A Typed Higher-Order Calculus for Querying XML Databases

Qing Wang **Klaus-Dieter Schewe**

Massey University, Information Science Research Centre
Private Bag 11 222, Palmerston North, New Zealand, email: [q.q.wang|k.d.schewe]@massey.ac.nz

Abstract

As the eXtensible Markup Language (XML) is about to emerge as a new standard for databases, the problem of providing solid logical grounds for XML query languages arises. For the relational data model first-order logic, i.e. the Relational Calculus turned out to be an intuitive basic approach to provide these foundations. For XML, however, it is necessary to deal with ordered trees. In this paper the problem is approached by viewing XML as a data model based on complex objects that are arranged in a class hierarchy. This results in the natural development of a higher-order type system for XML data, and henceforth a higher-order predicate typed logic, the XML calculus (XMLC). The paper presents the basics of the XML object model (XOM), the syntacs and semantics of XMLC, and discusses the expressiveness of the language by means of representative important query samples.

Keywords: query language, higher-order logic, eXtensible Markup Language, type system, object model.

1 Introduction

Over the last five years the eXtensible Markup Language (XML) (Abiteboul, Buneman & Suciu 2000) has attracted the attention of many database researchers. XML has been taken up as a novel data model based on trees rather than flat relations, and a lot of effort has been put into defining suitable and expressive query languages such as XQuery (World Wide Web Consortium 2005*b*), LOREL (Abiteboul, Quass, McHugh, Widom & Wiener 1997) and XML-QL (Deutsch, Fernandez, Florescu, Levy & Suciu 1999) – just to mention a few. Among these XQuery is emerging as a de facto standard in practice in the same way as SQL is the accepted standard for relational databases. From the intensity of the research and the interest in practice it can be concluded that XML is about to emerge as a new standard for databases. With this the problem of providing solid logical grounds for XML query languages arises.

For the relational data model first-order logic, i.e. the Relational Calculus (Codd 1970, Codd 1972) turned out to be an intuitive basic approach to provide these foundations. It is also the basis of various extensions towards fixed-point logics (Abiteboul, Hull & Vianu 1995) and complex objects (Bancilhon & Khoshafian 1986). In order to determine also an intuitive, calculus-like language for XML it is necessary to deal with ordered trees. In this case types as in Church's Simple Theory of Types (Church 1940) naturally come in, so we do not expect the underlying logic to be still first-order.

Though this poses more challenging problems than in the relational case, it is possible to build on experience that was already gained in using higher-order logic for the purpose of querying databases. This includes logics with higher-order syntax and first-order semantics such as F-logic (Kifer & Lausen 1989) and HiLog (Chen, Kifer & Warren 1993, Chen, Kifer & Warren 1989), logics with first-order syntax and higher-order semantics such as LDL (Beeri, Naqvi, Ramakrishnan, Shmueli & Tsur 1987, Naqvi & Tsur 1989), and logics with higher-order syntax and higher-order semantics such as COL (Abiteboul & Grumbach 1990). In case of first-order semantics on the top of a higher-order syntax some special mechanisms need to be built into the interpretation. One approach is to add mutual reflection between formulae and terms using typed λ-expressions, which has been adopted in λ-Prolog (Nadathur 1987, Miller & Nadathur 1986).

In this paper the problem is approached by viewing XML as a data model based on complex objects that are arranged in a class hierarchy. This results in the natural development of a higher-order type system for XML data, and henceforth a higher-order predicate typed logic, the XML calculus (XMLC). Our approach achieves the logic reflection by exploiting identifier and value semantics of objects, i.e. object identifiers function as first-order abstractions that encapsulate underlying higher-order structures.

In Section 2 we present the basics of the XML object model (XOM). The model follows ideas from object bases, in which identifier and value semantics of objects are formalized. Moreover, XML database instances and schemata are formalized along with the notions of class schema and class tuple. Some insights are given into class hierarchy and object order. Based on this model, the purely declarative language XMLC is introduced in Section 3. We describe the formal syntax and semantics of the language. This is followed by a discussion of expressiveness through examples in Section 4. This is meant to illustrate crucial language features and key characteristics that are required for querying XML data. We conclude with a brief summary and outlook in Section 5. All sample XML documents used in this paper come from (World Wide Web Consortium 2005*a*) with some modifications.

2 The XML Object Model

In this section we present the XML Object Model (XOM), a data model based on tree structures and object units. This model supports a rich description of XML data at a rather straightforward and relatively high abstraction level. XOM will serve as a formal specification for establishing XMLC.

2.1 Objects and Object Types

To precisely describe atomic values and objects, we adopt two respective concepts, atomic types and class names, and use the notations $\mathbb{A}$ for a countably infinite set of atomic types, $\mathbb{C}$ for a countably infinite set of class names.

Each object in the XOM contains some attributes, which can be categorized into two kinds: value attributes that are associated with atomic types and object attributes that are associated with class names. Each value attribute is associated with a *value domain*, expressed by dom_V, the set of all possible values that the value attribute can obtain, and each object attribute is associated with an *object domain*, expressed by dom_O, the set of all possible objects that the object attribute can obtain. For convenience, we let $\mathcal{D} = \{dom_{V_i}\}_{i\in[1,n]}$ be a fixed family of value domains $\{dom_{V_1}, ..., dom_{V_n}\}$, $\mathcal{I} = \{dom_{O_i}\}_{i\in[1,n]}$ be a fixed family of object domains $\{dom_{O_1}, ..., dom_{O_n}\}$, and $\mathcal{O}$ be a fixed set of objects, respectively.

XOM considers objects using a both simple and uniform expression which is formally defined in the following definition.

Definition 2.1 An *object type OT* consists of

1. a class name $nam(OT) \in \mathbb{C}$;
2. a finite, non-empty set of attributes $attr(OT) = \{A_1, ..., A_n\}$, where $A_k \in \mathbb{A} \cup \mathbb{C}$ for $k \in [1, n]$;
3. a *domain* assignment $Dom : nam(OT) \cup attr(OT) \to \mathcal{D} \cup \mathcal{I}$ which associates with the class name $c = nam(OT)$ its domain $Dom(c)$ or each attribute $A \in attr(OT)$ its domain $Dom(A)$.

The expression $OT = nam(OT)[attr(OT)]$ is used to denote an object type OT in a class named $nam(OT)$ having a finite set of attributes $attr(OT)$.

Definition 2.2 An *object* of type OT is denoted by an expression $c(i, u)$ where $c = nam(OT)$ is the class name, $i \in Dom(c)$ is an object identifier, and u is an object value, which takes the form $[A_1 : t_1, ..., A_n : t_n]$ with $\{A_1, ..., A_n\} = attr(OT)$ and $t_k \in Dom(A_k)$ for $k \in [1, n]$.

An object may be classified as being either atomic or complex depending on its attributes. Atomic objects have only value attributes, while complex objects have at least one object attribute. Moreover, each object must belong to exactly one class. That is, given two object types OT_1 and OT_2 with $nam(OT_1) = c_1$, $nam(OT_2) = c_2$ and $c_1 \neq c_2$, then it implies that $Dom(c_1) \cap Dom(c_2) = \emptyset$. Indeed, classes can further be grouped into atomic classes and complex classes. An atomic class contains only atomic objects, while a complex class contains at least one complex object. The fundamental distinction between atomic classes and complex classes depends on whether object identifiers as structure primitives are involved in object values within a particular class.

To capture the variant structures of XML data in a precise mathematical way, the notion of *object value type* is developed for describing heterogeneous structures embedded in objects, i.e. a function τ mapping from an object o to its object value type $\tau(o)$ containing a set of distinct attribute names occurring in its value, such that, if the given object value of o is $u = [A_1 : t_1, ..., A_n : t_n]$, then $\tau(o) = [A'_1, ..., A'_m]$ ($n \geq m$), where $\bigwedge_{1\leq i<j\leq m} A'_i \neq A'_j$ and $A'_i, A'_j \in \{A_k \mid k \leq n\} \cup \{\{A_k\} \mid k \leq n\}$. Here $\{A\}$ denotes a set type, i.e. $Dom(\{A\}) = \{\{v_1, \ldots, v_k\} \mid k \in \mathbb{N}, v_i \in Dom(A)$ for all $1 \leq i \leq k\}$. The following example illustrates this notion.

```
<book>
  <title>"Internet"</title>
  <author>"Tom"</author>
  <author>"Peter"</author>
  <author>"Song"</author>
  <section id="1" difficulty="easy" >
    <title>"HTML"</title><p>"T1"</p>
    <section>
      <title>"Links"</title><p>"T2" </p>
    </section>
    <section>
      <title>"Elements"</title><p>"T3"</p>
      <figure height="400" width="400">
        <title>"g1"</title>
      </figure>
      <p>"T4"</p>
    </section>
  </section>
  <section id="2" difficulty="medium" >
    <title>"XML"</title><p>"T5"</p>
    <figure>
      <title>"g2"</title>
    </figure>
    <p>"T6"</p>
    <section>
      <title>"Documents"</title><p>"T7"</p>
    </section>
    <section>
      <title>"Databases"</title><p>"T8"</p>
      <section>
        <title>"Constraints"</title><p>"T9"</p>
      </section>
    </section>
  </section>
</book>
```

Figure 1: book.xml

Example 2.1 For the object $o = c(i, [A_1 : t_1, A_2 : t_2, A_3 : t_3, A_2 : t_4])$ we obtain $\tau(o) = [A_1, \{A_2\}, A_3]$.

More precisely, XML data are treated as ordered trees. Although order is not an issue receiving significant attention in most situations, it should be taken into consideration as a distinguished feature of XML data. Under object paradigms, we can use a succinct expression $\{[i, o_i] : 1 \leq i \leq n\}$, where i is the order of object o_i, to capture a sequence of objects $[o_1, ..., o_n]$. One observation about this expression is that object order and object identifiers are at least similar in representations with respect to objects. Furthermore, we find out that both object identifiers and orders are invisible for users. Individual object identifiers or orders themselves are meaningless, and only interrelationships among them are important. Based on this intuition, rather than introduce an additional property to capture object orders, we use object identifiers as an order primitive in the XOM. An extra accompanying functionality with object identifiers being order primitives is that object identifiers can be involved in computations.

Example 2.2 Assume that there are two objects $o_1 = (i_1, u_1)$ and $o_2 = (i_2, u_2)$, then we can use the expression "$i_1 < i_2$" as a constraint on these two objects, which simply means that the order of o_1 is less than the order of o_2.

At the implementation level, the issue regarding order could be addressed by assigning a distinct number to each object identifier in a desirable sequence. For the sake of update concerns that might be involved, there is a gap left between two successive object identifiers so that inserted objects can be assigned a proper number as identifers within the gap.

To facilitate the formalization, three assignments are defined in advance.

Definition 2.3 A *name projection* is a function σ restricting each object expression $o = c(i, u)$ to the object's class name c.

A *value projection* is a function π restricting each object expression $o = c(i, u)$ to the object's value u.

A *structure projection* is a function η restricting each object expression $o = c(i, u)$ to the finite set of object identifiers occurred in the object's value u.

Objects in the XOM can be interpreted under different semantics. We introduce identifier and value semantics of objects in the following, which indeed provide a flexible interpretation for class predicates developed in the XMLC discussed in Section 3.

Definition 2.4 Let $\mathcal{U}$ be a fixed set of object values.

An object (in identifier semantics) o of type OT is an injective mapping $z : \mathcal{O} \rightarrow \mathcal{I}$ with $z(o) \in Dom(nam(OT))$, i.e. for any $i_1, i_2 \in \mathcal{I}$, $z(o_1) = i_1$ and $z(o_2) = i_2$, if $i_1 = i_2$, then $o_1 = o_2$ must hold.

An object (in value semantics) o of type OT is a mapping $f : \mathcal{O} \rightarrow \mathcal{U}$ with $f(o) = \{A_i : t_i | A_i \in attr(OT), t_i \in Dom(A_i)\}$ on a condition that if $f(o_1) = f(o_2)$ and $\eta(o_1) = \eta(o_2) \neq \emptyset$, then $o_1 = o_2$ must hold.

For objects in identifier semantics, object identifers can uniquely identify the objects, while for objects in value semantics, only complex objects can be uniquely identified because atomic objects may coincide on their values.

2.2 Class Schemata and Tuples

Unlike the rigid structures of relations in the relational data model, XML data is well known for having an irregular and flexible structure. However, a uniform expression for XML data is very appealing from the data manipulation point of view. In this section, we introduce the concepts of class schema and class tuple to achieve this purpose.

The fundamental idea is, for a set of objects collected into a class, to obtain class schemata by generalizing their object value types using optional superscript characters $\{?, +, *\}$ (World Wide Web Consortium 2004), and then treating their object values appropriately as class tuples over the specified class schema. Due to optional characters being additionally introduced into class schemata, it is necessary to extend the concepts of value attribute and object attribute domains to the concept of class attribute domain by capturing this feature. To distinguish from previous expressions we use $dom(A)$ as a unified representation for a domain of a class attribute A, which might either be a value attribute or an object attribute, and $edom(A^f)$ to express a domain of a class attribute A with an optional character f. For simplicity and uniformity of formalization we assume that all class attributes without optional characters from $\{?, +, *\}$ have the superscript character 1. For instance, the class attribute A is equivalent to the class attribute A^1. Now we can get

$edom(A^1) = dom(A)$;

$edom(A^?) = dom(A) \cup \{\lambda\}$;

$edom(A^*) = \mathcal{P}(dom(A))$, where $\mathcal{P}$ denotes powersets;

$edom(A^+) = \mathcal{P}(dom(A)) - \{\emptyset\}$.

Similar to the relational data model there are two alternatives for specifying class schemata and tuples using either an anonymous or a named perspective.

Definition 2.5 An *anonymous class schema* of a class with name c consists of a finite set of distinct implicit attribute names with superscript characters $\{A_1^{f_1}, ..., A_n^{f_n}\}$ with $f_i \in \{1, ?, +, *\}(1 \leq i \leq n)$ that have a fixed order, together with an assignment of domains $edom(A_i^{f_i})$ for $i = 1, \ldots, n$.

A *named class schema* of a class with name c consists of a finite set of distinct explicit attribute names with superscript characters $\{A_1^{f_1}, ..., A_n^{f_n}\}$ with $f_i \in \{1, ?, +, *\}(1 \leq i \leq n)$, together with an assignment of domains $\{A_1^{f_1}, ..., A_n^{f_n}\} \rightarrow \bigcup_{1 \leq i \leq n} edom(A_i^{f_i})$ such that $A_i^{f_i} \mapsto edom(A_i^{f_i})$.

Correspondingly, a class tuple over a particular class schema can be defined under the two perspectives.

Definition 2.6 A *class tuple* t over an anonymous class schema having implicit attributes with superscript characters $\{A_1^{f_1}, ..., A_n^{f_n}\}$ is a mapping: $\{1, ..., n\} \rightarrow \{v_1, ..., v_n\}$, where $v_i \in edom(A_i^{f_i})$ for $1 \leq i \leq n$.

A *class tuple* t over a named class schema having explicit attributes with superscript characters $\{A_1^{f_1}, ..., A_n^{f_n}\}$ is a mapping: $\{A_1^{f_1}, ..., A_n^{f_n}\} \rightarrow \{A_1 : v_1, ..., A_n : v_n\}$, where $v_i \in edom(A_i^{f_i})$ for $1 \leq i \leq n$.

Example 2.3 Consider the class with name *section* in the XML document book.xml (see Figure 1). In this case we obtain that the named class schema for this class is $[@id^?, @difficult^?, title, p^+, figure^?, section^*]$, and a set of objects contained in this class is $\{i_6, i_{11}, i_{14}, i_{24}, i_{34}, i_{37}, i_{40}\}$. Furthermore, the corresponding tuples can be represented using the named perspective as follows:

$$[@id^? : i_7, @difficulty^? : i_8, title : i_9, p^+ : \{i_{10}\}, figure^? : \lambda, section^* : \{i_{11}, i_{14}\}];$$
$$[@id^? : \lambda, @difficulty^? : \lambda, title : i_{12}, p^+ : \{i_{13}\}, figure^? : \lambda, section^* : \emptyset];$$
$$[@id^? : \lambda, @difficulty^? : \lambda, title : i_{15}, p^+ : \{i_{16}, i_{23}\}, figure^? : i_{17}, section^* : \emptyset];$$
$$[@id^? : i_{25}, @difficulty^? : i_{26}, title : i_{27}, p^+ : \{i_{28}, i_{33}\}, figure^? : i_{29}, section^* : \{i_{34}, i_{37}\}];$$
$$[@id^? : \lambda, @difficulty^? : \lambda, title : i_{35}, p^+ : \{i_{36}\}, figure^? : \lambda, section^* : \emptyset];$$
$$[@id^? : \lambda, @difficulty^? : \lambda, title : i_{38}, p^+ : \{i_{39}\}, figure^? : \lambda, section^* : \{i_{40}\}];$$
$$[@id^? : \lambda, @difficulty^? : \lambda, title : i_{41}, p^+ : \{i_{42}\}, figure^? : \lambda, section^* : \emptyset].$$

One observation is that, although both atomic classes and complex classes can be described with these features, the notions of class schema and class tuple only make sense for complex classes. The reason is that atomic classes as a collection of value wrappers can simply be treated as a bridge between the object base and the value base, and no complex structures are involved.

2.3 XML Databases

From a database point of view XML data can be represented by tree structures (we do not consider ref-

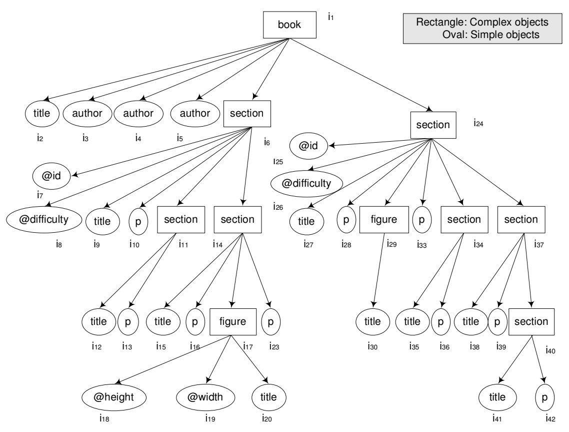


Figure 2: An XML data graph

erences here). Our interest here is to discuss XML database instances and schemata on the basis of the concepts introduced in this section. For the sake of simplicity, only element, attribute and text nodes are considered in the formalization.

In the XOM an XML document can be regarded as an XML data graph, and correspondingly an XML database instance is a collection of XML data graphs. Let e, a and p represent element, attribute and text nodes in XML data, respectively, and let o, d represent objects and atomic values in the XOM. In addition, we use the expression $name(x)$ referring to the name of element or attribute node, $content(x)$ referring to the content of text or attribute node, and $subelement(x)$ referring to the identifier of an object corresponding to some subelement node of element node. Then the mapping relationships between XML data and objects are presented in the following:

- Each element node e will be modelled as an object o with $\sigma(o) = name(e)$ and $\pi(o) = [A_1 : subelement_1(e), \ldots, A_m : subelement_m(e), A_1^{'} : content(p_1), \ldots, A_n^{'} : content(p_n)]$, where $m \geq 0$, $n \geq 0$, $A_i \in \mathbb{C}$ for $i \in [1, m]$ and $A_j^{'} \in \mathbb{A}$ for $j \in [1, n]$. Here $subelement_1(e), \ldots, subelement_m(e)$ form the set of all subelement nodes of element e, and $p_1, ..., p_n$ are a set of all text nodes under element e.
- Each attribute node a will be modelled as an object o with $\sigma(o) =$ "@" $+ name(a)$ and $\pi(o) = [A : content(a)]$, where $A \in \mathbb{A}$.
- Each text node p will be modelled as an atomic value d with $d = content(p)$.

Example 2.4 In Figure 2, we show the XML data graph for the XML document book.xml from Figure 1. Note that object identifiers do not appear in an XML data graph, since they are invisible to the users. We indicate them here with $i_k (k \in [1, 42])$ to ease the discussion. In particular,

- the element node *book* corresponds to the object $book(i_1, [title : i_2, author : i_3, author : i_4, author : i_5, section : i_6, section : i_{24}])$, and
- the attribute node *height* is represented as the object $@height(i_{18}, [STRING :$ "400"$])$, where the text node "400" is considered as an atomic value appeared in the object's value.

Indeed, path expressions can also be easily described within this model. Given two objects with identifiers i and $i^{'}$, then the object with i is the parent of the object with $i^{'}$, denoted by $i^{'} \sqsubseteq i$ iff $i^{'}$ occurs in the value of the object with i. Moreover, the object with i is the ancestor of the object with $i^{'}$, denoted by $i^{'} \sqsubset i$, iff there exists a finite set (can be empty) of objects with identifiers $i_1, ..., i_n$ such that $i^{'} \sqsubseteq i_1 \sqsubseteq ... \sqsubseteq i_n \sqsubseteq i$ holds.

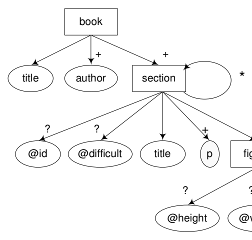


Figure 3: An XML schema graph

Example 2.5 Look at Example 2.4, clearly, $i_6 \sqsubseteq i_1$ and $i_{17} \sqsubset i_1$ hold.

Similarly, we describe XML database schemata as a collection of XML schema graphs. Given a set of class schemata, an XML schema graph can be constructed by using an algorithm called *Schema Reconstruction* as follows.

Algorithm 2.1 (Schema Reconstruction)

```
Input: A finite set of class schemata $S$
        for complex classes
Output: An XML schema graph $G$
Begin
FOR EACH $s_c \in S$
    DO AddClassSchema($s_c$)
ENDFOR
End
Procedure AddClassSchema (in: $s_c \in S$)
Begin
IF a vertex labelled $c$ does not exist in $G$
THEN add a vertex labelled $c$
ENDIF
FOR EACH $A^f \in Attr(s_c)$
    IF $A$ is a class name
    THEN add a vertex labelled $A$
            and an edge $e$ from $c$ to $A$
        IF $f \in \{+, *, ?\}$
        THEN label the edge $e$ with $f$
        ENDIF
    ENDIF
ENDFOR
End
```

Example 2.6 Figure 3 shows the XML schema graph for the XML document book.xml in Figure 1. The following three complex class schemata can be obtained using a named perspective.

$$s_{book} = [title, author^+, section^+]$$
$$s_{section} = [@id^?, @difficult^?, title, p^+, figure^?, section^*]$$
$$s_{figure} = [@height^?, @width^?, title]$$

Comparing the XML data graph with the XML schema graph shown in Figures 2 and 3, respectively, it is quite straightforward to see that the XML schema graph is a homomorphic image of the XML data graph.

Definition 2.7 An XML data graph is said to be *valid* with respect to an XML schema graph iff the following two conditions are satisfied:

1. There exists a surjective graph homomorphism from the XML data graph to the XML schema graph that preserves labels of vertices.

2. The XML data graph satisfies all the constraints imposed by labels of edges in the XML schema graph.

At the database level, a database instance $\mathbb{D}$ is *valid* for a database schema $\mathbb{S}$ if and only if each XML data graph in $\mathbb{D}$ is valid for at least one XML schema graph in $\mathbb{S}$.

2.4 Class Hierarchy

To show that classes of objects in an XML data graph constitute a hierarchical structure, we first need to define the subclass relations between classes. Let K be a set of class names over an XML data graph G, and $c_1, c_2 \in K$, then c_1 is a subclass of c_2, denoted by $c_1 \leq c_2$, by using the following rules inductively:

1. $c_1 \leq c_1$

2. $\perp \leq c_1$, where $\perp$ denotes the base class

3. $\top \leq c_1$, where $\top$ denotes the super class

4. $c_1 \leq c_2$ iff there exists an object o satisfying $c_2 = \sigma(o)$, and c_1 or $\{c_1\} \in \tau(o)$

Correspondingly, when $c_1 \leq c_2$ holds, we can say that c_2 is a *superclass* of c_1. Moreover, with the above rules the following result with respect to the subclass relations between classes in the XOM can be easily obtained.

Theorem 2.1 *Given an XML data graph G, and let K be a set of class names over G, then the partially ordered set $(K \cup \{\perp, \top\}, \leq)$ is a complete lattice.*

Indeed, this subclass relation over an XML data graph can be expressed with a diagram, called a class hierarchy.

Example 2.7 The class hierarchy over the XML data graph shown in Figure 2 is presented in Figure 4.

The class $\top$ is a superclass of all other classes as a least upper bound, while $\perp$ is the subclass of all other classes as a greatest lower bound. Because of $\sigma(o_1) = book$ and $\tau(o_1) = [title, author, section]$ it is obvious that classes *title*, *author* and *section* are the subclasses of class *book* in accordance with rule 4 above. Similarly, we can obtain that the subclasses of class *section* are classes *figure*, *title*, *p*, *@difficult* and *@id* by inductively using the above rules. Finally, the class hierarchy is constructed as shown in Figure 4.

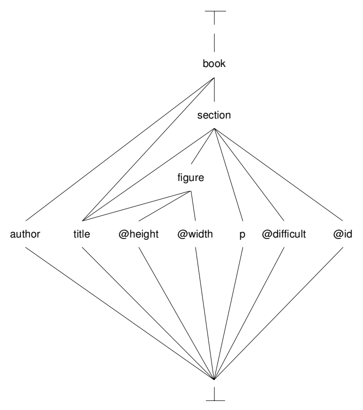


Figure 4: A class hierarchy

3 An XML Calculus

In this section, we introduce a novel logic-based query language for XML, called XML calculus (XMLC), which is a counterpart of the Relational Calculus. Instead of first-order logic used in Relational Calculus, XMLC employs higher-order logic as a natural approach to specify declarative queries over XML data. After defining a fundamental type system as a basis for formalizing the concept of higher order, the formal syntax and semantics of XMLC are presented.

3.1 Types

To facilitate the formalization, besides $\mathbb{C}$ and $\mathbb{A}$, we adopt the notation $\mathbb{X}$ to denote a countably infinite set of variables. Furthermore, $\mathbb{C}$, $\mathbb{A}$ and $\mathbb{X}$ are assumed to be pairwise disjoint. The type system of XMLC is defined by

$$\tau \;=\; \oslash \mid b \mid \tau_c \mid \tau_x \mid l : \tau \mid \{\tau\} \mid [\tau_1, \ldots, \tau_n] \mid \tau_1 \vee \tau_2.$$

In this system, $\oslash$ denotes a trivial type with only one value (see below), b represents any atomic type $b \in \mathbb{A}$, and τ_c represents a class type for $c \in \mathbb{C}$. τ_x refers to a type variable for $x \in \mathbb{X}$, while $l : \tau$ represents a type τ labelled by l. Furthermore, three type constructors are defined comprising a tuple constructor [ ], a set constructor {}, and a union constructor $\vee$.

The purpose of introducing a trivial type $\oslash$ into the system is to enhance the flexibility of the language when handling optionality of XML data at the type level. In addition, union type constructors are needed to support this feature. The details regarding how XMLC queries can deal with the optional elements of XML documents will be explained later in Section 4. Moreover, since the self-descriptive feature of XML data requires that an XML query language provides the possibility for querying over schema information as well as instance data, we develop type variables τ_x as fixed but unknown types to describe incomplete type information accompanied with querying over database schema. To distinguish other types from type variables, we adopt the convention that all types except for type variables defined in the type system are called type values in this paper.

We associate a set of values $tdom(\tau)$ with each type τ. These sets are defined as follows:

- $tdom(\oslash) = \{\lambda\}$, where λ means that no specified value exists.
- $tdom(b) = dom_V(b)$, where $dom_V(b)$ is the value domain associating with atomic type $b \in \mathbb{A}$.
- $tdom(\tau_c) = \{\mathcal{I} \mid c$ refers to a complex class$\} \cup \{\mathcal{D} \mid c$ refers to an atomic class$\}$.
- $tdom(\tau_x) = \{\tau \mid \tau$ refers to a type value$\}$.
- $tdom(l : \tau) = \{l : v \mid l \in \mathbb{C} \cup \mathbb{A} \cup \mathbb{X}, v \in tdom(\tau)\}$.
- $tdom(\{\tau\}) = \{\{v_1, ..., v_m\} \mid m \geq 1, v_i \in tdom(\tau), 1 \leq i \leq m\}$.
- $tdom([\tau_1, ..., \tau_n]) = \{[v_1, ..., v_n] \mid v_i \in tdom(\tau_i), 1 \leq i \leq n\}$.
- $tdom(\tau_1 \vee \tau_2) = tdom(\tau_1) \cup tdom(\tau_2)$.

As a type can be constructed recursively from other types by means of type constructors, recursive and higher-order types may occur within this type system. To describe higher-order features we need the concept of order. For convenience, terms and formulae are considered as being associated with an order in the sense that the order of a term is the order of its type, while the order of a formula is the maximum order among the orders of terms within that formula.

Definition 3.1 The *order of a type* τ is 0 for $\tau = \oslash$, $\tau = b$ and $\tau = \tau_c$, the same as the order of τ' for $\tau = l : \tau'$, $k+1$ for $\tau = \{\tau'\}$ with τ' of order k, $k+1$ for $\tau = [\tau_1, ..., \tau_n]$, where k is the maximal order of $\tau_1, ..., \tau_n$, $\max\{k_1, k_2\}$ for $\tau = \tau_1 \vee \tau_2$, where k_i is the order of type τ_i, and determined by the corresponding type value for $\tau = \tau_x$.

By these definitions the key to produce higher-order types apparently lies in tuple and set constructors. Therefore, XMLC develops two kinds of higher-order constructs: class constructs and group constructs in the language to handle tuple and set types, respectively. More specific, class constructs can be used for capturing objects that are the aggregation of other objects, while group constructs provide necessary means for dealing with transformation between non-set or set objects.

Using the type system above, the links to the underlying class schemata within an XML database can be easily established. Indeed, any attribute within a class schema can always be described by a type, and therefore, it is straightforward to represent a class schema with a tuple of types. Informally, let us look at a simple example to illustrate this.

Example 3.1 Recall the class schemata of the XML document book.xml shown in Example 2.6. The corresponding type expressions are as follows:

- For S_{book} we obtain $[title : \tau_{title}, author : \{\tau_{author}\}, section : \{\tau_{section}\}]$.
- For $S_{section}$ we obtain $[@id : \tau_{@id} \vee \oslash, @difficult : \tau_{@difficult} \vee \oslash, title : \tau_{title}, p : \{\tau_p\}, figure : \tau_{figure} \vee \oslash, section : \{\tau_{section}\} \vee \oslash]$.
- For S_{figure} we obtain $[@height : \tau_{@height} \vee \oslash, @width : \tau_{@width} \vee \oslash, title : \tau_{title}]$.

3.2 Syntax

In this section, we present the syntax of XMLC in a style similar to the Relational Calculus (Abiteboul et al. 1995). However, these two languages differ from each other in many aspects. Due to the embedded higher-order logic XMLC is much more expressive than the Relational Calculus. Furthermore, with the definition of XMLC, the Relational Calculus can be treated as a special case under certain restrictions on types and constructs.

Definition 3.2 In addition to variables and constants, XML calculus provides four more kinds of *complex terms* as follows:

Labeled terms: A *labeled term* can be expressed as $t_1 : t_2$, where t_1 is a variable or constant and t_2 is a term, which may be of higher order. If t_1 denotes some label l and t_2 is of type τ_2, then the labeled term $t_1 : t_2$ has the type $l : \tau_2$.

Class terms: A *class term* has the form $l : t(t_1, \dots, t_n)$, where $l : t$ is a labeled term indicating class name l and class variable t, and $t_1, \dots, t_n$ are labeled terms functioning as arguments of such a class term. For class term $l : t(t_1, ..., t_n)$, if $t, t_1, \dots, t_n$ have types of $\tau, \tau_1, \dots, \tau_n$, respectively, then $\tau = [\tau_1, ..., \tau_n]$ must hold, and $l : t(t_1, \dots, t_n)$ is of type $l : \tau$ or $l : [\tau_1, \dots, \tau_n]$.

The types of class terms are closed under composition. For instance, $l : t(t_1, ..., t_n)$ can be an argument for class term $l' : t'(t_0, l : t(t_1, ..., t_n))$.

Group terms: A *group term*, denoted as $t_1(t_2)$, is a function that groups objects of the same type into an object of a set type. If t_2 has a type of τ_2, then t_1 is of type $\{\tau_2\}$.

Path terms: The expression $t_1.t_2$ is used to denote *path terms*. For each path term $t_1.t_2$, if t_1, t_2 have types τ_1 and τ_2, respectively, then $t_1.t_2$ is of type τ_2.

A term is *ground* if and only if no variable occurs in it. For example, *1* and *title* : "*Databases*" are ground terms. In XMLC, terms in higher-order logic can be inductively defined over the type system. For convenience, we use $\#t$ in the following to refer to the type of term t.

Within a particular XML database types of class terms are determined by the underlying class schemata. Therefore, class terms with the same class name have the same type. For example, the types of both $book : B_1(title : t_1)$ and $book : B_2(title : t_2, author : a_2)$ are determined by class schema S_{book}, which has the corresponding type expression $[title : \tau_{title}, author : \{\tau_{author}\}, section : \{\tau_{section}\}]$ as shown in Example 3.1. Therefore, we have $\tau_{B_1} = \tau_{B_2} = [title : \tau_{title}, author : \{\tau_{author}\}, section : \{\tau_{section}\}]$, $\tau_{t_1} = \tau_{t_2} = \tau_{title}$ and $\tau_{a_2} = \{\tau_{author}\}$. To simplify the expressions, $book : B_1(title : t_1)$ and $book : B_2(title : t_2, author : a_2)$ are treated as abbreviations for $book : B_1(title : t_1, author : -, section : -)$ and $book : B_2(title : t_2, author : a_2, section : -)$, respectively, where "$-$" means that values occurring in that place are always ok since we are not concerned about them.

If class terms and group terms are regarded as special terms based on truth values, then they are atoms occurring in the language. Atoms are generally categorized into predicate formulae and comparison formulae.

Definition 3.3 *Predicate formulae* include class predicates $t(t_1, ..., t_n)$, where $t, t_1, ..., t_n$ are labeled terms, and group predicates $t(t_1)$, where t, t_1 are variable terms.

Path formulae are expressed as $t_1 \ . \ t_2$, where t_1 and t_2 are terms.

Comparison formulae are expressed by t_1 *cop* t_2, where t_1 and t_2 are terms of some common type, and *cop* represents comparison operators such as $=$, $\neq$, $<$, $>$, $\leq$ and $\geq$.

Starting from atoms, well-formed formulae (wff for short) of XMLC can be defined inductively as usual with a set of logic symbols $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \exists, \forall$ as follows.

Definition 3.4 The set of *well-formed formulae (wff)* of XMLC is the smallest set satisfying:

- Each atomic formula is a wff.
- φ *bop* ψ is a wff, if *bop* is a logical symbol amongst $\vee, \wedge, \Rightarrow$ and $\Leftrightarrow$, and φ and ψ are well-formed formulae.
- $\neg\psi$ is a wff, if ψ is a well-formed formula.

- $\exists x.\ (\psi)$ and $\forall x.\ (\psi)$ are wff, if ψ is a well-formed formula and x is a variable.

Although one of important features of XML calculus is to be able to quantify higher-order variables as well as first-order variables, the approach to identify free and bound variables appearing in a formula is the same with the Relational Calculus. Analogously, a variable appearing with a quantifier is a bound variable within the scope of that quantifier, otherwise, a variable is free with respect to that scope. It may happen that a variable is free and bound in a formula at the same time as shown in the following example.

Example 3.2 Given a formula $\psi = \forall n, d.(author : A(name : n, address : d) \wedge \exists A.(book : B(title : t, author : A) \wedge author : A(name : n)))$, then free variables occurring in this formula are $\{A, B, t\}$, while bound variables occurring in this formula are $\{n, d, A\}$. Obviously, the variable A is both a free variable and a bound variable in the formula ψ. Through variable renaming, the formula ψ can also be expressed by the formula $\varphi = \forall n, d.(author : A_1(name : n, address : d) \wedge \exists A_2.(book : B(title : t, author : A_2) \wedge author : A_2(name : n)))$ without changing the semantics.

3.3 Semantics

Let us now discuss the semantics of terms and formulae defined in XMLC. The basic idea is to treat types as domains containing all possible values of these types, respectively. Then terms can be interpreted as values from the domains of their types, and formulae can be interpreted as Boolean values on the basis of the interpretation of terms.

Definition 3.5 Given a database schema $\mathbb{S}$ and a fixed set $\mathfrak{D} = \{tdom_i\}_{i \in [1,n]}$ of type domains over $\mathbb{S}$, then an *interpretation* of XMLC over $\mathbb{S}$ consists of a database db over $\mathbb{S}$ and a valuation function ν: $\mathbb{X} \to \mathfrak{D}$ with $\nu(x) \in tdom(\tau)$, where $x \in \mathbb{X}$ and $\#x = \tau$.

To distinguish the interpretation for terms and formulae, we use notations ω and $\hat{\omega}$ to denote the interpretation for terms and formulae, respectively.

Definition 3.6 The *continuation of an interpretation* to terms and formulae is defined as follows: For terms,

- $\omega(t) = \nu(t)$ if t is a variable
- $\omega(t) = t$ if t is a constant
- $\omega(t_1 : t_2) = \omega(t_1) : \omega(t_2)$ if $t_1 : t_2$ is a labeled term.
- $\omega(t(t_1, ..., t_n)) = \omega(t)(\omega(t_1), ..., \omega(t_n))$ if $t(t_1, ..., t_n)$ is a class term.
- $\omega(t_1(t_2)) = \omega(t_1)(\omega(t_2))$ if $t_1(t_2)$ is a group term.

For a formula φ, its interpretation $\hat{\omega}(\varphi)$ by Boolean values true (T) or false (F) is obtained as follows:

- For a class predicate formula $\hat{\omega}(t(t_1, ..., t_n)) = T$ iff $\omega(t)[\omega(t_1), ..., \omega(t_n)] \in db$ holds. In all other cases it evaluates to F.
- For a group predicate formula $\hat{\omega}(t_1(t_2)) = T$ iff $\omega(t_2) \in \omega(t_1)$ holds. In all other cases it evaluates to F.
- For a path formula $\hat{\omega}(t_1.t_2) = T$ iff $\omega(t_2) \sqsubset \omega(t_1)$ holds. In all other cases it evaluates to F.
- For an existentially quantified formula $\hat{\omega}(\exists x.(\psi)) = T$ iff the replacement of x in ψ by some $\omega(x)$ results in a formula φ with $\hat{\omega}(\varphi) = T$.
- For a universally quantified formula $\hat{\omega}(\forall x.(\psi)) = T$ iff the replacement of x in ψ by every $\omega(x)$ results in a formula φ with $\hat{\omega}(\varphi) = T$.
- Other formulae are interpreted in the same way as in the first-order logic.

According to the above interpretation of universally quantified formulae the set of values $\omega(x)$ is infinite but only finitely many objects exist in an XML database, so that $\hat{\omega}(\forall x.(\psi))$ will always result in F. To avoid this problem we employ the approach that restricts the semantics of the language using active type domains . Roughly speaking, the idea is to associate variable assignments with active type domains specific to some database instance, while the underlying type domains are still allowed to be infinite. Therefore, given a query Q on an XML database db, let $adom(Q)$ be the finite set of constants and objects occurring in the query Q, and $adom(db)$ be the finite sets of constants and objects occurring in the database db, then the active type domains, denoted by $Adom$, consist of $adom(Q)$ and $adom(db)$ such that $Adom = adom(Q) \cup adom(db)$. To be precise, valuations under active type domain semantics need to be revisited as "in a database db, a valuation ν on variable x of type τ in a query Q is of the form $\nu(x) \in tdom(\tau) \cap (adom(Q) \cup adom(db))$. This change makes sure that valuations can be limited within active type domains.

Since XMLC is a purely declarative language, queries can be written down in a very elegant and natural way. A query expression has the form $Q = \{x |\ \varphi\}$, where x represents a variable that might be of higher-order, and φ is a well-formed formula with only one free variable, i.e. x. Furthermore, linking back to the underlying XOM the answer schema of Q, denoted by $ans(Q)$, is an XML schema graph rooted at the class schema corresponding to x. Therefore, each query in the language is always associated with an input-schema $in(Q) = \mathbb{S}$ and an output-schema $out(Q) = ans(Q)$. Given an XML database db over $\mathbb{S}$, the query is a mapping q over db such that

$$q(Q)(db) = \{G_{\omega(x)} \mid \hat{\omega}(\varphi) = T\}$$

where $G_{\omega(x)}$ represents an XML data graph rooted at object $\omega(x)$.

In the language new objects can be created during querying in accordance with the logic. For this we have to make the assumption of disjoint sets of intensional and extensional class predicates, which is important for the expressive power of the language up to object creation. Given a query Q with associated input schema $in(Q)$ and output schema $out(Q)$, then class names for intensional and extensional class predicates over Q, denoted by $\mathcal{T}_{int}$ and $\mathcal{T}_{ext}$, respectively, can be obtained by the following rules:

- $\mathcal{T}_{ext} = \{l \mid l : t(t_1, ..., t_n)$ is a class predicate in query $Q, l \in in(Q)\}$.
- $\mathcal{T}_{int} = \{l \mid l : t(t_1, ..., t_n)$ is a class predicate in query $Q, l \in out(Q) - in(Q)\}$.

Obviously, $\mathcal{T}_{int} \cap \mathcal{T}_{ext} = \emptyset$ holds. Furthermore,

1. objects can only be created within intensional class predicates, and
2. class types of created objects are determined by the associating intensional class predicates.

To better see the mechanism of object creation in XMLC, the following example is provided, in which some *book'* objects are created in the database.

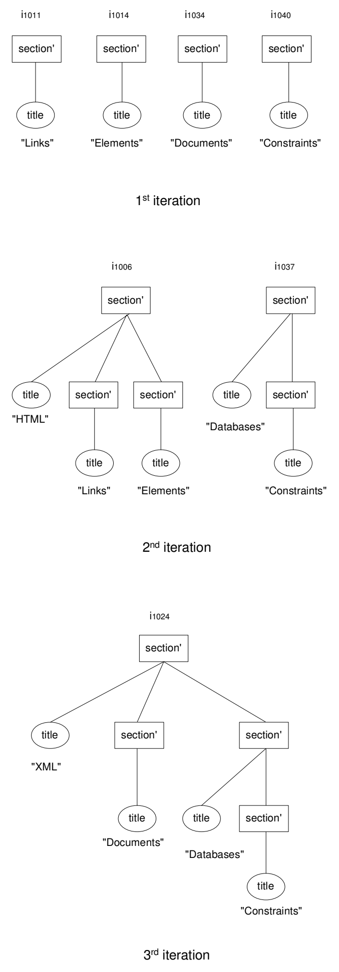


Figure 5: Representation as a graph

Example 3.3 Let us consider to list books in the XML document bib.xml in Figure 6, which were published by Addison-Wesley after 1991, including their year and title:

$$\{B' \mid \forall B.(\exists x, y.book : B(publisher : \text{“Addison-Wesley”}, \\ @year : y, title : x) \wedge y > 1991 \\ \Rightarrow book' : B'(@year : y, title : x))\}$$

Clearly, in this query, $\mathcal{T}_{ext} = \{book\}$ and $\mathcal{T}_{int} = \{book'\}$. All objects in the class *book'* are new objects created by the class predicate $book' : B'(@year : y, title : x)$ having the class type $\tau_{book'} = [@year : \tau_{@year}, title : \tau_{title}]$.

4 Expressiveness of XML Calculus

The purpose of this section is to illustrate some issues that are relevant for handling XML data. The provided examples show how XMLC deals with recursive structures, nest and unnest operations, and query composition, and how distinguished XML data characteristics such as optionality, object order and schema queries can be captured in the language.

4.1 Recursive Structures

XMLC is more expressive than the Relational Calculus due to its embedded higher-order logic. For example, Relational Calculus cannot express fixed-points, but XMLC can. As shown in the following example, XMLC is able to inductively create objects, so recursive structures can be reconstructed within an XML document.

Example 4.1 Consider the XML document book.xml (see Figure 1) and its XML data graph (see Figure 2). Look at a query to find all the sections in book.xml. For such sections, return $section'$ objects with the section titles, while preserving the original hierarchy. We can write down the following query Q for this:

$$\{S' \mid \forall x, y.((\exists S_1.\ section : S_1(title : x, section : y) \wedge \\ y = \lambda) \vee (\exists S_3, S_2.\ section : S_2(title : x, \\ section : S_3) \wedge \forall S_4.(y(S_4) \Leftrightarrow \forall w.(section : \\ S_3(title : w) \Rightarrow \exists z.section' : S_4(title : w, \\ section' : z)))) \Rightarrow section' : S'(title : x, section' : y))\}$$

For this query the following answer is obtained

$$\{i_{1011}, i_{1014}, i_{1034}, i_{1040}, i_{1006}, i_{1037}, i_{1024}\},$$

and the evaluation procedure is illustrated step by step as follows:

$section'$	title	$section'$
i_{1011}	i_{12}	λ
i_{1014}	i_{15}	λ
i_{1034}	i_{35}	λ
i_{1040}	i_{41}	λ

$section'$	title	$section'$
i_{1011}	i_{12}	λ
i_{1014}	i_{15}	λ
i_{1034}	i_{35}	λ
i_{1040}	i_{41}	λ
i_{1006}	i_{9}	i_{1011}, i_{1014}
i_{1037}	i_{38}	i_{1040}

$section'$	title	$section'$
i_{1011}	i_{12}	λ
i_{1014}	i_{15}	λ
i_{1034}	i_{35}	λ
i_{1040}	i_{41}	λ
i_{1006}	i_{9}	i_{1011}, i_{1014}
i_{1037}	i_{38}	i_{1040}
i_{1024}	i_{27}	i_{1034}, i_{1037}

In each iteration, some new *section'* objects are created on the basis of the existing objects in the input database or new objects which have already been created in the previous iterations. The query terminates, when there are no more new objects produced in the result. Figure 5 provides the corresponding graph representation for new objects created in each iteration.

In addition, this example shows the means adopted in XMLC to handle optionality in XML data. Variable y is specified as null value through the formula $y = \lambda$ to obtain *section* objects that contain no other *section* objects. This approach is very useful when we are particularly interested in the lack of some kinds of objects within a class term.

4.2 Nest and Unnest

Nesting and unnesting constitute well-known problems encountered in the development of query languages dealing with complex values. XMLC handles this problem by introducing group constructs in the language. With group constructs, a two-way type conversion between τ and $\{\tau\}$ can be easily established. This is shown in the example below.

Example 4.2 Let us look at the XML document bib.xml, then create a list of all the title-author pairs, with each pair enclosed in a *pair* element:

$$Q = \{R \mid \exists y.(\forall P.(\forall B.(\exists z, x.\ book : B(title : z, author : x) \wedge x \neq \lambda \Rightarrow \forall a.(x(a) \Rightarrow pair : P(title : z, author : a))) \Leftrightarrow y(P)) \wedge root' : R(pair : y))\}$$

By using $x(a)$ in the formula $x(a) \Rightarrow pair : P(title : z, author : a)$, we can unnest a set of *author* objects in x for some book into many title-author pairs (z, a) enclosed in *pair* objects correspondingly. As for nesting, $y(P)$ is used to nest a set of *pair* objects into y of type $\{\tau_{pair}\} = \{[title : \tau_{title}, author : \tau_{author}]\}$, and then y is passed into an intensional class predicate $root' : R(pair : y)$ to produce a new $root'$ object. Therefore, this example also illustrates how to get a single tree in the final result by integrating separate objects into the $root'$ object's value.

4.3 Composition

From the definition of XMLC we know that a query generates a set of trees as its result. Since irregular expressions are identified as one of the main advantages of XML data, it is a desirable goal to provide flexibility at the language level to enable heterogenous trees to be created in answers to queries. To achieve this goal the composition of queries becomes important. To see how XMLC tackles this problem, an example is provided here.

Example 4.3 Look at the XML document bib.xml again. Now, for each book with an author, return a *book′* with the book's title and its authors. For each book with an editor, return a *book′* with the book's title and the editor's affiliation.

In this example, the query can be considered as the composition of two subqueries. One is to find all books having authors, and then return such a book with its title and authors. Another one is to find all books having editors, and then return such a book with its title and the editor's affiliation. These two subqueries can be written down individually with XMLC as follows:

$$\{B' \mid \forall B.(\exists t_1, a_1.\ book : B(title : t_1, author : a_1) \wedge a_1 \neq \lambda \Rightarrow book' : B'(title : t_1, author : a_1))\}$$

$$\{B' \mid \forall B.(\exists E, t_2, a_2.\ book : B(editor : E, title : t_2) \wedge editor : E(affiliation : a_2) \Rightarrow book' : B'(title : t_2, affiliation : a_2))\}$$

In fact, these two subqueries can easily be composed into one query by using a logic symbol ”$\vee$” linking the two subqueries together. We show the

```
<bib>
   <book year=1994>
      <title>"TCP/IP Illustrated"</title>
      <author>"W. Stevens"</author>
      <publisher>"Addison-Wesley"</publisher>
      <price> 65.95</price>
   </book>
   <book year=1992>
      <title>"Advanced Programming in the Unix environment"</title>
      <author>"W. Stevens"</author>
      <publisher>"Addison-Wesley"</publisher>
   </book>
   <book year=2000>
      <title>"Data on the Web"</title>
      <author>"Serge Abiteboul"</author>
      <author>"Peter Buneman"</author>
      <author>"Dan Suciu"</author>
      <publisher>"Morgan Kaufmann Publishers"</publisher>
      <price>39.95</price>
   </book>
   <book year=1999>
      <title>"The Economics of Technology and Content for Digital TV"</title>
      <editor>
           <last>"Gerbarg"</last><first>"Darcy"</first>
            <affiliation>"CITI"</affiliation>
      </editor>
         <publisher>"Kluwer Academic Publishers"</publisher>
   </book>
</bib>
```

Figure 6: bib.xml

composite query:

$$\{B' \mid \forall B_1.(\exists t_1, a_1.\ book : B_1(title : t_1, author : a_1) \wedge a_1 \neq \lambda \Rightarrow book' : B'(title : t_1, author : a_1)) \vee \forall B_2.(\exists E, t_2, a_2.\ book : B_2(editor : E, title : t_2) \wedge editor : E(affiliation : a_2) \Rightarrow book' : B'(title : t_2, affiliation : a_2))\}$$

After executing this query, a class named *book′* containing some new objects is produced in the database. The corresponding class schema for *book′* is $S_{book'} = [title, author^*, affiliation^?]$, or alternatively expressed by class type expression $[title : \tau_{title}, author : \{\tau_{author}\} \vee \oslash, affiliation : \tau_{affiliation} \vee \oslash]$.

4.4 Class Names and Atomic Values

Although the notions of class name and atomic value seem quite irrelevant at the first glance, they correspond to two significant portions of an XML document: markup and character data, respectively, as specified in (World Wide Web Consortium 2004). For this reason, they are discussed together here. Moreover, we assume that the wild card % refers to matching zero or more occurrences of characters.

Example 4.4 Let us consider the XML document bib.xml in Figure 6. In order to find books in which the name of some subelement ends with the string “ce”, and for each such book, return the title and the qualifying subelement, we write the following XMLC query:

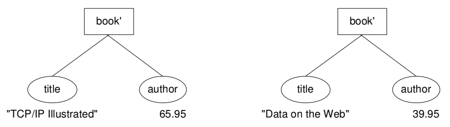
$$\{B' \mid \forall B.(\exists t, z, s.book : B(title : t,\ s : z) \wedge s = \text{“\%ce”} \Rightarrow book' : B'(title : t,\ s : z))\}$$

In class predicate $book : B(title : t,\ s : z)$, $\tau_B = [title : \tau_{title}, s : \tau_x]$. As discussed in the preceding type system, τ_x is a type variable indicating that the type of z is unknown in this case. The reason resulting in unknown types lies in the fact that the label of z is a variable s, and thus the type of z can not be determined from the underlying class schema S_{book}. After executing the query on the running database, the

```
<prices>
   <book>
      <title>"Advanced Programming in the Unix environment"</title>
      <source>"bstore2.example.com"</source>
      <price>65.95</price>
   </book>
   <book>
      <title>"Advanced Programming in the Unix environment"</title>
      <source>"bstore1.example.com"</source>
      <price>65.95</price>
   </book>
   <book>
      <title>"TCP/IP Illustrated"</title>
      <source>"bstore2.example.com"</source>
      <price>65.95</price>
   </book>
   <book>
      <title>"TCP/IP Illustrated"</title>
      <source>"bstore1.example.com"</source>
      <price>65.95</price>
   </book>
   <book>
      <title>"Data on the Web"</title>
      <source>"bstore2.example.com"</source>
      <price>34.95</price>
   </book>
   <book>
      <title>"Data on the Web"</title>
      <source>"bstore1.example.com"</source>
      <price>39.95</price>
   </book>
</prices>
```

Figure 7: prices.xml

following results are obtained along with the interpretation of variable s. That is *price* and $\tau_x = \tau_{price} \vee \emptyset$.



To show how atomic values can be queried in XML databases, we slightly change the above question to “find books with the title containing the string “Web”, and return each such book with its title”. The following query can be used for this purpose: $\{B' \mid \forall B.(\exists x.book : B(title : x) \wedge x = “\%web\%” \Rightarrow book' : B'(title : x))\}$.

4.5 Object Order

When object order is a concern for users, it is easy to tackle this problem by means of functions. For example, we can define a function $pos(oid)$ such that function $pos(oid)$ takes an object identifier oid as an argument and returns the relative order of that object with identifer oid amongst objects of the same class appeared in the current class tuple. We show how object order can be taken into consideration with such a function by means of a simple example.

Example 4.5 Consider the XML document bib.xml again. For each book that has at least one author, list the title and first two authors, and an empty “et-al” element if the book has additional authors:

$$\{A \mid \exists z.(\forall B'.(\forall B_1.(\exists t_1, x_1.book : B_1(title : t_1,
author : x_1) \wedge \exists a.(x_1(a) \wedge pos(a) > 2)
\Rightarrow \exists y.(\forall b.(x_1(b) \wedge pos(b) \leq 2 \Leftrightarrow y(b)) \wedge
book^{'} : B'(title : t_1, author : y, et\text{-}al : \emptyset)))
\wedge \forall B_2.(\exists t_2, x_2.book : B_2(title : t_2, author : x_2) \wedge
\forall a.(x(a) \Rightarrow pos(a) \leq 2) \Rightarrow book^{'} : B'(title : t_2,
author : x_2)) \Leftrightarrow z(B')) \wedge ans : A(book' : z))\}$$

4.6 Aggregation Functions

XMLC has very strong expressive power, so even some aggregation functions can also be simulated, such as $min()$ and $max()$, which are widely used in many query languages. Here, we provide one example to show how the semantics of $min()$ in a particular context can be captured by a proper formula in XMLC:

Example 4.6 Consider the XML document prices.xml in Figure 7, find the minimum price for each book in this document, return in the form of a “minprice” element with the book’s title as its attribute and the minimum price as its subelement:

$$\{A \mid \exists y.(\forall M.(\forall B_1.(\exists t, x_1.book : B_1(title : t, price : x_1)
\wedge \forall x_2.(\exists B_2.book : B_2(title : t, price : x_2) \Rightarrow
x_2 \geq x_1) \Rightarrow minprice : M(@title : t, price : x_1))
\Leftrightarrow y(M)) \wedge ans : A(minprice : y))\}$$

5 Conclusion

This paper arose from research on logical foundations of XML and XML query languages. We asked the simple question what would be an analog of the Relational Calculus for querying XML databases. In order to answer this question we defined the XML object model (XOM), which exploits the similarity between the tree structures in XML and those arising in similar complex value or object-oriented data models. We showed that the heterogeneous structures in XML can be encapsulated into several object values that are combined as class tuples under flexible class schemata. The resulting classes in an XML data graph give rise to a structure of a complete lattice, thus an elegant class hierarchy can be constituted. Following the object paradigm we observed that object identifers cannot only serve for uniquely identifying objects, but also be used as order primitives.

On a basis of XOM a novel logic-based query language, the XML calculus (XMLC) has been formally defined. We showed that XMLC is a very natural language tailored for XML, as a higher-order type system can be easily built upon the XOM. The identifer and value semantics associated with objects provide the capability to use higher-order syntax while preserving first-order interpretations without any loss of expressive power of the language. Moreover, XMLC can elegantly reflect fixed-point semantics to incorporate recursion, hence makes it possible to express recursive structures and transitive closure in the language. We illustrated these features with meaningful examples that demonstrate the key characteristics of the language.

The work in (Kirchberg, Riaz-ud-Din, Schewe & Tretiakov 2006) follows an approach to map the popular XML query language XQuery to a rational tree algebra for the purpose of algebraic query optimisation, thus in a sense complements the work in this paper by an algebraic analog relating to Relational Algebra. As we are interested in the logical foundations of XML and not so much in practical issues of query optimisation, we will not attempt a decent comparison with this analog line of research.

In order to continue this research we are interested in refining the XMLC with respect to decidability and safety. In parallel to our investigation of a calculus for XML we already investigated logic-programming-based approaches to querying XML following the idea of the Identity Query Language (IQL) (Abiteboul & Kanellakis 1989, Abiteboul & Kanellakis 1998). The results of this research will soon be submitted for publication.

References

Abiteboul, S., Buneman, P. & Suciu, D. (2000), *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann Publishers, San Francisco, California.

Abiteboul, S. & Grumbach, S. (1990), COL: a logic-based language for complex objects, *in* 'Advances in database programming languages', ACM Press, New York, NY, USA, pp. 347–374.

Abiteboul, S., Hull, R. & Vianu, V. (1995), *Foundations of Databases: The Logical Level*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Abiteboul, S. & Kanellakis, P. C. (1989), Object identity as a query language primitive, *in* 'SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data', ACM Press, New York, NY, USA, pp. 159–173.

Abiteboul, S. & Kanellakis, P. C. (1998), 'Object identity as a query language primitive', *J. ACM* **45**(5), 798–842.

Abiteboul, S., Quass, D., McHugh, J., Widom, J. & Wiener, J. L. (1997), 'The lorel query language for semistructured data.', *Int. J. on Digital Libraries* **1**(1), 68–88.

Bancilhon, F. & Khoshafian, S. (1986), A calculus for complex objects, *in* 'PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems', ACM Press, New York, NY, USA, pp. 53–60.

Beeri, C., Naqvi, S., Ramakrishnan, R., Shmueli, O. & Tsur, S. (1987), Sets and negation in a logic data base language (LDL1), *in* 'PODS '87: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems', ACM Press, New York, NY, USA, pp. 21–37.

Chen, W., Kifer, M. & Warren, D. S. (1989), HiLog: A first-order semantics for higher-order logic programming constructs., *in* 'NACLP', pp. 1090–1114.

Chen, W., Kifer, M. & Warren, D. S. (1993), 'HILOG: A foundation for higher-order logic programming.', *J. Log. Program.* **15**(3), 187–230.

Church, A. (1940), 'A formalulation of the simple theory of types.', *Journal of Symbolic Logic* **5**(56-68).

Codd, E. F. (1970), 'A relational model of data for large shared data banks', *Commun. ACM* **13**(6), 377–387.

Codd, E. F. (1972), 'Relational completeness of data base sublanguages.', *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California* .

Deutsch, A., Fernandez, M., Florescu, D., Levy, A. & Suciu, D. (1999), A query language for XML, *in* 'WWW,Xpath '99: Proceeding of the eighth international conference on World Wide Web', Elsevier North-Holland, Inc., New York, NY, USA, pp. 1155–1169.

Kifer, M. & Lausen, G. (1989), F-logic: a higher-order language for reasoning about objects, inheritance, and scheme, *in* 'SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data', ACM Press, New York, NY, USA, pp. 134–146.

Kirchberg, M., Riaz-ud-Din, F., Schewe, K.-D. & Tretiakov, A. (2006), 'Towards algebraic query optimisation for XML', *Journal on Data Semantics* **VII**, 165–195.

Miller, D. & Nadathur, G. (1986), Higher-order logic programming, *in* 'Proceedings on Third international conference on logic programming', Springer-Verlag New York, Inc., New York, NY, USA, pp. 448–462.

Nadathur, G. (1987), A higher-order logic as the basis for logic programming, PhD thesis, University of Pennsylvania, Philadelphia, PA, USA.

Naqvi, S. & Tsur, S. (1989), *A logical language for data and knowledge bases*, Computer Science Press, Inc., New York, NY, USA.

World Wide Web Consortium (2004), 'Extensible markup language (XML) 1.0 (third edition)', W3C Recommendation. Editors: Tim Bray and Jean Paoli and C. M. Sperberg-McQueen and Eve Maler and François Yergeau. *http://www.w3.org/TR/2004/REC-xml-20040204/

World Wide Web Consortium (2005*a*), 'XML query use cases', W3C Working Draft. Editors: Don Chamberlin and Peter Fankhauser and Daniela Florescu and Massimo Marchiori and Jonathan Robie. *http://www.w3.org/TR/2005/WD-xquery-use-cases-20050915/

World Wide Web Consortium (2005*b*), 'XQuery 1.0: An XML query language', W3C Working Draft. Editors: Scott Boag and Don Chamberlin and Mary F. Fernández and Daniela Florescu and Jonathan Robie and Jérôme Siméon. *http://www.w3.org/TR/2005/WD-xquery-20050404/