr{X}}(expr, v, Bdgs)) \text{ and } \beta \in \mathrm{Bdgs}(\mathscr{SB}_{\mathscr{X}}(expr, v, Bdgs), x)) \Leftrightarrow$$
$$(x \in \mathscr{S}_{\mathscr{X}}(expr, v, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(expr)).$$

- for every step qualifier *stepQ*, node v, and every set *Bdgs* of variable bindings,

$$\beta \in \mathscr{QB}_{\mathscr{X}}(stepQ, v, Bdgs) \Leftrightarrow$$
$$\mathscr{Q}_{\mathscr{X}}(stepQ, v, \beta) \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(stepQ)).$$

The proof can be found in Appendix A.

3.1.4 Semantics of queries

According to Definition 6, XPathLog queries are conjunctions of XPathLog literals. In the following, the evaluation of safe queries is defined. The definition of safety guarantees that a left-to-right evaluation of the body is well-defined (i.e. all variable evaluations in Definition 10(15) are safe). Definition 10(13) already applied left-to-right propagation when evaluating step qualifiers.

Definition 11
The evaluation $\mathscr{QB}$ is extended to atoms by

$$\begin{aligned}\mathscr{QB}_{\mathscr{X}} \;:\; & (\text{Atoms} \times \text{Var\_Bindings}) \to \text{Var\_Bindings}\\ & (A, Bdgs) \mapsto \mathscr{QB}_{\mathscr{X}}(A, root, Bdgs)\end{aligned}$$

For safe queries ?- *atom* consisting of only one atom, $\mathscr{QB}$ yields the answer bindings:

$$(\mathscr{X}, \beta) \models atom \;:\Leftrightarrow\; \beta \in \mathscr{QB}_{\mathscr{X}}(atom, \emptyset)).$$

Definition 12 (*Evaluation of Negated Literals*)
The evaluation of negated literals L is defined wrt. a set of input bindings which must cover the free variables in L, similar to negation in step qualifiers in Definition 10(12):

$$\mathscr{QB}_{\mathscr{X}}(\mathsf{not}\ A, Bdgs) :=$$
$$Bdgs - \{\beta \in Bdgs \mid \text{ there is a } \beta' \in \mathscr{QB}_{\mathscr{X}}(A, Bdgs) \text{ s.t. } \beta \leqslant \beta'\}.$$

Definition 13 (*Evaluation of Queries*)
The evaluation of a safe query ?- $L_1,\ldots,L_n$ is defined similar to the evaluation of conjunctive step qualifiers in Definition 10(13):

$$\begin{aligned}
\mathcal{QB}_{\mathcal{X}} &: \text{Conj\_Literals} \to \text{Var\_Bindings} \\
\mathcal{QB}_{\mathcal{X}}(L_1 \wedge \ldots \wedge L_i) &:= \mathcal{QB}_{\mathcal{X}}(L_1 \wedge \ldots \wedge L_{i-1}) \bowtie \\
&\quad\ \mathcal{QB}_{\mathcal{X}}(L_i, (\mathcal{QB}_{\mathcal{X}}(L_1 \wedge \ldots \wedge L_{i-1}))|_{\mathsf{free}(L_i)})
\end{aligned}$$

Given an X-Structure $\mathcal{X}$, the answer to a query ?- $\mathsf{L}_1,\ldots,\mathsf{L}_n$ is the set
$\mathsf{answers}_{\mathcal{X}}(\mathsf{L}_1,\ldots,\mathsf{L}_n) := \mathcal{QB}_{\mathcal{X}}(\mathsf{L}_1 \wedge \ldots \wedge \mathsf{L}_n)$ of variable bindings.

Theorem 5 (*Correctness: Evaluation of Queries*)
For all safe XPathLog queries Q, $\beta \in \mathcal{QB}_{\mathcal{X}}(Q) \Leftrightarrow (\mathcal{X},\beta) \models Q$.

Note that the semantics of formulas is not based on a Herbrand structure consisting of ground atoms (as "usual" Herbrand semantics are), but directly on the interpretations $\mathcal{A}_{\mathcal{X}}$ of the axes in the X-Structure, and on an interpretation of predicate symbols that can be represented as a finite set of tuples over $\mathcal{V} \cup \mathcal{L} \cup \mathcal{N}$.

4 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 (top-down) checking if something is derivable from a given set of facts, or (bottom-up) extending a given set of facts by additional, derived knowledge. In this work, we mainly investigate the bottom-up strategy, regarding XPathLog as an *update language for XML databases*: 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.

Positive XPathLog programs (i.e. the rules contain only positive literals; also step qualifiers may only contain positive expressions) are evaluated bottom-up by a T_P-like operator over the X-Structure, providing a minimal model semantics. The formal definition of a T_P operator will be given in Definition 16 for XPathLog programs after explaining the semantics of insertions and updates.

4.1 Atomization

In this section, an alternative semantics of conjunctions of *definite XPathLog atoms* is defined which provides the base for the *constructive* semantics of reference expressions in *rule heads*. The semantics is defined by resolving reference expressions syntactically into their constituting atomic steps in the same way as in F-Logic (Frohn *et al.*, 1994). A similar strategy for resolving expressions into atomic steps is followed by several approaches which store XML data in relational databases (Deutsch *et al.*, 2000; Shanmugasundaram *et al.*; Florescu and Kossmann, 1999), by flattening the XML instance to one or more universal relations.

Definition 14 (*Atomization of Formulas*)
The function atomize : XPathLogAtoms $\rightarrow 2^{\text{XPathLogAtoms}}$ resolves a definite XPathLog atom into atoms of the form *node*[*axis*::*nodetest*→*result*] and predicates over variables and constants. It will be used in Definition 16 for specifying the semantics of rule heads. atomize is defined by structural induction corresponding to the induction steps when defining $\mathcal{S}_{\mathcal{X}}$. In the following, *path* stands for a *path expression* (or a variable), and *name* for a name (or a variable).

- the entry case: $\mathsf{atomize}(/remainder) := \mathsf{atomize}(\mathsf{root}/remainder)$
- Paths are resolved into steps and step qualifiers are isolated (since context functions are not allowed in definite atoms, it can be assumed that there is at most one step qualifier, optionally preceded by a variable assignment):

$$\begin{aligned}
&\mathsf{atomize}(path/axis :: nodetest \rightarrow var[stepQualifier]\ /remainder) := \\
&\qquad \mathsf{atomize}(path[axis :: nodetest \rightarrow var]) \cup \\
&\qquad \mathsf{atomize}(var[stepQualifier]) \cup \mathsf{atomize}(var/remainder), \\
&\mathsf{atomize}(path/axis :: nodetest[stepQualifier]\ /remainder) := \\
&\qquad \mathsf{atomize}(path[axis :: nodetest \rightarrow \_X]) \cup \\
&\qquad \mathsf{atomize}(\_X[stepQualifier]) \cup \mathsf{atomize}(\_X/remainder)
\end{aligned}$$

  where $\_X$ is a new don't care variable.
- Conjunctions in step qualifiers are separated:

$$\begin{aligned}
&\mathsf{atomize}(var[pred_1 \mathsf{and}\ \ldots \mathsf{and}\ pred_n]) := \\
&\qquad \mathsf{atomize}(var[pred_1]) \cup \ldots \cup \mathsf{atomize}(var[pred_n])
\end{aligned}$$

- Predicates in step qualifiers:

$$\begin{aligned}
\mathsf{atomize}(var[pred(expr_1, \ldots, expr_n)]) := \ &\mathsf{atomize}(equality(var, expr_1, \_X_1)) \cup \ldots \\
&\mathsf{atomize}(equality(var, expr_n, \_X_n)) \cup \\
&\{pred(\_X_1, \ldots, \_X_n)\}
\end{aligned}$$

  where $equality(var, expr, X)$ is defined as follows (if $expr_i$ is a constant, it is not replaced by a variable):

  * $equality(var, expr, X) =$ "$expr \rightarrow X$" if *expr* is of the form *//remainder*,
  * $equality(var, expr, X) =$ "$var/expr \rightarrow X$" if *expr* is of the form *axis* :: *nodetest remainder*.
- Predicate atoms are handled in the same way. Note that here all arguments are absolute expressions (rooted, or starting at a constant, or at a variable).

Example 8 (*Atomization*)

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

is atomized into

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

Theorem 6 (*Correctness of* atomize)
The above semantics is equivalent to the one presented in Definition 10 for all definite XPathLog atoms A and every X-Structure $\mathscr{X}$, i.e.

$$\mathsf{answers}_{\mathscr{X}}(A) = \mathsf{answers}_{\mathscr{X}}(\mathsf{atomize}(A))$$

Again, the theorem uses a lemma which encapsulates the structural induction, using the logical semantics for showing the correctness of atomize.

Lemma 7 (*Correctness of* atomize*: Structural Induction*)
For every X-Structure $\mathscr{X}$ and every definite XPath-Logic atom A,

- for every variable assignment β of $\mathsf{free}(A)$ such that $(\mathscr{X},\beta) \models A$, there exists a variable assignment $\beta' \supseteq \beta$ of $\mathsf{free}(\mathsf{atomize}(A))$ such that $(\mathscr{X},\beta') \models \mathsf{atomize}(A)$, and
- for every variable assignment β' of $\mathsf{free}(\mathsf{atomize}(A))$ such that $(\mathscr{X},\beta') \models \mathsf{atomize}(A)$, $(\mathscr{X},\beta'|_{\mathsf{free}(A)}) \models A$.

The proof can be found in Appendix A.

4.2 Left hand side

Using *logical expressions* for specifying an update is perhaps the most important difference to approaches like XSLT, XML-QL, or XQuery where the structure to be *generated* is always specified by XML patterns, or to the update proposal for XML described in Tatarinov *et al.* (2001). 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 semantics of the left hand side of XPathLog rules – which is a list of *definite* XPathLog atoms – is now investigated based on the atomization of expressions. When used in the head, the "/" operator and the "[. . .]" construct specify which properties should be added (thus, "[. . .]" does not act as a step qualifier, but as a *constructor*). 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.

Note that the (pure) XPathLog language does *not* allow to delete or replace existing elements or attributes[3] – modifications are always monotonic in the sense that existing "things" remain.

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 9 (*Adding Attributes*)
We add the data code to Switzerland, and make it a member of the European

[3] suitable extensions, e.g. of the form delete(*elem*,*prop*,*val*) can be defined. Such extensions which would turn XPathLog into a rule-based imperative language are not investigated in this work.

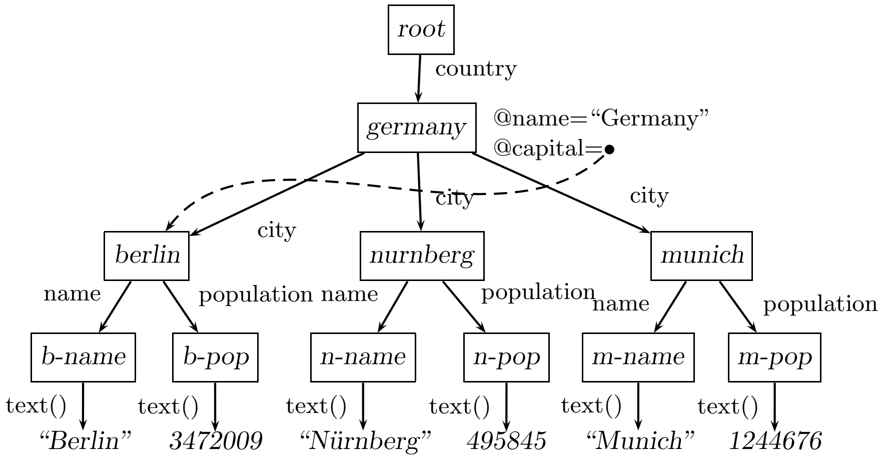


Fig. 2. Linking – before.

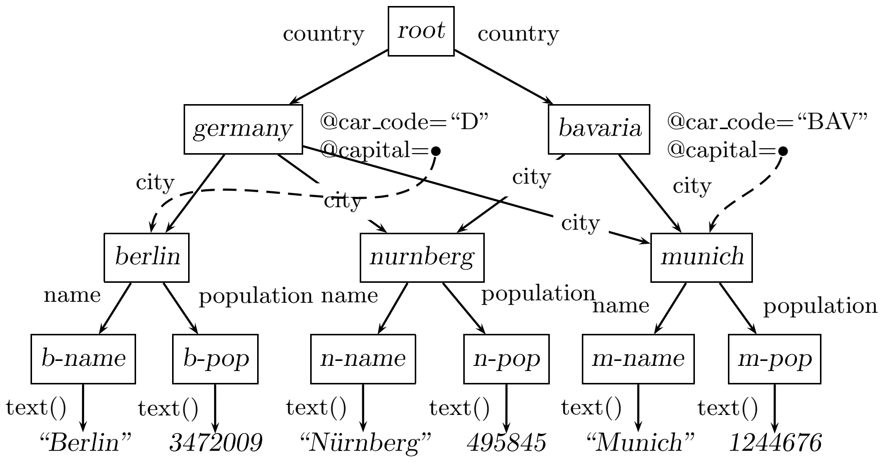


Fig. 3. Linking – after.

Union:

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

results in

<country $\boxed{\text{datacode="ch"}}$ car_code="CH" industry="machinery chemicals watches"
memberships="org-efta org-un $\boxed{\text{org-eu}}$ . . . "> . . . </country>

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

Example 10 (*Creating Elements*)
We create a new (free) country element with some properties (cf. Figures 2 and 3):

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

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. Thus, changing the elements effects both trees. *Linking* is a crucial feature for efficient restructuring and integration of data (May and Behrends, 2001).

Insertion of subelements. Existing elements can be assigned as subelements to other elements: a ground instantiated atom n[child :: $s \rightarrow m$] makes m a subelement of type s of n. In this case, m is *linked* as n/s at the end of n's children list.

Example 11 (*Inserting Subelements*)
The following two rules are equivalent to the above ones:

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

Here, the first rule creates a free element, whereas the second rule uses the variable binding of C to this element 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 h[child(i)::$s \rightarrow v$] or h[following/preceding-sibling(j)::$s \rightarrow v$], this means that the new element to be inserted should be made the ith subelement of h or jth following/preceding sibling of h, respectively.

Generation of elements by path expressions. Additionally, subelements can be created by *reference expressions* in the rule head which create nested elements that satisfy the given reference expression. The atomization introduces local variables that occur *only in the head* of the rule. Here, we follow the semantics of PathLog (Frohn *et al.*, 1994) which is implemented in Florid (Ludäscher *et al.*, 1998) for object creation. After the atomization, the resulting atoms are processed in an order such that the local variables are bound to the nodes/objects which are generated.

Example 12 (*Inserting Text Children*)
Bavaria gets a text 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.

Once-for-each-binding. In contrast to classical logic programming where it does not matter if a fact is "inserted" into the database several times (e.g. once in every T_P round), here subelements must be created exactly once for each instantiation of a rule. We define a revised T_P-operator in Definition 16.

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 (e.g. as in SchemaSQL (Lakshmanan *et al.*, 1996)). Such features have already been used for HTML-based Web data integration with F-Logic (Ludäscher *et al.*, 1998).

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.

Example 13 (*Restructuring, Name Variables*)
Consider a data source which provides data about waters according to the DTD

```
<!ELEMENT terra (water+,...)>
<!ELEMENT water (...)>   <!ATTLIST water name CDATA #REQUIRED . . . >
```

which contains, e.g. the following elements:

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

This tree should be converted into the target DTD

```
<!ELEMENT geo ((river|lake|sea)*)>
<!ELEMENT river (. . . )>      <!ATTLIST river name CDATA #REQUIRED . . . >
      (analogously for lakes and seas)
```

The first rule, `result/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 copy properties 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].
```

4.3 Global semantics of positive XPathLog programs

An XPathLog program is a declarative specification how to manipulate an XML database, starting with one or more input documents. The semantics of XPathLog programs is defined by bottom-up evaluation based on a T_P operator. Thus, the semantics coincides with the usual understanding of a stepwise process.

For implementing the once-for-each-binding approach, the T_P operator has to be extended with bookkeeping about the instances of inserted rule heads. Additionally, the insertion of subelements adds some nonmonotonicity: adding an atom n[child(i)::e→v] adds a new subelement at the ith position, making the original ith child/sibling the i+1st, etc. In case of multiple extensions to the same element, the positions are determined wrt. the original structure.

Definition 15 (*Extension of X-structures*)
Given an X-Structure $\mathcal{X}$ and a set $\mathcal{I}$ of ground-instantiated atoms as obtained from atomize to be inserted, the new X-Structure $\mathcal{X}' = \mathcal{X} \prec \mathcal{I}$ is obtained as follows:

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

- for all elements of $\mathscr{A}_{\mathscr{X}}(\mathsf{child}, x)$, let $\alpha(\mathscr{A}_{\mathscr{X}}(\mathsf{child}, x)[i]) := \mathscr{A}_{\mathscr{X}'}(\mathsf{child}, x)[i]$ (α maps the indexing from the old list to the new one).
- for all atoms $x[\mathsf{child}(i) :: e \to y] \in \mathscr{I}$, insert (y, e) into $\mathscr{A}_{\mathscr{X}'}(\mathsf{child}, x)$ immediately after $\alpha(\mathscr{A}_{\mathscr{X}}(\mathsf{child}, x)[i])$.
- for all atoms $x[\mathsf{child} :: e \to y] \in \mathscr{I}$, append (y, e) at the end of $\mathscr{A}_{\mathscr{X}'}(\mathsf{child}, x)$.
- analogously for sibling axes.
- for all atoms $x[@a \to y] \in \mathscr{I}$, append (y, a) to $\mathscr{A}_{\mathscr{X}'}(\mathsf{attribute}, x)$.

Proposition 8 (*Extension of X-Structures*)
The extension operation is correct: $\mathscr{X} \prec \mathscr{I} \models \mathscr{I}$, i.e. when querying the inserted atoms, the query evaluates to true.

With the correctness of atomize, the insertion of rule heads performs correctly:

Corollary 9 (*Correctness of Insertions*)
For inserting the ground-instantiated head of a rule, it is correct to insert the atomized head: For all ground XPathLog atoms A, $\mathscr{X} \prec \mathsf{atomize}(A) \models A$.

Definition 16 (*TX_P-Operator for XPath-Logic Programs*)
The TX-operator works on pairs $(\mathscr{X}, Dic)$ where $\mathscr{X}$ is an X-Structure, and Dic is a dictionary which associates to every rule a set ξ of bindings which have been instantiated in the current iteration:

$$\begin{aligned}(\mathscr{X}, Dic) + (\{(r_1, \beta_1), \ldots, (r_n, \beta_n)\}) \quad &:= \quad (\mathscr{X} \prec \{\beta_i(\mathsf{atomize}(head(r_i))) \mid 1 \leqslant i \leqslant n\}, \\ &\qquad\ Dic.insert(\{(r_1, \beta_1), \ldots, (r_n, \beta_n)\})), \\ (\mathscr{X}, Dic) \downarrow_1 \quad &:= \quad \mathscr{X}.\end{aligned}$$

For an XPathLog program P and an X-Structure $\mathscr{X}$,

$$\begin{aligned} TX_P(\mathscr{X}, B) &:= (\mathscr{X}, B) + \{(r, \beta) \mid r = (h \leftarrow b) \in P \text{ and } \mathscr{X} \models \beta(b), \text{ and } (r, \beta) \notin B\}, \\ TX_P^0(\mathscr{X}) &:= (\mathscr{X}, \emptyset), \\ TX_P^{i+1}(\mathscr{X}) &:= TX_P(TX_P^i(\mathscr{X})), \\ TX_P^\omega(\mathscr{X}) &:= \begin{cases} (\lim_{i \to \infty} TX_P^i(\mathscr{X})) \downarrow_1 & \text{if } TX_P^0(\mathscr{X}), TX_P^1(\mathscr{X}), \ldots \text{ converges,} \\ \bot \ \text{ otherwise.} \end{cases}\end{aligned}$$

Remark 3
Note that for pure Datalog programs P (i.e. only predicates over first-order terms), the evaluation wrt. TX_P does not change the semantics, i.e. $TX_P^\omega(\mathscr{X}) = T_P^\omega(\mathscr{X})$.

Proposition 10 (*Properties of the TX_P operator*)
The TX_P operator extends the well-known T_P operator. For all positive XPathLog programs P, the following holds:

- without considering context functions, the TX_P operator is monotonous (which guarantees that a minimal fixpoint $TX_P^\omega(\mathscr{X})$ exists),
- $TX_P^\omega(\mathscr{X}) \models P$,
- TX_P is order-preserving: for all XPathLog reference expressions *expr* which do not use negation or context functions, $\mathscr{S}_{\mathscr{X}}(expr)$ is a sublist of $\mathscr{S}_{TX_P(\mathscr{X})}(expr)$,
- for all atoms A that do not contain aggregations or function applications, if A holds in $\mathscr{X}$, then it also holds after application of $TX_P : \mathscr{X} \models A \Rightarrow TX_P(\mathscr{X}) \models A$.

Proof Both properties follow immediately from the definition. The child and attribute axes are extended solely by appending and inserting new "facts".

4.4 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* (Przymusinski, 1988) provides a solution to the problems raised by negation and other nonmonotonic features. Stratification expresses the intuitive notion of process which executes as a sequence of steps.

Note that already not all Datalog programs are stratifiable. For logics over complex structures such as F-Logic, a reasonable notion of stratification can be defined based on the names occurring at property position – as long as variables are not allowed at the property position. With variables allowed at property position, it has been showed for F-Logic in Frohn (1998) that programs are in general not stratifiable. Since (i) even without variables at property position, 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 13), syntax-based stratification is not suitable for our approach. Since the intention of XPathLog programs is in general to implement a stepwise process by bottom-up evaluation, often there is a natural, *user-defined* stratification. User defined stratification is supported in the LoPiX system May (2001d) (cf. section 5). The semantics is computed in the same way as for positive programs by iterating the TX_P operator for each stratum.

Language extensions. In addition to the pure language as described above, XPathLog supports several extensions. A detailed description of, for example, aggregation (as known, e.g. from SQL) and a class hierarchy and signatures (taken from F-Logic), and data-driven Web access can be found in May (2001a).

5 Implementation and application

5.1 Implementation: The LoPiX *system*

XPathLog has been implemented in the LoPiX system May (2001d) which extends the pure XPathLog language with a Web-aware environment and additional functionality for data integration. LoPiX has been developed using major components from the Florid system (FLORID, 1998; Ludäscher *et al*., 1998), an implementation (in C++) of F-Logic. Due to the similarities between the F-Logic data model and the XML data model in general, and XPathLogic's multi-overlapping-tree model in particular, the Florid modules provided a solid base for an XPathLog implementation. Especially the functionality of the complete module for the evaluation of a deductive language over a data model with complex objects could be reused. The system architecture of LoPiX is depicted in Figure 4.

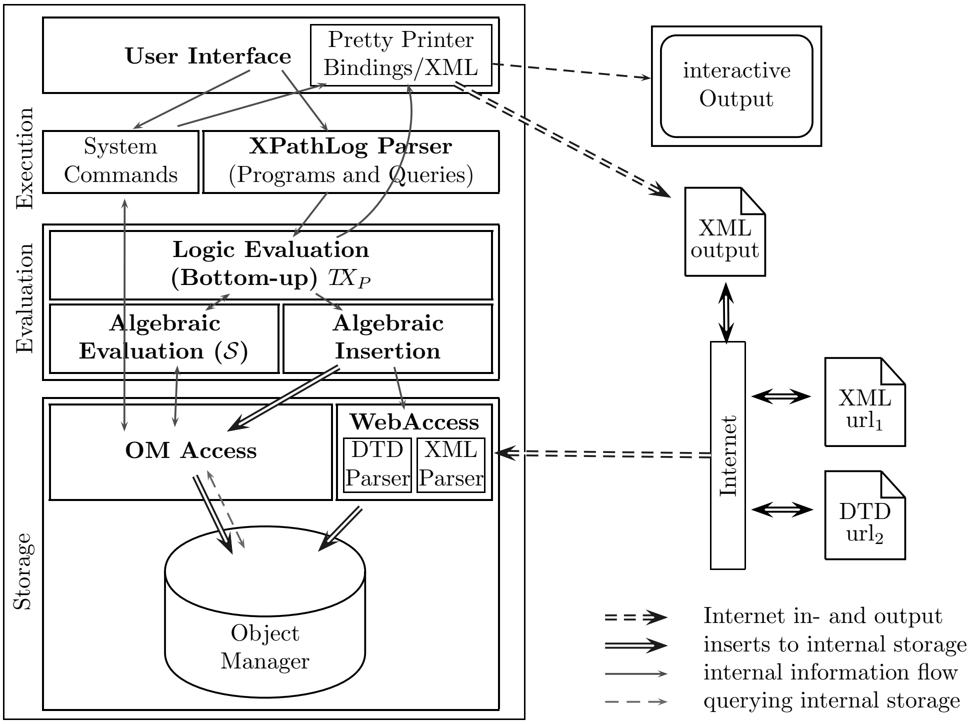

Execution
Evaluation
Storage
User Interface
Pretty Printer Bindings/XML
System Commands
XPathLog Parser (Programs and Queries)
Logic Evaluation (Bottom-up) T_P
Algebraic Evaluation ($\mathcal{S}$)
Algebraic Insertion
OM Access
WebAccess
DTD Parser
XML Parser
Object Manager
interactive Output
XML output
Internet
XML url_1
DTD url_2
Internet in- and output
inserts to internal storage
internal information flow
querying internal storage

Fig. 4. Architecture of the LoPiX system.

Storage. The (extensional) database, is stored in the *ObjectManager*. Here, two variants have been developed: the first one uses a proprietary integrated, frame-based model (from Florid) that is equipped with indexes for optimized access, whereas the second one is based on a standard DOM implementation.

The *ObjectManagerAccess* encapsulates the storage by implementing the abstract XTreeGraph data model based on the contents of the *ObjectManager*. This abstraction level also adds intensional properties including derived axes, transitivity of class hierarchy, downwards closure of signatures, inheritance, object fusion, synonyms, built-in functionality for data conversion, string handling including matching regular expressions, arithmetics, and aggregation operators.

The *WebAccess* functionality is closely intertwined with the *OMAccess* module: XML sources are mapped to trees in the internal database. Additionally, a method for mapping a DTD to XPathLog signature atoms is provided.

Evaluation. The central *Evaluation* module (*LogicEvaluation, AlgebraicEvaluation*, and *AlgebraicInsert*) is taken nearly unchanged from Florid and provides in fact a *generic* implementation of a deductive language over a data model with complex objects. *LogicEvaluation* implements a seminaive bottom-up evaluation of rules. *AlgebraicEvaluation* translates rule bodies and heads into the underlying object algebra and evaluates the generated algebraic expressions using the query interface of *OMAccess*. The object algebra implements the semantics of XPathLog queries described in section 3.1, generating sets of tuples of variable bindings. *AlgebraicInsert* instantiates the rule heads with the generated variable bindings and adds the corresponding facts into the database using again the *OMAccess* interface, implementing the TX_P-semantics defined in section 4. The evaluation of algebraic expressions does not materialize any intermediate result, but is purely based on nested iterators.

Execution and UserInterface. The execution module provides the infrastructure for the system, consisting of a *Parser* (lex/yacc-based) and a *SystemCommands* module that implements (partially) non-logical commands for controlling the evaluation process. The *UserInterface* module allows to use LoPiX from the command shell by invoking system commands and stating interactive queries. The *PrettyPrinter* outputs answers in the variable bindings format known from Datalog; additionally, the result of queries that bind only a single variable can be output as a result set in XML ASCII representation. Additionally, *result views*, i.e. the projections of trees rooted in a given node to a given signature can be exported.

5.2 *Case study:* **Mondial**

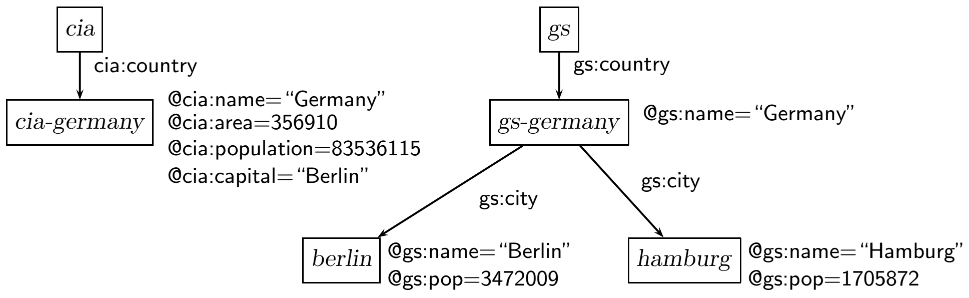
XPathLog/LoPiX has successfully been applied in the Mondial case study (May, 2001e; May 2001b). There, the practicability of the approach for data integration is illustrated by integrating a geographical database from the XML representations of its sources (which have been created by Florid wrappers in May (1999)).

The CIA World Factbook: The CIA World Factbook Country Listing (cia:, `http://www.odci.gov/cia/publications/pubs.html`) provides political, economic, and social and some geographical information about the countries. A separate part of the CIA World Factbook provides information about political and economical organizations (orgs:). Here, the data sources overlap by the membership relation: with every organization, the member countries are stored in orgs by name (using the same names as in the cia part).

Global Statistics: Cities and Provinces: The *Global Statistics* data (gs:, `http://www.stats.demon.nl`) provides information (grouped by countries) about administrative divisions (area and population, sometimes capital) and main cities (population with year, and province). Whereas the country names are the same as in CIA, the names of cities, that are for example capitals of countries or where the headquarter of a political organization is located, may differ.

The case-study showed that XPathLog allows for an effective, and elegant programming of the integration process. The nature of an XPathLog program as a list of rules allows for grouping rules which together handle a certain task. The programs are modular which also allows for adapting them to potential changes in the source structure and ontology.

For data integration in general, not only "simple" updates are desired, but also specialized operations on tree fragments. The result is constructed using subtrees, elements, and literals of the input sources by the integration operations that extend



(a) Element fusion – before

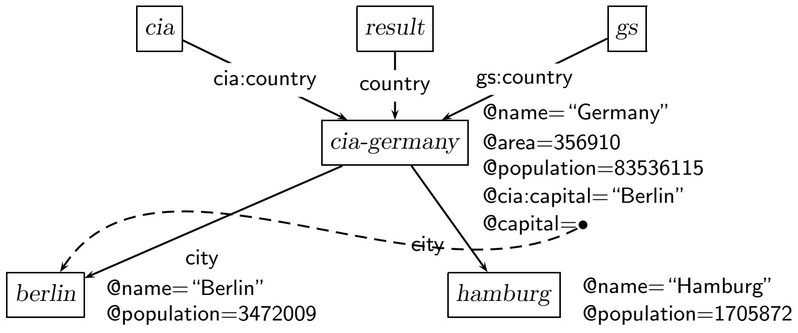


Fig. 5. (b) Element fusion – after.

the basic XPathLog in LoPiX. These operations heavily depend on the use of the XTreeGraph data model (May and Behrends, 2001).

Fusing elements and subtrees. Fusing elements that represent the same real-world entity from different data sources into a unified element is an important task in information integration. The result is still an element of *both* source trees, and collects the attributes and subelements of both original elements.

Example 14 (*Object Fusion*)
Consider two data sources as shown below and in Figure 5(a). Both describe countries, where cia contains information about name, area, population, and capital, and gs contains information about cities.

```
<!ELEMENT cia (country+)>
<!ELEMENT country (border*)>
    <!ATTLIST country  name CDATA #REQUIRED       car_code ID #REQUIRED
                       area CDATA #IMPLIED        population CDATA #IMPLIED
                       capital CDATA #REQUIRED>
<!ELEMENT border (#PCDATA)>   <!ATTLIST border country IDREF #REQUIRED>
<!ELEMENT gs (country+)>
<!ELEMENT country (city+)>  <!ATTLIST country  name CDATA #REQUIRED>
<!ELEMENT city EMPTY>
    <!ATTLIST city name CDATA #REQUIRED pop CDATA #REQUIRED>
```

Excerpts of the instances:

```
<cia>                                           <gs>
  <country car_code='D'  capital='Berlin'         <country name='Germany'>
           name='Germany'  area='356910'            <city name="Berlin" pop="3472009"/>
           population='83536115'>                 <city name="Hamburg" pop="1705872"/>
     <border country='F'>451</border>                 :
     <border country='A'>784</border>             </country>
       :                                             :
  </country>                                    </gs>
     :
</cia>
```

An obvious and typical integration step is to unify the countries in the cia tree with the countries in the gs tree. In XPathLog, this is done by the rule

```
C1 = C2 :- cia/cia:country→C1[@cia:name→N], gs/gs:country→C2[@gs:name→N].
```

The example is continued below – Figure 5(b) depicts the final result.

Synonyms. Names are also subject of operations, for example, the integrated database uses a unified terminology that differs from the source terminologies. Instead of generating new relationships between nodes, target terminology is introduced by synonyms for already existing relationships.

Example 15 (*Integration: Synonyms*)
Especially, synonyms are an efficient means for taking a whole property from a source tree (and namespace) to the result tree: Consider the situation obtained in Example 14 where the following synonyms are defined:

```
cia:name = name.                 gs:city = city.          gs:text() = text().
cia:area = area.                 gs:name = name.
cia:population = population.     gs:pop = population.
```

Adding Links. The integrated database often contains additional links (by subelement or reference attribute relationships) between elements that originally belong to different sources.

Example 16 (*Integration: Additional Links*)
The integration is completed by linking the country subtrees to a result tree and adding the capital reference attributes, here, using *germany*[@cia:capital="Berlin"] and *berlin*[name="Berlin"]. The resulting tree fragment is given in Figure 5(b).
In XPathLog, this is done by the rules

```
result[country→C] :- cia[cia:country→C].
C[@capital→City] :-
     result/country→C[@cia:capital→Name and city→City[@name=Name]].
```

Projection. When the integration and restructuring process is completed, projections are used to define *result views* of the internal database. A result view is an XML tree, e.g. specified by a root node and a DTD.

The complete case study in May (2001b) describes the process of data integration, data cleaning, data restructuring and distinguishing result tree views. The program is easily extendible by additional rules for adding another data source.

6 Analysis, related work, and conclusion

6.1 Comparison with other XML languages

XPathLog vs. requirements. In Fernandez *et al.* (1999), XQL, XML-QL, and the languages YATL Cluet *et al.* (1999) and *Lorel* Abiteboul *et al.* (1997); Goldman *et al.* (1999) are compared and essential features of an XML query language have been identified. XPathLog relates to their requirements as follows:

- existence of some kind of pattern clause, step qualifier clause, and constructor clause: pattern and step qualifier clause are the same as in XPath, extended with variable bindings. The path patterns are superior to XML patterns (e.g. as used in XML-QL) since they allow for dereferencing and navigation along different axes. The constructor clause uses the same XPath-based syntax.
- constructs for imposing nesting and order: nested elements in the result tree are generated by subsequent rules which stepwise generate the result. Grouping (via stepwise generation) and order (via child(*i*)::name) is supported.
- combining data from different sources is supported.
- tag variables or regular path expressions: tag variables are supported, regular path expressions are not included in the basic XPathLog language (also not in XPath). They are definable as derived relations.
- alternatives are expressible using a separate rule for each alternative.
- checking for absence of information: existence or non-existence of properties can be tested using negation, e.g. //country[not @indep_date].
- external functions: aggregation, string functions and some data conversion is built-in; the set of functions is extensible.
- navigation along references: implicit dereferencing is supported.

Semistructured data languages. We have already mentioned the use of logic programming style languages in pre-XML projects on semistructured data in section 1.

GraphLog (Consens and Mendelzon, 1990) and F-Logic/FLORID (Kifer and Lausen, 1989; Kifer *et al.*, 1995) presented logic-programming languages over graph data models that cover the semistructured data model, but did not yet use that notion.

In GraphLog, graphical queries are defined as patterns that are matched with an underlying graph database. The matched vertices are bound to variables that are then used for generating an output instance or for adding edges to the input graph in the rule head. In the graphical representation, the “rule head” is represented as a distinguished edge in the graphical pattern (to be added to the input graph). The language can be seen as a graphical representation of Datalog over binary relations. Thus, according to our criteria stated in section 1.1, GraphLog qualifies as a logic-programming language. GraphLog excludes recursive rules, but allows for *closure literals* that represent the closure of a binary predicate; thus the expressiveness of the language is the same as for stratified linear Datalog.

F-Logic (Kifer and Lausen, 1989; Kifer *et al.*, 1995) is a *deductive object-oriented database language* that can be seen as an early concept of a *semistructured, self-describing* data model. F-Logic defines a data model, a logic, and a database query and programming language (similar to the relationship between the X-Structures, XPath-Logic and XPathLog). The experiences with F-Logic as a formal framework and as a language for data extraction and integration from the Web (Ludäscher *et al.*, 1998; May, 1999) provided the background for the design of XPath-Logic and XPathLog as a crossbreed between XPath and F-Logic, combining the experiences with F-Logic as a successful (but “proprietary”) language for data integration with the standards of XML and XPath was a well-grounded evolution step. Especially, the power of the graph-based F-Logic data model compared with the restricted tree model of XML made up a central requirement in the design of XPathLog, leading to the XTreeGraph data model for *virtual trees in a graph database*. Another important aspect taken from F-Logic is to have names as first-order citizens of the language for a seamless incorporation of metadata information. Due to these similarities, it was possible to base the implementation of XPathLog in the LoPiX system on the F-Logic system Florid.

The *OEM (Object Exchange Model)* of the Tsimmis project (Garcia-Molina *et al.*, 1997; Abiteboul *et al.*, 1997) was the first data model that was dedicated explicitly to the notion of *semistructured data*. OEM is a graph based model, for which node-labeled and edge-labeled presentations have been given. With *WSL* and *MSL (Wrapper/Mediator Specification Language)*, Datalog-style programming languages have been presented. The *Lorel* language (McHugh *et al.*, 1997) is similar to OQL, combining navigational access (extended with regular path expressions) with clauses. Lorel supports SQL-like, procedural update constructs. *Lorel* has been migrated to XML in Goldman *et al.* (1999). In contrast to the XPathLog/LoPiX migration, Lorel does not support the XML axes.

UnQL (Buneman *et al.*, 1996, 2000) operates on rooted, edge-labeled graphs. It embeds *graph schemata* that are matched as patterns with the underlying database, combined with navigational access into SQL-like clauses. UnQL’s semantics is based on *structural recursion* – similar to the later XSL.

Strudel/StruQL (Fernandez *et al.*, 1997, 1998) also uses an edge-labeled graph model. Its syntax embeds query patterns that are matched with the underlying database into SQL-like clauses. StruQL rules specify what new elementary structures are created, and what links between them are created. The Strudel project has been continued for XML with XML-QL.

The *YATL* language of the *YAT* system (Cluet *et al.*, 1999) is a pre-XML proposal, already using SGML and DTDs. Its trees provide a unified model for relational, object-oriented (ODMG), and semistructured/document data (SGML). The YATL language follows a rule-based design for complex objects in the style of MSL or F-Logic; it supports regular path expressions and tree algebraic operations. In Christophides *et al.* (2000), the YAT system is turned into an XML system for data integration, which still does not use any XML/XPath language constructs. After mapping an XML instance to a YAT tree, there is no notion of attributes. Dereferencing is not explicitly supported, and it has no notion of the XML axes (similar to the same issue for XML-QL).

XML languages. XML-QL and XQuery embed XML patterns and XPath expressions, respectively, into SQL-style clauses. Expressions can be nested.

XML-QL (Deutsch *et al.*, 1999) uses XML patterns in the head (`CONSTRUCT`) and body (`WHERE`) clause. In that aspect, it is the XML-pattern-counterpart to the XPath-based XPathLog. The XML-QL patterns for selecting elements do not support the XML *axes* except the child axis, and indirectly the descendant by regular path expressions. XML-QL does not support updates; a potential combination of XML patterns and updates is not obvious.

XQuery (XQuery, 2001) embeds XPath expressions in SQL-style `FOR - LET - WHERE - RETURN` clauses, where the `RETURN` clause specifies the result as an XML pattern. A proposal for specifying *updates* in XQuery has been published in Tatarinov *et al.* (2001). A more detailed proposal is described in Lehti (2001) and implemented in (Software AG, 2001).

XML-GL (Ceri *et al.*, 1999; Comai *et al.*, 2001) continued the idea of GraphLog for XML. In contrast to GraphLog, the rule body and the rule head are represented by separate graphs, called *extract-match-construct-clip-queries*. The rule heads *generate* separate XML structures. Recursion is excluded. The MIX *(Mediation in XML)* system (Baru *et al.*, 1999) uses the XMAS *(XML Matching and Structuring)* language, derived from XML-QL for data integration; a graphical user interface similar to XML-GL is provided. *XDuce* (Hosoya and Pierce, 2000) is a functional-style tree transformation language which uses *regular expression pattern matching* of (originally, SGML) DTDs for formulating queries against XML instances.

Xcerpt (Bry and Schaffert, 2002) is a *pattern-based* language for querying and transforming XML data. It follows a clean, rule-based design where the query (matching) part in the body is separated from the generation part in the rule head. XML instances are regarded as terms that are matched by a term pattern in the rule body, generating variable bindings. The semantics and the implementation is given by *simulation unification* that computes answer substitutions for the variables in the match pattern against the underlying XML term (similar to UnQL). Then, the term in the rule head is instantiated with these variable bindings. Since rule heads have only a *generating* semantics, but not an update semantics, Xcerpt can only be used for querying and transforming XML data, but not for updating/extending an existing internal XML database. It has a rule-based semantics, but there is no global logic programming semantics for the evaluation of programs.

Elog (Baumgartner *et al.*, 2001a) is a logic programming language for XML which is used as *internal* language for XML data extraction in the Lixto project (Baumgartner *et al.*, 2001b). It is based on flattening XML data into Datalog with specialized Web Access predicates.

Table 1 gives a comparison of some of the above-mentioned languages. The "paradigm" column indicates the underlying semantics of the languages: the semantics of SQL-like languages is best given as an algebraic semantics that specifies the type and value of expression, allowing for nested expressions. For rule-based languages, a denotational specification of the outcome of the right-hand side (query) and of the result of the left-hand side is required. Logic programming languages

Table 1. *Comparison*

SSD	GraphLog	WSL/MSL	Lorel	UnQL	StruQL	F-Logic
DataModel	graph	graph/atoms	graph	graph	graph	graph
Access	patterns	patterns	pat/nav	term unif.	patterns	navigation
Views	y	y	y	y	y	y
Interfering additions	y	n	y	n	n	y
Paradigm	LP	rules	SQL	SQL	SQL	LP
XML	XML-QL	XQuery	XML-GL	Xcerpt	Elog	XPathLog
DataModel	XML	XML	XML tree	XML tree	atoms	XTreeGraph
Access	patterns	navigation	patterns	term unif.	(atoms)	navigation
Views	y	y	y	y	y	y
Interf. add.	n	(XUpdate)	n	n	(+)	y
standard-based	no	is standard	no	no	(no)	yes: Xpath
Paradigm	SQL	SQL	rules	rules	LP	LP

require both a model-theoretic semantics (to specify the outcome of rule heads, and for the *global* semantics), and an answer semantics for the querying part.

6.2 Contributions

We have described XPath-Logic as a logic-based framework for handling XML data, together with an extended XML data model that is suitable for XML querying, manipulation, and integration. XPathLog combines the intuitive "local" semantics of addressing XML data by XPath with the appeal of the "global" logic programming semantics: it is completely XPath-based, i.e. both the rule bodies and the rule heads use an extended XPath syntax, thereby defining a *constructive* semantics for XPath expressions. Although the syntactic difference between XPath and XPathLog is small, the extension adds much to the language by turning it into a data manipulation language. The close relationship with XPath ensures that its declarative semantics is well understood from the XML perspective. Since both XPath and rule-based programming by using variable bindings are well-known, intuitive concepts, the "effect" of the language is easy to understand on an intuitive basis, making programming easy. The logic programming background provides a strong theoretical foundation of the language concept.

The data model and the language are implemented in the LoPiX system. Its practicability has been demonstrated by the Mondial case study.

Appendix A Proofs

Proof of Theorem 1 and Lemma 2: The proof is done by structural induction. The enumeration is the same as in Definition 4. Below, β is an assignment of the pseudo variables *Size* and *Pos* (often even empty). We write $\overset{**}{=}$ for "equals by definition in (Wadler, 1999)". The individual items of the theorem are referred to below by IH1, …, IH4 (induction hypotheses).

1. For closed, absolute expressions (i.e. without free variables),
   $\mathcal{S}_{\mathcal{X}}(/expr) \overset{\text{Def}}{=} \mathcal{S}^{any}_{\mathcal{X}}(/expr, any, \emptyset) \overset{\text{IH1}}{=} \mathcal{S}[[/expr]](x)$ for arbitrary x.
2. Reference expressions ((Wadler, 1999): only absolute expressions):
   $\mathcal{S}^{any}_{\mathcal{X}}(/p, any, \beta) \overset{\text{Def}}{=} \mathcal{S}^{any}_{\mathcal{X}}(p, root, \beta) \overset{\text{IH2}}{=} \mathcal{S}^{any}[[/expr]](root)$.
3. Axis step:
   $\mathcal{S}^{any}_{\mathcal{X}}(axis :: pattern, x, \beta) \overset{\text{Def}}{=} \mathcal{S}^{axis}_{\mathcal{X}}(pattern, x, \beta) \overset{\text{IH2}}{=} \mathcal{S}^{axis}[[/pattern]](x)$.
4. The node test is the base case which is directly mapped to the axes:
   $\mathcal{S}^{a}_{\mathcal{X}}(name, x, \beta) \overset{\text{Def}}{=} \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x)}(v \mid n = name)$
   which is characterized in Wadler (1999) ($\mathcal{A}[[a]]$ enumerates the axes, $\mathcal{P}(a)$ gives the axes' principal nodetype) by
   $\{x_1 \mid x_1 \in \mathcal{A}[[a]]x,\ nodetype(x_1) = \mathcal{P}(a),\ \mathsf{name}(x_1) = name\}$
   which is the definition of $\mathcal{S}^{a}[[name]](x)$. Note that dereferencing `IDREF(S)` and splitting `NMTOKENS` has been excluded, thus, the result list is still in document order. Similar (note that node() is not defined in Wadler (1999), we extend the definition according to the XPath specification)

$$\begin{aligned}\mathcal{S}^{a}_{\mathcal{X}}(\mathsf{node}(), x, \beta) &\overset{\text{Def}}{=} \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x)}(v \mid v \in \mathcal{V})\\ &= \{x_1 \mid x_1 \in \mathcal{A}[[a]]x,\ nodetype(x_1) = element\} = \mathcal{S}^{a}[[\mathsf{node}()]](x)\\ \mathcal{S}^{a}_{\mathcal{X}}(\mathsf{text}(), x, \beta) &\overset{\text{Def}}{=} \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,x)}(v \mid v \in \mathcal{V})\\ &= \{x_1 \mid x_1 \in \mathcal{A}[[a]]x,\ nodetype(x_1) = Text\} = \mathcal{S}^{a}[[\mathsf{text}()]](x).\end{aligned}$$

5. Step with variable binding: obvious
6. Step qualifiers:

$$\mathcal{S}^{a}_{\mathcal{X}}(pattern[stepQ], x, \beta) \overset{\text{Def}}{=} \mathsf{list}_{y\in\mathcal{S}^{a}_{\mathcal{X}}(pattern,x,\beta)}\big(y \mid \mathcal{Q}_{\mathcal{X}}\big(stepQ, y, \beta^{k,n}_{Pos,Size}\big)\big)$$

   where $L_1 \overset{\text{Def}}{=} \mathcal{S}^{a}_{\mathcal{X}}(pattern, x, \beta)$ which equals $\mathcal{S}^{a}[[pattern]](x, k, n)$ by induction hypothesis IH3 and $n := size(L_1)$, and for every y, let j the index of y in L_1 (which equals $size(\{x_1 \mid x_1 \in L_1, x_1 \leqslant_{doc} y\})$), $k := j$ if a is a forward axis, and $k := n+1-j$ if a is a backward axis. This is the same as defined for $\mathcal{S}^{a}[[pattern[stepQualifier]]](x)$ and, by induction hypothesis IH3, the same as

$$\overset{\text{IH3}}{=} \mathsf{list}_{y\in\mathcal{S}^{a}_{\mathcal{X}}(pattern,x,\beta)}(y \mid \mathcal{Q}[[stepQ]](, y, k, n)) \overset{**}{=} \mathcal{S}^{a}[[pattern[stepQ]]](x).$$

7. Path: $\mathcal{S}^{a}_{\mathcal{X}}(p_1/p_2, x, \beta) \overset{\text{Def}}{=} \mathsf{concat}_{y\in\mathcal{S}^{a}_{\mathcal{X}}(p_1,x,\beta)}\big(\mathcal{S}^{any}_{\mathcal{X}}(p_2, y, \beta)\big)$
   $\overset{\text{IH2}}{=} \mathsf{concat}_{y\in\mathcal{S}^{a}[[p_1]](x)}(\mathcal{S}^{a}[[p_2]](y)) \overset{**}{=} \mathcal{S}^{a}[[p_1/p_2]](x)$.
8. Reference expressions (existential semantics) in step qualifiers:

$$\begin{aligned}\mathcal{Q}_{\mathcal{X}}(refExpr, x, \beta) &\overset{\text{Def}}{\Leftrightarrow} \mathcal{S}^{any}_{\mathcal{X}}(refExpr, x, \beta) \neq \emptyset\\ &\overset{\text{IH3}}{\Leftrightarrow} \mathcal{S}^{child}[[refExpr]](x) \neq \emptyset \overset{**}{\Leftrightarrow} \mathcal{Q}[[refExpr]](x, k, n).\end{aligned}$$

   (for all k, n since these are not used in *refExpr*).
9. Predicates: Wadler (1999) knows only the "=" comparison. The definition is although not complete: e.g. for step qualifiers of the form [a/b/c = "foo"]

which are allowed in XPath, there is no semantics defined. We extend the semantics according to the XPath specification, applying either $\mathscr{S}$ or $\mathscr{E}$.

$$\begin{aligned}
&\mathscr{Q}_{\mathscr{X}}(pred(expr_1,\dots,expr_n),x,\beta)\\
&\overset{\text{Def}}{\Leftrightarrow}\ \text{there are } x_1\in\mathscr{S}^{any}_{\mathscr{X}}(expr_1,x,\beta),\dots,x_n\in\mathscr{S}^{any}_{\mathscr{X}}(expr_n,x,\beta)\\
&\qquad \text{such that } (x_1,\dots,x_n)\in\mathscr{I}_P(pred)\\
&\overset{\text{IH2/4}}{\Leftrightarrow}\ \text{there are } x_1\in\mathscr{S}^{child}[[expr_1]](x) \text{ or } x_1\in\mathscr{E}[[expr_1]](x,\beta(Pos),\beta(Size)),\dots,\\
&\qquad\qquad x_n\in\mathscr{S}^{child}[[expr_n]](x) \text{ or } x_n\in\mathscr{E}[[expr_n]](x,\beta(Pos),\beta(Size))\\
&\qquad \text{such that } (x_1,\dots,x_n)\in\mathscr{I}(pred).
\end{aligned}$$

10. – 13. Boolean connectives and quantification, constants, and variables: obvious. Functions are not defined in Wadler (1999), but the extension is obvious.
14. Context-related functions use the extension of variable bindings by pseudo-variables $Size$ and Pos in rule (6):

$$\begin{aligned}
\mathscr{S}^{any}_{\mathscr{X}}(\mathsf{position}(),x,\beta) &\overset{\text{Def}}{\Leftrightarrow} \beta(Pos) \overset{**}{=} \mathscr{E}[[\mathsf{position}()]](x,\beta(Pos),\beta(Size))\\
\mathscr{S}^{any}_{\mathscr{X}}(\mathsf{last}(),x,\beta) &\overset{\text{Def}}{\Leftrightarrow} \beta(Size) \overset{**}{=} \mathscr{E}[[\mathsf{last}()]](x,\beta(Pos),\beta(Size)).
\end{aligned}$$

Proof of Lemma 4:

Note: A bit sloppy, we write $(x,\beta)\in\mathscr{SB}_{\mathscr{X}}(expr)$ for "$x\in\text{Res}(\mathscr{SB}_{\mathscr{X}}(expr))$ and $\beta\in\text{Bdgs}(\mathscr{SB}_{\mathscr{X}}(expr),x)$".

1. For closed expressions, $x\in\text{Res}(\mathscr{SB}_{\mathscr{X}}(refExpr)) \overset{\text{Def}}{\Leftrightarrow}$

$$x\in\text{Res}(\mathscr{SB}_{\mathscr{X}}(refExpr,\emptyset)) \overset{\text{IH}}{\Leftrightarrow} x\in\mathscr{S}_{\mathscr{X}}(refExpr,\emptyset) \overset{\text{Def.4}}{\Leftrightarrow} x\in\mathscr{S}_{\mathscr{X}}(refExpr).$$

2. Reference expressions are translated into path expressions wrt. a start node:
   - entry points: rooted path

$$\begin{aligned}
(x,\beta)\in\mathscr{SB}_{\mathscr{X}}(/p,Bdgs) &\overset{\text{Def}}{\Leftrightarrow} (x,\beta)\in\mathscr{SB}^{any}_{\mathscr{X}}(p,root,Bdgs)\\
&\overset{\text{IH}}{\Leftrightarrow} x\in\mathscr{S}^{any}_{\mathscr{X}}(p,root,\beta) \text{ and } \beta \text{ completes some } \beta'\in Bdgs \text{ with } \mathsf{free}(/p))\\
&\overset{\text{Def.4}}{\Leftrightarrow} x\in\mathscr{S}^{any}_{\mathscr{X}}(/p,\beta) \text{ and } \beta \text{ completes some } \beta'\in Bdgs \text{ with } \mathsf{free}(/p)).
\end{aligned}$$

   - entry points: constants $c\in\mathscr{V}$ analogously (set c instead of $root$ above).
   - entry points: variables $V\in\mathsf{Var}$. By definition,

$$\begin{aligned}
&(x,\beta)\in\mathscr{SB}_{\mathscr{X}}(V/p,Bdgs) \overset{\text{Def}}{\Leftrightarrow}\\
&\qquad (x,\beta)\in\mathsf{concat}_{x\in\mathscr{A}_{\mathscr{X}}(\mathsf{descendants},root)\downarrow_1}\left(\mathscr{SB}^{any}_{\mathscr{X}}\left(p,x,Bdgs\bowtie\{V/x\}\right)\right)
\end{aligned}$$

   which is exactly the case if there is an $x\in\mathscr{A}_{\mathscr{X}}(\mathsf{descendants},root)\downarrow_1$ such that $(x,\beta)\in(\mathscr{SB}^{any}_{\mathscr{X}}(p,x,Bdgs\bowtie\{V/x\}))$.
   By induction hypothesis, this is equivalent with

$$x\in\mathscr{S}^{any}_{\mathscr{X}}(p,x,\beta) \text{ and } \beta \text{ completes some } \beta'\in Bdgs\bowtie\{V/x\} \text{ with } \mathsf{free}(p)$$

   which is exactly the case if $x=\beta(V)$ and β completes some $\beta'\in Bdgs$ with $\mathsf{free}(V/p)$. By Def. 4, this again is equivalent with

$$x\in\mathscr{S}^{any}_{\mathscr{X}}(V/p,\beta) \text{ and } \beta \text{ completes some } \beta'\in Bdgs \text{ with } \mathsf{free}(V/p).$$

3. Axis step: $(x,\beta) \in \mathcal{SB}^{any}_{\mathcal{X}}(axis :: pattern, z, Bdgs)$

$$\begin{aligned}
&\overset{\text{Def}}{\Leftrightarrow} && (x,\beta) \in \mathcal{SB}^{axis}_{\mathcal{X}}(pattern, z, Bdgs)\\
&\overset{\text{IH}}{\Leftrightarrow} && x \in \mathcal{S}^{axis}_{\mathcal{X}}(pattern, z, \beta)\\
& && \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(pattern)\\
&\overset{\text{Def.4}}{\Leftrightarrow} && x \in \mathcal{S}^{any}_{\mathcal{X}}(axis :: pattern, z, \beta)\\
& && \text{ and } \beta \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(axis :: pattern).
\end{aligned}$$

4. Node test: $(x,\beta) \in \mathcal{SB}^{a}_{\mathcal{X}}(name, z, Bdgs) \overset{\text{Def}}{\Leftrightarrow}$
$$(x,\beta) \in \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,z),\ n=name}(v, \{true\} \bowtie Bdgs)$$
which is exactly the case if $x \in \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,z),\ n=name}(v)$ and $\beta \in Bdgs$ which, by Def. 4 is equivalent with $x \in \mathcal{S}^{a}_{\mathcal{X}}(name, z, \beta)$ and β completes some $\beta' \in Bdgs$ with $\mathsf{free}(name) = \emptyset$. Analogously for $\mathsf{node()}$ and $\mathsf{text()}$.
Variables at nodetest position:
$$(x,\beta) \in \mathcal{SB}^{a}_{\mathcal{X}}(N, z, Bdgs) \overset{\text{Def}}{\Leftrightarrow} (x,\beta) \in \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,z)}(v, \{N/n\} \bowtie Bdgs)$$
which is exactly the case if $x \in \mathsf{list}_{(v,n)\in\mathcal{A}_{\mathcal{X}}(a,z)}(v)$ and $\beta \in \{N/n\} \bowtie Bdgs$ which, by Def. 4 is equivalent with $x \in \mathcal{S}^{a}_{\mathcal{X}}(N, z, \beta)$ and β completes some $\beta' \in Bdgs$ with $\mathsf{free}(N) = \{N\}$.

5. Step with variable binding:
$$\begin{aligned}
&(x,\beta) \in \mathcal{SB}^{a}_{\mathcal{X}}(pattern \rightarrow V, z, Bdgs)\\
&\overset{\text{Def}}{\Leftrightarrow} (x,\beta) \in \mathsf{list}_{(y,\xi)\in\mathcal{SB}^{a}_{\mathcal{X}}(pattern,z,Bdgs)}(y, \xi \bowtie \{V/y\})\\
&\Leftrightarrow \text{ there is a } \beta'' \text{ s.t. } (x,\beta'') \in \mathcal{SB}^{a}_{\mathcal{X}}(pattern, z, Bdgs) \text{ and } \beta = \beta'' \bowtie \{V/x\}.
\end{aligned}$$
By induction hypothesis, this is exactly the case if there is a β'' such that $x \in \mathcal{S}^{a}_{\mathcal{X}}(pattern, z, \beta'')$ and β'' completes some $\beta' \in Bdgs$ with $\mathsf{free}(pattern)$, and $\beta = \beta'' \bowtie \{V/x\}$. Exactly then, since $x = \beta(V)$, by Definition 4, $x \in \mathcal{S}^{a}_{\mathcal{X}}(pattern \rightarrow V, z, \beta)$ and β completes β' with $\mathsf{free}(pattern \rightarrow V) = \mathsf{free}(pattern) \cup \{V\}$.

6. Step Qualifier(s): $(x,\beta) \in \mathcal{SB}^{a}_{\mathcal{X}}(pattern[stepQualifier], z, Bdgs)$
$$\overset{\text{Def}}{\Leftrightarrow} (x,\beta) \in \mathsf{list}_{\substack{(y,\xi)\in\mathcal{SB}^{a}_{\mathcal{X}}(pattern,z,Bdgs),\\ \mathcal{QB}_{\mathcal{X}}(stepQualifier,y,\xi')\neq\emptyset}}(y, \mathcal{QB}_{\mathcal{X}}(stepQualifier, y, \xi') \setminus \{Pos, Size\})$$
for ξ as defined in Definition 10(6). This is exactly the case if (i) there is a β'' s.t. $\beta'' \in \mathcal{QB}_{\mathcal{X}}(stepQualifier, x, \xi')$ and $\beta = \beta'' \setminus \{Pos, Size\}$, and (ii) $(x,\xi) \in \mathcal{SB}^{a}_{\mathcal{X}}(pattern, z, Bdgs)$ i.e. ξ is the corresponding set of variable bindings, and (iii) $\mathcal{QB}_{\mathcal{X}}(stepQualifier, x, \xi') \neq \emptyset$.
The first item is by induction hypothesis equivalent to $\mathcal{Q}_{\mathcal{X}}(stepQualifier, x, \beta'')$ and β'' completes some $\beta' \in \xi'$ with $\mathsf{free}(stepQualifier)$ (*).
The third item is redundant here (it avoids the addition of elements with empty bindings list to the result). Since β'' completes some $\beta' \in \xi'$ with $\mathsf{free}(stepQualifier)$, we know that $\gamma := \beta' \setminus \{Pos, Size\}$ is an element of ξ. Specializing the second item to γ yields $(x,\gamma) \in \mathcal{SB}^{a}_{\mathcal{X}}(pattern, z, Bdgs)$.
By induction hypothesis, $x \in \mathcal{S}^{a}_{\mathcal{X}}(pattern, z, \gamma)$ (**) and γ completes some $\gamma' \in Bdgs$ with $\mathsf{free}(pattern)$. Above, we derived $\gamma = \beta' \setminus \{Pos, Size\}$. Using (*),

since β'' is a completion of β' with $\mathsf{free}(\mathit{stepQualifier})$, completing $\gamma' \in \mathit{Bdgs}$ first to γ (binding $\mathsf{free}(\mathit{pattern})$), then to β' (binding Size and Pos), then to β'' (binding $\mathsf{free}(\mathit{stepQualifier})$), we have $\mathcal{Q}_{\mathcal{X}}(\mathit{stepQualifier}, y, \beta'')$.
From (**), since β'' completes γ, $x \in \mathcal{S}^a_{\mathcal{X}}(\mathit{pattern}, z, \beta'')$ thus by Def. 4, the desired result $x \in \mathcal{S}^a_{\mathcal{X}}(\mathit{pattern}[\mathit{stepQualifier}], z, \mathit{Bdgs})$ for β'' which completes $\gamma' \in \mathit{Bdgs}$ with $\mathsf{free}(\mathit{pattern}[\mathit{stepQualifier}])$.
The argumentation showed the "$\Rightarrow$" direction (which is the more difficult direction since γ must be guessed). "$\Leftarrow$" uses the same relationships and variable bindings.

7. Path: $(x, \beta) \in \mathcal{SB}^a_{\mathcal{X}}(p_1/p_2, z, \mathit{Bdgs})$

$$\begin{aligned}
&\overset{\text{Def}}{\Leftrightarrow} && (x,\beta) \in \mathsf{concat}_{(y,\xi)\in\mathcal{SB}^{any}_{\mathcal{X}}(p_1,z,\mathit{Bdgs})}\mathcal{SB}^a_{\mathcal{X}}(p_2,y,\xi)\\
&\Leftrightarrow && \text{there is an } (y,\xi) \in \mathcal{SB}^{any}_{\mathcal{X}}(p_1,z,\mathit{Bdgs}) \text{ s.t. } (x,\beta) \in \mathcal{SB}^a_{\mathcal{X}}(p_2,y,\xi)\\
&\overset{\text{IH}}{\Leftrightarrow} && \text{there is a } \gamma \in \xi \text{ s.t. there is a } \gamma' \text{ s.t. } x \in \mathcal{S}^a_{\mathcal{X}}(p_2,y,\gamma') \text{ and}\\
& && \gamma' \text{ completes } \gamma \text{ with } \mathsf{free}(p_2).
\end{aligned}$$

For this γ, $(y, \gamma) \in \mathcal{SB}^{any}_{\mathcal{X}}(p_1, z, \mathit{Bdgs})$ and by induction hypothesis again $y \in \mathcal{S}^a_{\mathcal{X}}(p_1, z, \gamma)$ and γ completes some $\beta' \in \mathit{Bdgs}$ with $\mathsf{free}(p_1)$. Thus, also $x \in \mathcal{S}^a_{\mathcal{X}}(p_2, y, \gamma')$ and $y \in \mathcal{S}^a_{\mathcal{X}}(p_1, z, \gamma')$ and by Def. 4, $x \in \mathcal{S}^a_{\mathcal{X}}(p_1/p_2, z, \gamma')$. γ' completes some $\beta' \in \mathit{Bdgs}$ with $\mathsf{free}(p_1) \cup \mathsf{free}(p_2)$.

8. Reference expressions (existential semantics) in step qualifiers:

$$\begin{aligned}
&\beta \in \mathcal{QB}_{\mathcal{X}}(\mathit{refExpr}, z, \mathit{Bdgs}) \overset{\text{Def}}{\Leftrightarrow} \beta \in \textstyle\bigcup_{(y,\xi)\in\mathcal{SB}^{any}_{\mathcal{X}}(\mathit{refExpr},z,\mathit{Bdgs})} \xi\\
&\quad\Leftrightarrow \text{there is a } y \text{ s.t. } (y,\beta) \in \mathcal{SB}^{any}_{\mathcal{X}}(\mathit{refExpr}, z, \mathit{Bdgs})\\
&\quad\overset{\text{IH}}{\Leftrightarrow} y \in \mathcal{S}^{any}_{\mathcal{X}}(\mathit{refExpr}, z, \beta)\\
&\qquad\ \text{and } \beta \text{ completes some } \beta' \in \mathit{Bdgs} \text{ with } \mathsf{free}(\mathit{refExpr})\\
&\quad\overset{\text{Def.4}}{\Leftrightarrow} \mathcal{Q}_{\mathcal{X}}(\mathit{refExpr}, z, \beta) \text{ and } \beta \text{ completes some } \beta' \in \mathit{Bdgs} \text{ with } \mathsf{free}(\mathit{refExpr}).
\end{aligned}$$

9. Built-in equality predicate "=": similar to predicates and variable assigments by $\rightarrow V$. All other predicates: $\beta \in \mathcal{QB}_{\mathcal{X}}(\mathit{pred}(\mathit{arg}_1, \ldots, \mathit{arg}_n), z, \mathit{Bdgs})$

$$\begin{aligned}
&\overset{\text{Def}}{\Leftrightarrow} && \beta \in \bigcup_{(x_i,\xi_i)\in\mathcal{SB}^{any}_{\mathcal{X}}(\mathit{arg}_1,z,\mathit{Bdgs}),\ (x_1,\ldots,x_n)\in\mathcal{I}(\mathit{pred})} \xi_1 \bowtie \ldots \bowtie \xi_n\\
&\Leftrightarrow && \text{there are } (x_1,\xi_1),\ldots,(x_n,\xi_n) \text{ s.t. } (x_i,\xi_i) \in \mathcal{SB}^{any}_{\mathcal{X}}(\mathit{arg}_i, z, \mathit{Bdgs})\\
& && \text{and } (x_1,\ldots,x_n) \in \mathcal{I}(\mathit{pred}) \text{ and } \beta \in \xi_1 \bowtie \ldots \bowtie \xi_n\\
&\Leftrightarrow && (\text{take the right } \beta_i \in \xi_i)\\
& && \text{there are } (x_1,\beta_1),\ldots,(x_n,\beta_n) \text{ s.t. } (x_i,\beta_i) \in \mathcal{SB}^{any}_{\mathcal{X}}(\mathit{arg}_i, z, \mathit{Bdgs})\\
& && \text{and } (x_1,\ldots,x_n) \in \mathcal{I}(\mathit{pred}) \text{ and } \beta = \beta_1 \bowtie \ldots \bowtie \beta_n\\
&\overset{\text{IH}}{\Leftrightarrow} && \text{there are } (x_1,\beta_1),\ldots,(x_n,\beta_n) \text{ s.t. } x_i \in \mathcal{S}^{any}_{\mathcal{X}}(\mathit{arg}_i, z, \beta_i)\\
& && \text{and } \beta_i \text{ extends some } \beta'_i \in \mathit{Bdgs} \text{ with } \mathsf{free}(\mathit{arg}_i)\\
& && \text{and } (x_1,\ldots,x_n) \in \mathcal{I}(\mathit{pred}) \text{ and } \beta = \beta_1 \bowtie \ldots \bowtie \beta_n\\
&\Leftrightarrow && (\text{the join guarantees that } \beta' := \beta'_1 = \ldots = \beta'_n \text{ holds})\\
& && \text{there are } x_1,\ldots,x_n \text{ s.t. } x_i \in \mathcal{S}^{any}_{\mathcal{X}}(\mathit{arg}_i, z, \beta_i)\\
& && \text{and } \beta \text{ extends some } \beta' \in \mathit{Bdgs} \text{ with } \mathsf{free}(\mathit{arg}_1) \cup \ldots \cup \mathsf{free}(\mathit{arg}_n)\\
&\overset{\text{Def.4}}{\Leftrightarrow} && \mathcal{Q}_{\mathcal{X}}(\mathit{pred}(\mathit{arg}_1,\ldots,\mathit{arg}_n), z, \beta)\\
& && \text{and } \beta \text{ completes some } \beta' \in \mathit{Bdgs} \text{ with } \mathsf{free}(\mathit{pred}(\mathit{arg}_1,\ldots,\mathit{arg}_n)).
\end{aligned}$$

10. Negated expressions which do *not contain any free variable*: trivial.
    For negated expressions which contain free variables: note that all variables in $\mathsf{free}(\mathsf{not}\ expr)$ are required to be bound by $Bdgs$ (safety).

$$\begin{aligned}
&\beta \in \mathcal{QB}_{\mathcal{X}}(\mathsf{not}\ expr, z, Bdgs)\\
&\overset{\text{Def}}{\Leftrightarrow} \quad \beta \in Bdgs \text{ and there is no } \beta' \in \mathcal{QB}_{\mathcal{X}}(expr, z, Bdgs) \text{ s.t. } \beta \leqslant \beta'\\
&\overset{\text{IH}}{\Leftrightarrow} \quad \beta \in Bdgs \text{ and there is no } \beta'' \text{ such that}\\
&\qquad \mathcal{Q}_{\mathcal{X}}(expr, z, \beta'') \text{ and } \beta'' \text{ extends } \beta' \text{ with } \mathsf{free}(expr) \text{ and } \beta \leqslant \beta'\\
&\overset{\text{Safety}}{\Leftrightarrow} \quad \beta \in Bdgs \text{ and not } \mathcal{Q}_{\mathcal{X}}(expr, z, \beta)\\
&\overset{\text{Def.4}}{\Leftrightarrow} \quad \beta \in Bdgs \text{ and } \mathcal{Q}_{\mathcal{X}}(\ \mathsf{not}\ expr, z, \beta).
\end{aligned}$$

Conjunction: $\beta \in \mathcal{QB}_{\mathcal{X}}(expr_1\ \mathsf{and}\ expr_2, z, Bdgs)$

$$\begin{aligned}
&\overset{\text{Def}}{\Leftrightarrow} \beta \in \mathcal{QB}_{\mathcal{X}}(expr_1, z, Bdgs) \bowtie \mathcal{QB}_{\mathcal{X}}(expr_2, z, \mathcal{QB}_{\mathcal{X}}(expr_1, z, Bdgs))\\
&\overset{\text{IH}}{\Leftrightarrow} \text{there are } \gamma_1 \in \mathcal{QB}_{\mathcal{X}}(expr_1, z, Bdgs)\\
&\quad \text{and } \gamma_2 \in \mathcal{QB}_{\mathcal{X}}(expr_1, z, \mathcal{QB}_{\mathcal{X}}(expr_1, z, Bdgs)) \text{ s.t. } \mathcal{Q}_{\mathcal{X}}(expr_1, z, \gamma_1)\\
&\quad \text{and } \gamma_1 \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(expr_1) \text{ and}\\
&\quad \mathcal{Q}_{\mathcal{X}}(expr_2, z, \gamma_2) \text{ and } \gamma_1 \text{ completes some } \gamma'' \in \mathcal{QB}_{\mathcal{X}}(expr_1, z, Bdgs)\\
&\quad \text{with } \mathsf{free}(expr_2) \text{ and } \beta = \gamma_1 \bowtie \gamma_2.
\end{aligned}$$

$$\begin{aligned}
&\Leftrightarrow \quad (\text{join condition: } \gamma_1 = \gamma'' \leqslant \gamma_2)\ \ \mathcal{Q}_{\mathcal{X}}(expr_1, z, \gamma_2) \text{ and } \mathcal{Q}_{\mathcal{X}}(expr_2, z, \gamma_2)\\
&\qquad \text{and } \gamma_2 \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(expr_1) \cup \mathsf{free}(expr_2)\\
&\Leftrightarrow \quad \mathcal{Q}_{\mathcal{X}}(expr_1 \mathsf{and}\ expr_2, z, \gamma_2)\\
&\qquad \text{and } \gamma_2 \text{ completes some } \beta' \in Bdgs \text{ with } \mathsf{free}(expr_1) \cup \mathsf{free}(expr_2).
\end{aligned}$$

11. – 13.: trivial. (safety for variables; functions similar to predicates).
14. Context-related functions use the extension of variable bindings by pseudo-variables $Size$ and Pos in rule (6):

$$\begin{aligned}
&(x, \beta) \in \mathcal{SB}^{any}_{\mathcal{X}}(\mathsf{position}(), z, Bdgs)\\
&\overset{\text{Def}}{\Leftrightarrow} \quad (x, \beta) \in \mathsf{list}_{\beta \in Bdgs}(\beta(Pos), \{\beta' \in Bdgs \mid \beta(Pos) = \beta'(Pos)\})\\
&\Leftrightarrow \quad \beta(Pos) = x \text{ for some } \beta \in Bdgs\\
&\Leftrightarrow \quad x \in \mathcal{S}^{any}_{\mathcal{X}}(\mathsf{position}(), z, \beta) \text{ for some } \beta \in Bdgs\\
&\Leftrightarrow \quad x \in \mathcal{S}^{any}_{\mathcal{X}}(\mathsf{position}(), z, \beta)\\
&\qquad \text{and } \beta \text{ completes some } \beta' \in Bdgs \text{ by } \mathsf{free}(\mathsf{position}()) \text{ (which is empty).}
\end{aligned}$$

Analogously for $\mathsf{last}()$.

Proof of Lemma 7: Structural induction.

- entry case (using $\beta = \beta'$): $(\mathcal{X}, \beta) \models /p \overset{\text{Def.5}}{\Leftrightarrow} (\mathcal{S}_{\mathcal{X}}(/p, \beta)) \neq \emptyset$

$$\begin{aligned}
&\overset{\text{Def.4}}{\Leftrightarrow} \quad (\mathcal{S}_{\mathcal{X}}(p, root, \beta)) \neq \emptyset \overset{\text{Def.4}}{\Leftrightarrow} (\mathcal{S}_{\mathcal{X}}(root/p, \beta)) \neq \emptyset \overset{\text{Def.5}}{\Leftrightarrow} (\mathcal{X}, \beta) \models root/p\\
&\overset{\text{IH}}{\Leftrightarrow} \quad (\mathcal{X}, \beta) \models \mathsf{atomize}(root/p) \overset{\text{Def}}{\Leftrightarrow} (\mathcal{X}, \beta) \models \mathsf{atomize}(/p).
\end{aligned}$$

- Paths are resolved into steps and step qualifiers are isolated (the case where a don't care variable is introduced is shown; w.l.o.g., *path* is an absolute path expression)

$(\mathcal{X},\beta) \models path/axis :: nodetest[stepQualifier]\ /remainder$
$\Leftrightarrow\ \mathcal{S}_{\mathcal{X}}(path/axis :: nodetest[stepQualifier]\ /remainder, \beta) \neq \emptyset$
$\Leftrightarrow\ \mathcal{S}_{\mathcal{X}}(path/axis :: nodetest[stepQualifier]\ /remainder, root, \beta) \neq \emptyset$
$\Leftrightarrow\ \mathsf{concat}_{y \in \mathcal{S}^a_{\mathcal{X}}(path/axis::nodetest[stepQualifier],root,\beta)}(\mathcal{S}^{any}_{\mathcal{X}}(remainder, y, \beta)) \neq \emptyset$
$\Leftrightarrow$ there is a node $v \in \mathcal{S}^a_{\mathcal{X}}(path/axis :: nodetest[stepQualifier], root, \beta)$ s.t. $\mathcal{S}^{any}_{\mathcal{X}}(remainder, v, \beta) \neq \emptyset$
$\Leftrightarrow$ there is a node $v \in \mathsf{list}_{y \in \mathcal{S}^a_{\mathcal{X}}(path/axis::nodetest,x,\beta)}(y \mid \mathcal{Q}_{\mathcal{X}}(stepQualifier, y, \beta))$ s.t. $\mathcal{S}^{any}_{\mathcal{X}}(remainder, v, \beta) \neq \emptyset$
$\Leftrightarrow$ there is a node v s.t. $v \in \mathcal{S}^a_{\mathcal{X}}(path/axis :: nodetest, x, \beta)$ and $\mathcal{Q}_{\mathcal{X}}(stepQualifier, v, \beta)$ and $\mathcal{S}^{any}_{\mathcal{X}}(remainder, v, \beta) \neq \emptyset$
$\Leftrightarrow$ there is a node v s.t. $v \in \mathcal{S}^a_{\mathcal{X}}(path/axis :: nodetest \rightarrow \_X, x, \beta^v{}_{\_X})$ and $\mathcal{Q}_{\mathcal{X}}(V[stepQualifier], v, \beta^v{}_{\_X})$ and $\mathcal{S}^{any}_{\mathcal{X}}(V/remainder, v, \beta^v{}_{\_X}) \neq \emptyset$
$\Leftrightarrow$ there is a node v s.t. $v \in \mathcal{S}^a_{\mathcal{X}}(path[axis :: nodetest \rightarrow \_X], x, \beta^v{}_{\_X})$ and $\mathcal{Q}_{\mathcal{X}}(V[stepQualifier], v, \beta^v{}_{\_X})$ and $\mathcal{S}^{any}_{\mathcal{X}}(V/remainder, v, \beta^v{}_{\_X}) \neq \emptyset$
$\Leftrightarrow$ there is a node v s.t. $\mathcal{S}^a_{\mathcal{X}}(path[axis :: nodetest \rightarrow \_X], x, \beta^v{}_{\_X}) \neq \emptyset$ and $\mathcal{Q}_{\mathcal{X}}(V[stepQualifier], v, \beta^v{}_{\_X})$ and $\mathcal{S}^{any}_{\mathcal{X}}(V/remainder, x, \beta^v{}_{\_X}) \neq \emptyset$
$\Leftrightarrow$ there is a node v s.t. $\mathcal{Q}_{\mathcal{X}}(path[axis :: nodetest \rightarrow \_X], \beta^v{}_{\_X})$ and $\mathcal{Q}_{\mathcal{X}}(V[stepQualifier], \beta^v{}_{\_X})$ and $\mathcal{Q}_{\mathcal{X}}(V/remainder, \beta^v{}_{\_X})$

$\overset{\text{IH}}{\Leftrightarrow}$ there is a node v s.t. $\mathcal{Q}_{\mathcal{X}}(\mathsf{atomize}(path[axis :: nodetest \rightarrow \_X]), \beta^v{}_{\_X})$ and $\mathcal{Q}_{\mathcal{X}}(\mathsf{atomize}(V[stepQualifier]), \beta^v{}_{_X})$and $\mathcal{Q}_{\mathcal{X}}(\mathsf{atomize}(V/remainder), \beta^v{}_{\_X})$
$\Leftrightarrow$ there is a node v s.t. $\mathcal{Q}_{\mathcal{X}}(\mathsf{atomize}(\ldots), \beta^v{}_{\_X})$.

- Conjunctions in step qualifiers: obvious.
- Predicates in step qualifiers: W.l.o.g., consider a unary predicate with a relative argument expression:

$(\mathcal{X},\beta) \models V[pred(expr)]$
$\Leftrightarrow\ \mathcal{S}_{\mathcal{X}}(V[pred(expr)], \beta(V), \beta) \neq \emptyset$
$\Leftrightarrow\ \mathsf{list}_{y \in \mathcal{S}^a_{\mathcal{X}}(V,\beta(V),\beta)}(y \mid \mathcal{Q}_{\mathcal{X}}(pred(expr), y, \beta)) \neq \emptyset$
$\Leftrightarrow$ $(\beta(V)$ is the only element in $\mathcal{S}^a_{\mathcal{X}}(V, \beta(V), \beta))$ s.t. $\mathcal{Q}_{\mathcal{X}}(pred(expr), \beta(V), \beta)$
$\Leftrightarrow$ there is an $x \in \mathcal{S}_{\mathcal{X}}(expr, \beta(V), \beta)$ such that $pred(x) \in \mathcal{X}$
$\Leftrightarrow$ there is an x s.t. $x \in \mathcal{S}_{\mathcal{X}}(V/expr \rightarrow \_X, root, \beta^x{}_{\_X})$ and $(\mathcal{X}, \beta^x{}_{\_X}) \models pred(\_X)$
$\Leftrightarrow$ there is an x s.t. $(\mathcal{X}, \beta^x{}_{\_X}) \models V/expr \rightarrow \_X$ and $(\mathcal{X}, \beta^x{}_{\_X}) \models pred(\_X)$
$\overset{\text{IH}}{\Leftrightarrow}$ there is an x s.t. $(\mathcal{X}, \beta^x{}_{\_X}) \models \mathsf{atomize}(V/expr \rightarrow \_X)$ and $(\mathcal{X}, \beta^x{}_{\_X}) \models pred(\_X)$
$\Leftrightarrow$ there is an x s.t. $(\mathcal{X}, \beta^x{}_{\_X}) \models \mathsf{atomize}(V[pred(expr)])$.

- Predicate atoms: analogous.

Acknowledgments

Most of this work has been done when I was a member of the database group at Freiburg University. I want to thank my former colleagues during that time: Lule Ahmedi, Matthias Ihle, Georg Lausen, Pedro Marrón, Martin Weber, and Fang Wei.

References

Abiteboul, S., Quass, D., McHugh, J., Widom, J. and Wiener, J. 1997. The Lorel query language for semistructured data. *Journal on Digital Libraries (JODL) 1*, 1, 68–88.

Baru, C., Gupta, A., Ludäscher, B., Marciano, R., Papakonstantinou, Y., Velikhov, P. and Chu, V. 1999. XML-based information mediation with MIX. *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 597–599.

Baumgartner, R., Flesca, S. and Gottlob, G. 2001a. The Elog web extraction language. *Intl. Conference on Logic Programming and Automated Reasoning (LPNMR): Lecture Notes in Computer Science 2250*, pp. 548–560. Springer.

Baumgartner, R., Flesca, S. and Gottlob, G. 2001b. Visual web information extraction with Lixto. *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 119–128.

Bry, F. and Schaffert, S. 2002. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. *Intl. Conference on Logic Programming (ICLP)*, pp. 255–270.

Buneman, P., Davidson, S., Hillebrandt, G. and Suciu, D. 1996. A query language and optimization techniques for unstructured data. *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 505–516. Montreal, Canada.

Buneman, P., Fernandez, M. and Suciu, D. 2000. UnQL: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal 9*, 76–110.

Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S. and Tanca, L. 1999. XML-GL: a graphical language for querying and restructuring XML documents. *Proceedings 8th International World Wide Web Conference (WWW 8)*, pp. 1171–1187.

Christophides, V., Cluet, S. and Siméon, J. 2000. On wrapping query languages and efficient XML integration. *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 141–152.

Clark, J. 1998. XT: an implementation of XSL Transformations. `http://www.jclark.com/xml/xt.html`.

Cluet, S., Delobel, C., Siméon, J. and Smaga, K. 1999. Your mediators need data conversion. *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 177–188.

Comai, S., Damiani, E. and Fraternali, P. 2001. Computing graphical queries over XML data. *ACM Transactions on Information Systems (TOIS) 19*, 4, 371–430.

Consens, M. and Mendelzon, A. 1990. GraphLog: a visual formalism for real life recursion. *ACM Symposium on Principles of Database Systems (PODS)*, pp. 404–416.

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

Deutsch, A., Fernandez, M. and Suciu, D. 2000. Storing semistructured data with STORED. *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 431–442.

DOM-W3C. 1998. Document object model (DOM). `http://www.w3.org/DOM/`.

Fernandez, M., Florescu, D., Levy, A. and Suciu, D. 1997. A query language for a web-site management system. *SIGMOD Record 26*, 3, 4–11.

Fernandez, M., Siméon, J. and Wadler, P. 1999. XML query languages: Experiences and exemplars. draft manuscript, communication to the XML Query W3C Working Group. `http://www-db.research.bell-labs.com/user/simeon/xquery.ps`.

Fernandez, M. F., Florescu, D., Kang, J., Levy, A. Y. and Suciu, D. 1998. Catching the boat with Strudel: Experiences with a web-site management system. *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 414–425.

Florescu, D. and Kossmann, D. 1999. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report 3684, INRIA.

Florid. 1998. Florid homepage. `http://www.informatik.uni-freiburg.de/~dbis/florid/`.

Frohn, J. 1998. Magic-Set Transformation in deduktiven, objektorientierten Datenbanksprachen. PhD thesis, Institut für Informatik, Universität Freiburg.

Frohn, J., Lausen, G. and Uphoff, H. 1994. Access to objects by path expressions and rules. *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 273–284.

Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Vassalos, V. and Widom, J. 1997. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems 8*, 2, 117–132.

Goldman, R., McHugh, J. and Widom, J. 1999. From semistructured data to XML: Migrating the Lore data model and query language. *WebDB 1999*, pp. 25–30.

Hosoya, H. and Pierce, B. C. 2000. Xduce: A typed XML processing language. *WebDB 2000*, pp. 111–116.

Kifer, M. and Lausen, G. 1989. F-Logic: A higher-order language for reasoning about objects, inheritance and scheme. *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 134–146.

Kifer, M., Lausen, G. and Wu, J. 1995. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM 42*, 4, 741–843.

Lakshmanan, L. V. S., Sadri, F. and Subramanian, I. N. 1996. SchemaSQL – a language for interoperability in relational multi-database systems. *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 239–250.

Lehti, P. 2001. Design and implementation of a data manipulation processor for an XML query language. MS thesis, Technische Universität Darmstadt.

Ludäscher, B., Himmeröder, R., Lausen, G., May, W. and Schlepphorst, C. 1998. Managing semistructured data with Florid: A deductive object-oriented perspective. *Information Systems 23*, 8, 589–612.

May, W. 1999. Information extraction and integration with Florid: The Mondial case study. Technical Report 131, Universität Freiburg, Institut für Informatik. Available from `http://www.dbis.informatik.uni-goettingen.de/Mondial/`.

May, W. 2001a. Habilitation thesis. PhD thesis, Universität Freiburg. Available from `http://www.dbis.informatik.uni-goettingen.de/lopix/`.

May, W. 2001b. Information integration in XML: The Mondial case study. Technical Report. Available from `http://www.dbis.informatik.uni-goettingen.de/lopix/lopix-mondial.html`.

May, W. 2001c. LoPiX: A system for XML data integration and manipulation. *Intl. Conference on Very Large Data Bases (VLDB), Demonstration Track*, pp. 707–708.

May, W. 2001d. The lopix system. `http://www.dbis.informatik.uni-goettingen.de/lopix/`.

May, W. 2001e. The Mondial database. `http://www.dbis.informatik.uni-goettingen.de/Mondial/`.

May, W. 2002. A rule-based querying and updating language for XML. *Workshop on Databases and Programming Languages (DBPL 2001): Lecture Notes in Computer Science 2397*, pp. 165–181.

May, W. and Behrends, E. 2001. On an XML data model for data integration. In *Intl. Workshop on Foundations of Models and Languages for Data and Objects (FMLDO 2001).*

McHugh, J., Abiteboul, S., Goldman, R., Quass, D. and Widom, J. 1997. Lore: A database management system for semistructured data. *SIGMOD Record 26*, 3, 54–66.

Przymusinski, T. C. 1988. On the declarative semantics of deductive databases and logic programs. In: J. Minker, Ed. *Foundations of Deductive Databases and Logic Programming*, pp. 191–216. Morgan Kaufmann.

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

Shanmugasundaram, J., Gang, H., Tufte, K., Zhang, C., Witt, D. J. D. and Naughton, J. Relational databases for querying XML documents: Limitations and opportunities. *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 302–314.

Software AG. 2001. Quip: An xquery implementation. `http://www.softwareag.com/developer/quip/`.

Tatarinov, I., Ives, Z. G., Halevy, A., and Weld, D. 2001. Updating xml. *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 133–154.

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

XMLInf. 1999. XML information set. `http://www.w3.org/TR/XML-infoset`.

XMQ-A. 2001. XML Query Algebra. `http://www.w3.org/TR/query-algebra`.

XMQ-D. 2001. XML Query Data Model. `http://www.w3.org/TR/query-datamodel`.

XPath. 1999. XML Path Language (XPath) version 1.0: 1999. `http://www.w3.org/TR/xpath`.

XPQOF. 2001. XQuery 1.0 and XPath 2.0 Functions and Operators. `http://www.w3.org/TR/xquery-operators`.

XQFS. 2001. XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics`.

XQuery. 2001. XQuery: A Query Language for XML. `http://www.w3.org/TR/xquery`.

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