LOGIC BASED QUERIES FOR XML DATABASES

By

Qing Wang

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF INFORMATION SYSTEMS
AT
MASSEY UNIVERSITY
PALMERSTON NORTH, NEW ZEALAND
DECEMBER 2005

MASSEY UNIVERSITY
DEPARTMENT OF
INFORMATION SYSTEMS

The undersigned hereby certify that they have read and recommend to the Department of Information Systems for acceptance a thesis entitled "**Logic Based Queries for XML Databases**" by **Qing Wang** in partial fulfillment of the requirements for the degree of **Master of Information Systems**.

Dated: December 2005

Supervisor: _____
Klaus-Dieter Schewe

Readers: _____

_____

MASSEY UNIVERSITY

Date: **December 2005**

Author: **Qing Wang**

Title: **Logic Based Queries for XML Databases**

Department: **Information Systems**

Degree: **M.I.S.** Convocation: **February** Year: **2006**

Permission is herewith granted to Massey University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

Table of Contents

Table of Contents iv

Acknowledgements vii

1 Introduction 1
- 1.1 XML Data Challenges . . . 1
- 1.2 Objectives . . . 3
  - 1.2.1 The Semantic XML Object Model . . . 3
  - 1.2.2 The XML Identity Query Language . . . 4
  - 1.2.3 The XML Calculus Language . . . 4
- 1.3 Principals . . . 5
  - 1.3.1 Object Bases . . . 5
  - 1.3.2 Logic Paradigm . . . 6
- 1.4 Overview of the Thesis . . . 6

2 The Semantic XML Object Model 8
- 2.1 Object Types and Objects . . . 8
- 2.2 XML Databases on Object Bases . . . 12
  - 2.2.1 Database Instances . . . 13
  - 2.2.2 Class Schemata and Tuples . . . 14
  - 2.2.3 Database Schemata . . . 19
- 2.3 The Semantics of Objects . . . 22
- 2.4 Object Connectivity . . . 23
  - 2.4.1 Dominant Relation . . . 23
  - 2.4.2 Class Hierarchy . . . 25
  - 2.4.3 Object Order . . . 26
- 2.5 Another Representation of Objects . . . 27

3 XML Identity Query Language: A Framework 30
- 3.1 Syntax . . . 30
- 3.2 Semantics . . . 33
  - 3.2.1 Valuations and Satisfaction . . . 34

3.2.2 Semantics of Queries . . . 36
3.2.3 Union and Intersection Types . . . 37
3.2.4 Inflationary Fixpoint Operator . . . 40
3.2.5 Safe Queries . . . 41
3.3 Object Creation Mechanism . . . 44
3.4 Key Characteristics and Examples . . . 47
3.4.1 Nest and Unnest . . . 47
3.4.2 Structural Recursion . . . 49
3.4.3 Path Expressions . . . 50
3.4.4 Schema Query . . . 51
3.4.5 Join Operation . . . 52
3.4.6 Sequences . . . 53
3.4.7 Function Symbols . . . 54

4 Further Issues on XML Identifier Query Language 55
4.1 Duplicate Elimination . . . 55
4.1.1 Structured Value Duplicates . . . 56
4.1.2 A Main Result . . . 57
4.2 Copy Elimination . . . 61
4.3 Database Transformations . . . 63
4.3.1 Determinate Transformations . . . 64
4.3.2 Hybrid-deterministic Transformations . . . 65
4.3.3 Completeness . . . 66

5 The XML Calculus Language 67
5.1 Preliminaries . . . 67
5.1.1 A Fundamental Type System . . . 68
5.1.2 Higher-order Constructs . . . 70
5.2 Syntax . . . 72
5.2.1 Terms and Variables . . . 72
5.2.2 Well-Formed Formulae . . . 73
5.2.3 Free and Bound Variables . . . 74
5.3 Semantics . . . 75
5.3.1 Interpretation . . . 76
5.3.2 Query Mapping . . . 78
5.3.3 Object Creation . . . 78
5.3.4 Active Domain Semantics . . . 79
5.4 Logical Reflection . . . 80
5.5 Simulation . . . 84
5.5.1 Recursive structures . . . 85
5.5.2 Nest and Unnest . . . 85
5.5.3 If-Then-Else . . . 88

5.5.4 Class Names and Atomic Values 89
5.5.5 Queries over Object Order 90
5.5.6 Aggregation Functions 90
5.5.7 Duplicate Elimination 91

6 Related Work **92**
6.1 Semantic Data Models 92
6.2 Object-Creating Languages 93
6.3 Pure Declarative Languages 94

7 Conclusions **96**

A Running XML Database **98**

Bibliography **101**

Acknowledgements

I would like to express my gratitude to the many people who have made this thesis possible.

Most of all I would like to thank Professor Klaus-Dieter Schewe, my supervisor, who recommended this interesting research topic to me when I was in confusion at the early stage of Masters study, and introduced me to the world of logic theory. Without his patient guidance and stimulating suggestions, I would have been lost somewhere. I am also very grateful to Dr. Sven Hartmann for being a constant source of support and encouragement at all times, which greatly inspired me to overcome many difficulties in my life.

Special thanks go to Thu Trinh and Madre Chrystall. They not only dedicated their time to being a reader of my manuscript, but also provided many helpful comments. Furthermore, I wish to thank all the many teachers, friends and classmates who helped me get where I am today although it is impossible to mention them here individually.

In particular, I wish to convey my thanks to Mrs Clark, Massey University, and Mr Todd and the NZVCC , for being awarded the Lovell & Berys Clark Scholarship, Massey University Masterate Scholarship and Todd Award for Excellent in 2005. These scholarships are crucial to the successful completion of this research because they not only free me from financial worries, but also motivate me towards higher goals for this research.

Lastly, and most importantly, I am forever indebted to my family for their understanding, endless patience and encouragement when it was most required, especially my three-year-old son Jiajun, he always seems quite understanding toward my study although he is not really aware of that.

Wang, Qing
December 1, 2005

Chapter 1

Introduction

With the significant increase of web-based applications, the eXtensible Markup Language (XML), as a de facto standard for data interchange on the web, has attracted considerable attention in theory and practice. XML deals with irregular and heterogeneous semi-structured data that give rise to trees, i.e. the predominant data structure in complex data models. This leads to new challenges for database research such as new data structures and models, and new database query languages in this area.

1.1 XML Data Challenges

In essence, the major challenge of XML lies in its data structure, which is greatly different from the traditional data structure treated as relations. A comparison with respect to the differences between XML data and relational data has been discussed in [17]. Generally speaking, XML data has the following unique characteristics:

- XML data have irregular and heterogeneous structures leading to optional values which are absent in relational data;
- Metadata that describes the structure of the data is distributed throughout the data in the form of markup, instead of being stored separately as in relational data;
- XML data is nested in a hierarchy, and complex nested structures might need to be decomposed and reconstructed on the fly to facilitate data manipulations;
- XML data are encoded with an intrinsic order that is an important property of data themselves. This means that the order property must be taken into consideration for modelling.

These features require more complicated data models in comparison to the relational data model, which merely deals with flat and rigid relations. In particular, problems such as rational trees, optional cardinality, repeated elements and sequence order demand additional functionalities to be provided to handle XML data in an efficient and sufficient manner.

Since the unique structures of XML data further require that XML query languages must have the capability for querying over schema information. In order to support this extra functionality at the database level, the techniques used in semantic data models have shown to be an appropriate approach, in which the separate notions of schema and instance are combined into an explicit expression. This allows the queries of schema information to be handled in the same manner as querying data at the language level.

Therefore, the first task investigated in this thesis is to design a simple and natural data model, which supports a rich description of XML data from a structural point of view. The relevant techniques for this are derived from semantic data models.

In the theory of relational databases, query languages have been developed mainly following three directions: an algebraic direction e.g. the Relational Algebra, a logical direction e.g. the Relational Calculus, and a logic programming direction e.g. deductive languages such as DATALOG. Many issues such as recursion, negation and the combination of them have been well investigated in relational query languages. However, some new problems have emerged with respect to XML query languages.

- the possible absence of a database schema;
- the capability to support querying over schema information;
- restructuring and creating XML data in a flexible approach;
- recursion under the feature of optionality.

Not surprisingly, the second task investigated in this thesis aims at developing declarative and powerful XML query languages in the context of XML data. My focus in this thesis is on a logical direction and a logic programming direction.

1.2 Objectives

The main goal of this thesis is to investigate the logical grounds of query languages over XML databases on the basis of a representation by object bases. In general, there are three objectives I want to achieve.

–Define a data model at the conceptual level for XML data aiming at capturing semantic capabilities on object bases.

–Develop the formal syntax and semantics for a logic programming query language for XML, which exploits object identifiers as primitives in the context of XML data structures.

–Propose a higher-order predicate calculus language tailored to XML data by studying the mathematical techniques used in relational calculus which is regarded as a specification of the first-order predicate calculus.

To clarify how the objectives are fulfilled in this thesis, we provide a brief introduction regarding each objective in this subsection.

1.2.1 The Semantic XML Object Model

The semantic XML object model aims at a natural and simple data model for XML-based databases that can capture and express XML data from a relatively high abstraction level with well-defined semantics. Furthermore, it can also serve as a formal specification for further establishing XML data manipulation languages.

The proposed semantic XML object model mainly has the following key features:

- It provides considerably rich structuring capability by using objects as basic semantic data abstractions that are associated with concise but flexible interpretations.
- It provides classes as the only construct to accommodate objects, thus enabling richness through class tuples and class schemata that link XML database instances and XML database schemata together.

- It provides attributes at the object level, which are used to incorporate values and objects into a unit, and attributes at the class level, which facilitate to capture objects having common semantics but variant structures within a specific class schema.
- It provides object identifiers that are used in a dual role as both structure and order primitive.

1.2.2 The XML Identity Query Language

The XML Identity Query Language (XIQL) is greatly influenced by the spirit of IQL in [6, 7] on a key design point that objects can be manipulated through having object identifiers as a powerful programming primitive. New objects can be created in a manner that is essentially equivalent to Skolem function techniques. However, more work needs to be done with respect to IQL to reflect the characteristics of XML data.

Compared with IQL, XIQL provides the following additional functionalities:

- XIQL incorporates the capability for type creation in queries. On one hand new types can be easily added into a database, while on the other hand types can be queried as a special part of an instance.
- XIQL enriches the language with union types, which can capture the optionality feature of XML data. Further, the underlying intersection types facilitate the flexible expressions for objects with class atoms.
- XIQL treats set and non-set variables in a unified fashion unless a particular interest is indicated. In this case, a set operator can be used to handle the transformation between set and non-set variables.

1.2.3 The XML Calculus Language

The XML calculus language is a pure declarative language incorporating higher-order logics on the basis of the SXO model, in which semantic structures can be elegantly encapsulated into objects.

The design goal of the XML calculus language is to obtain highly expressive power, but using a relatively simple interpretation. To achieve this goal, the combination of a higher-order syntax and a first-order semantics within a language becomes a suitable choice for the proposed XML query language. Furthermore, through a connection of two semantics of objects, an equation between the first-order and higher-order interpretations can be established.

To handle higher-order logics, it is essential to define a fundamental type system in the language, on which higher-order notions can be precisely developed. Moreover, for an XML query language, the ability to query schema information must be taken into consideration. For this reason, type variables are introduced into the type system.

Another interesting design point is to naturally reflect fixpoint semantics in this language by means of universal quantification, which is associated with variables under restricted domains. With this capability, the expressive power of the language will be greatly enhanced.

1.3 Principals

Throughout this thesis two main principles will be followed: considering XML databases as object bases and following a logic paradigm for querying them.

1.3.1 Object Bases

The data model developed in my thesis is built upon object bases. Instead of encapsulating behavioral aspects of objects in object-oriented databases, this data model concentrates on encapsulating structural aspects from a semantic point of view. The reasons for preference for an object-based model, rather than a relational model, principally arise from the following considerations.

Firstly, as argued in [63], the relational data model fails to capture much of the semantics associated with data because the fundamental modelling construct, i.e. the tuple does not constitute an atomic semantic unit. Hence, additional integrity constraints are required to establish the intended semantics of the database. In contrast to tuples, the features of objects determine themselves to be the natural semantic carriers in XML databases.

Secondly, to tackle the irregular and heterogenous structure of XML data, objects grouped

in a class can provide richer structure modelling facilities than tuples in a relation. By specifying classes, which represent collections of objects encoded with heterogenous structures, a uniform framework on object bases can be established.

Finally, it is customary that XML data has a graphical representation, which is also thought of as a representation for objects in the object-based model. Hence, they coincide with respect to this intuition. Furthermore, both objects and graphs can be constructed to be self-descriptive, which is an important feature identified in XML data. However, a tabular representation provided in the relational data model is not suitable in this case due to the separation of schema and instance.

In addition, although many XML query languages have been proposed such as XQuery [73], XQL [56], LOREL [8] and XML-QL [29], most of them aim at providing the XML-era analogue of SQL. However, I believe that the investigation on XML query languages from a perspective of object bases has not been sufficiently conducted yet so far, especially for exploiting object identifiers as structure and order primitives.

1.3.2 Logic Paradigm

In the thesis, a study towards logic-based query languages in the framework of an object-based XML data model will be conducted. The reason for considering the logical ground of query languages is based on the following two points:

- Firstly, logic-based approaches have been recognized as a means to provide remarkable simplicity and conciseness of syntax for query languages, as were the case in relational calculus and Datalog for the relational data model.
- Furthermore, logic-based approaches are essential for evaluating the expressive power and computable complexity of query languages.

Most of the current XML query languages focus on path-based query processing with pattern matching techniques. Research regarding theory, well-defined semantics and expressiveness is still needed.

1.4 Overview of the Thesis

The remainder of the thesis is organized according to the following logical sequence.

In Chapter 2, we focus on modelling XML data in a natural way by applying techniques of semantic data model. The developed data model, called the SXO model, is built upon 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. At the end of the chapter, some insights are given into dominant relation, class hierarchy and object order. Based on this model, XIQL, a simple and powerful logic programming language for manipulating XML data, is proposed in Chapter 3. A basic framework of this language with the formal syntax and semantics is presented there. Furthermore, we discuss the object creation mechanism adopted in XIQL in details. Chapter 4 investigates the issues about structured value duplicates and copies. It turns out that XIQL is complete with respect to determinate transformation since both structured value duplicates and copies can be eliminated in XIQL queries. To obtain a pure declarative XML query language, Chapter 5 incorporates higher order logics into a novel XML query language called XML calculus as a counterpart of relational calculus in the relational data model. This language is developed with higher-order syntax and first-order semantics. After introducing the formal syntax and semantics of the language, a logical reflection regarding the relationship between first-order semantics and higher-order semantics is discussed. Chapter 6 reviews the literature from three aspects: semantic data models, object-creating languages and pure declarative languages, as a complement to some related work having been introduced during Chapter 2, 3, 4 and 5. In the end, we summarize the main results exploited in this thesis in Chapter 7. Several issues left for future work are also identified at the end of Chapter 7.

Throughout the thesis we will use a simple XML database as presented in Figure A.1, Figure A.2 and Figure A.3 of Appendix A. All the data stem from XML Query Use Case [72] provided by the World Wide Web Consortium (W3C) with some minor modifications. The purpose of the running XML database is to apply the proposed query languages to specific application scenarios, and illustrate the capabilities of our languages.

Chapter 2

The Semantic XML Object Model

This chapter introduces a data model specific to XML, called the Semantic XML Object Model (SXO model), which is based on tree structures and has objects as semantic units. This data model attempts to capture more properties of XML databases at a conceptual level. We begin by presenting formal definitions of object types and objects, then show how XML database instances and schemata can be obtained under the SXO model with a rich description of characteristics. After that, the semantics of objects are investigated to provide a well-defined semantic framework. Furthermore, we are interested in discussing object connectivity from multiple aspects: dominant relation, class hierarchy and object order. In the end, another representation of objects which can be expressed under a value-based model is introduced to obtain a link between the SXO model and a value-based model.

The main purpose of this chapter is to present a data model for XML, which is as simple and natural as possible, so that further data manipulation languages can be built up in the following chapters.

2.1 Object Types and Objects

There are two kinds of properties considered in the SXO model, atomic values and objects, which are two distinct concepts that have been clearly discussed in [13]. In the SXO model, objects are adopted to represent basic semantic abstractions reflecting a natural understanding for objects in the real world, whereas atomic values can only exist as fundamental elements which must be appropriately incorporated into objects as properties. More specifically, semantic objects serve as the basic units of persistent data within a database on the one

hand, while on the other hand, first-order relationships are encapsulated into object values so that a structural association among objects can be captured in a semantic framework.

To precisely describe atomic values and objects, we start by introducing two basic concepts: atomic types and class names, which are developed as follows.

Atomic types: an atomic type is a non-decomposable type unit, including STRING, INT, NAT, BOOL and etc. Atomic types are base types.

Class names: a class name indicates a class serving as a container for a finite set of objects.

For the sake of conciseness, we adopt the notations $\mathbb{B}$ for a countably infinite set of atomic types, $\mathbb{C}$ for a countably infinite set of class names and $\mathcal{A}$ for a countably infinite set of attribute names.

Attribute names are either atomic types or class names in the SXO model. Therefore, in accordance with this distinction, attributes can be categorized into two kinds: value attributes that have atomic types and object attributes that have 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. Correspondingly, 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 $\{dom_{V_1}, ..., dom_{V_n}\}$, and $\mathcal{I} = \{dom_{O_i}\}_{i\in[1,n]}$ be a fixed family of object domains $\{dom_{O_1}, ..., dom_{O_n}\}$, respectively.

Instead of distinguishing between entities and relationships as in the ER model, the SXO model considers objects in a simple and unified expression. An object may be classified as being either atomic or complex in accordance with its attributes. Atomic objects have only value attributes, while complex objects have at least one object attribute. Let $\mathcal{T}$ be a set of object types including atomic object types and complex object types (as defined below), then there is a class domain assignment $Dom_C : \mathcal{T} \rightarrow \mathcal{I}$ such that each object type $OT \in \mathcal{T}$ is associated with its domain $Dom_C(OT)$.

Definition 2.1.1. (Atomic object type) An *atomic object type* AT consists of

(i) a class name $nam(AT) \in \mathbb{C}$;

(ii) a finite, non-empty set of value attributes $attr_V(AT) = \{A_1, ..., A_n\}$, where $A_k \in \mathbb{B}$ for $k \in [1, n]$;

(iii) a *value domain* assignment $Dom_V : attr_V(AT) \rightarrow \mathcal{D}$ which associates with each attribute $A \in attr_V(AT)$ its domain $Dom_V(A)$.

The expression $AT = nam(AT)[attr_V(AT)]$ is used to express an atomic object type AT in a class with name $nam(AT)$ having a finite set of value attributes $attr_V(AT)$.

Definition 2.1.2. (Atomic object) An *atomic object* of type AT is an expression $c(i, u)$ consisting of

(i) a class name c, where $c = nam(AT)$;

(ii) an object identifier i such that $i \in Dom_C(AT)$;

(iii) an object value $u = [A_1 : t_1, ..., A_n : t_n]$, where $\{A_1, ..., A_n\} = attr_V(AT)$ and $t_k \in Dom_V(A_k)$ $(1 \leq k \leq n)$.

Definition 2.1.3. (Complex object type) A *complex object type* CT consists of

(i) a class name $nam(CT) \in \mathbb{C}$;

(ii) a finite, non-empty set of object attributes $attr_O(CT) = \{A_1, ...A_m\}$, where $A_k \in \mathbb{C}$ for $k \in [1, m]$;

(iii) a finite set of value attributes $attr_V(CT) = \{A_{m+1}, ..., A_n\}$, where $A_k \in \mathbb{B}$ for $k \in [m + 1, n]$;

(iv) a *value domain* assignment $Dom_V : attr_V(CT) \rightarrow \mathcal{D}$ which associates with each attribute $A \in attr_V(CT)$ its domain $Dom_V(A)$;

(v) an *object domain* assignment $Dom_O : attr_O(CT) \rightarrow \mathcal{I}$ which associates with each attribute $A \in attr_O(CT)$ its domain $Dom_O(A)$.

The expression $CT = nam(CT)[attr_V(CT) \cup attr_O(CT)]$ is used to express a complex object type CT in a class with name $nam(CT)$ having a finite set of value attributes $attr_V(CT)$ and object attributes $attr_O(CT)$.

Definition 2.1.4. (Complex object) A *complex object* of type CT is an expression $c(i, u)$ consisting of

(i) a class name c, where $c = nam(CT)$;

(ii) an object identifier i such that $i \in Dom_C(CT)$;

(iii) an object value $u = [A_1 : t_1, ..., A_n : t_n]$, where $\{A_1, ..., A_n\} = attr_V(CT) \cup attr_O(CT)$, and $t_k \in Dom_V(A_k)$ iff $A_k \in attr_V(CT)$ or $t_k \in Dom_O(A_k)$ iff $A_k \in attr_O(CT)$ $(1 \leq k \leq n)$.

With respect to the preceding definitions, two things should be pointed out here. First of all, each object must belong to exactly one class. That is, given two object types OT_1 and OT_2, where $nam(OT_1) = c_1$, $nam(OT_2) = c_2$, and $c_1 \neq c_2$, then it implies that $dom_C(OT_1) \cap dom_C(OT_2) = \emptyset$. Secondly, recursive types may occur in complex object types since complex objects may be constructed from objects in the same class.

Intuitively, atomic objects function as wrappers of atomic values, while complex objects encode more complex structures among other objects. Correspondingly, based on this intuition, classes can 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 stems from whether object identifiers as structure primitives are involved in object values within the class.

In contrast to the notion of semantic object defined in [46], where an object identifier is defined by one or more attributes that the users employ to identify object instances, we adopt invisible and immutable object identifiers by following the idea used in the classical object approaches.

One of the main reasons for employing semantic objects in the SXO model is to support heterogeneous structures embedding within objects. To fully accomplish this intention, the notion of *object value type* is developed to capture the variant structures in object values.

Definition 2.1.5. (Object Value Type) Given an object o with value u, then there exists a function τ mapping from the object o to object value type $\tau(o)$ containing a set of distinct attribute names in value u, such that if $u = [A_1 : t_1, ..., A_n : t_n]$, 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. $Dom_V(\{A\}) = \{\{v_1, \ldots, v_k\} \mid k \in \mathbb{N}, v_i \in Dom_V(A) \text{ for all } 1 \leq i \leq k\}$ or $Dom_O(\{A\}) = \{\{v_1, \ldots, v_k\} \mid k \in \mathbb{N}, v_i \in Dom_O(A) \text{ for all } 1 \leq i \leq k\}$.

Note that both the order of attributes in an object value and the order of attribute names in an object value type are not significant. However, attributes in an object value might not be distinct, while attribute names must be unique over an object value type. The following example is given to illustrate these notions.

Example 2.1.1. For the object $o = c(i, [A_1 : t_1, A_2 : t_2, A_3 : t_3, A_2 : t_4])$, we obtain $\tau(o) = [A_1, \{A_2\}, A_3]$.

To clearly express objects in a graph representation, we specify that atomic objects are represented by ovals, while complex objects are represented as rectangles.

Remark 2.1.1. Finally, we give some comments on the notion of object developed in the SXO model. Although all objects are described with a simple and unified form such that $c(i, u)$, they can be interpreted from multiple points of view following the same line of semantic data models. As stated in [61], an object actually can be regarded as a real object, or a property of objects, or a relationship among objects. In essence, the flexibility of interpretation enriches the semantic abstraction embedded with objects.

2.2 XML Databases on Object Bases

From the database point of view, XML data can be represented with tree structures (we do not consider references here). Thus, in this section, our interest is to discuss XML database instances and schemata on the basis of the concepts introduced before. The desired goal is to establish a counterpart of relational databases in XML databases, which can serve to further develop data manipulation languages.

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

- A *name projection* for O is a function σ restricting each object expression $o = c(i, u)$, where $o \in O$, to the object's class name c.
- An *identifier projection* for O is a function δ restricting each object expression $o = c(i, u)$, where $o \in O$, to the object's identifier i.

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

- A *structure projection* for O is a function η restricting each object expression $o = c(i, u)$, where $o \in O$, to the finite set of object identifiers contained in object value u.

Example 2.2.1. Assume an object $o = c(i, u)$, where $u = [A_1 : i_1, A_2 : v_2, A_3 : i_3, A_4 : v_4]$ with $i_1, i_3 \in \mathcal{I}$ and $v_2, v_4 \in \mathcal{D}$, then we can get $\sigma(o) = c$, $\delta(o) = i$, $\pi(o) = u$ and $\eta(o) = \{i_1, i_3\}$.

2.2.1 Database Instances

For the sake of simplicity, only element, attribute, text and document nodes are considered in the formalization. Other nodes, including namespace, processing instruction and comment nodes are not relevant for our purposes here and therefore omitted.

Let e, a, b and p represent element, attribute, document and text nodes in XML data respectively, and let o, d represent objects and atomic values in the SXO Model. 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 or document node. Therefore, we can model XML data in the following manner:

- Each text node p will be modelled as an atomic value d with $d = content(p)$.

- Each element node e will be modelled as an object o with $\sigma(o) = name(e)$ and $\pi(o) = [A_1 : subelement_1(e), ..., A_m : subelement_m(e), A_1' : content(p_1), ..., A_n' : content(p_n)]$, where $m \geq 0$, $n \geq 0$, $A_i \in \mathbb{C}$ for $i \in [1, m]$ and $A_j' \in \mathbb{B}$ for $j \in [1, n]$. $subelement_1(e)$, ..., $subelement_m(e)$ are a 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 $A \in \mathbb{B}$.

- The document node b will be modelled as an object o with $\sigma(o) = root$ and $\pi(o) = [A : subelement(b)]$, where $A \in \mathbb{C}$.

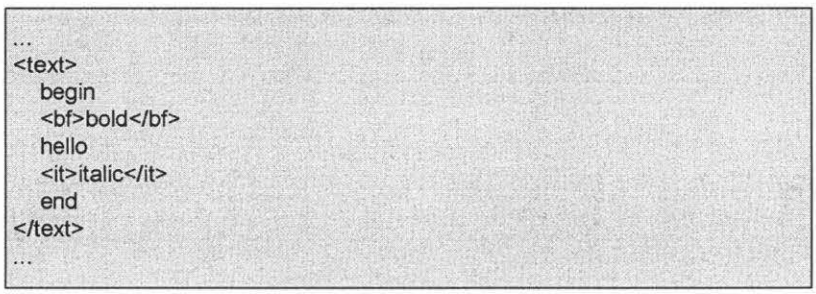


Figure 2.1: A fragment of XML document

Example 2.2.2. Consider a fragment of XML document shown in Figure 2.1, it shows a mixed content element node of *text*, which can be modelled as an object: $text(i, [STRING : \text{"}begin\text{"}, bf : i_1, STRING : \text{"}hello\text{"}, it : i_2, STRING : \text{"}end\text{"}])$. The element nodes *bf* and *it* correspond to the objects $bf(i_1, [STRING : \text{"}bold\text{"}])$ and $it(i_2, [STRING : \text{"}italic\text{"}])$.

Throughout this chapter, we use V_G, E_G to denote the finite sets of vertices and edges for a graph G respectively, such that $V_G = \{v_1, ...v_n\}$ and $E_G = \{(v_i, v_j) | v_i, v_j \in V_G\}$.

Definition 2.2.1. (XML Data Graph) Let O denotes a finite set of objects modelling XML data. An XML data graph G over O is a rooted and directed graph where $V_G = \bigcup_{o \in O} \sigma(o)$ and $(\sigma(o) : \sigma(o_i)) \in E_G$ if and only if $\delta(o_i) \in \eta(o)$.

Example 2.2.3. The XML data graph for the XML document book.xml is shown in Figure 2.2. Note that object identifiers do not appear in an XML data graph since they are invisible to the users. For the sake of clarity, we indicate objects in our examples, and assume that each object o_k has an object identifier i_k, where $k \in \mathbb{N}$.

Correspondingly, an XML document can be regarded as an XML data graph, and an XML database instance is a collection of XML data graphs.

2.2.2 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.

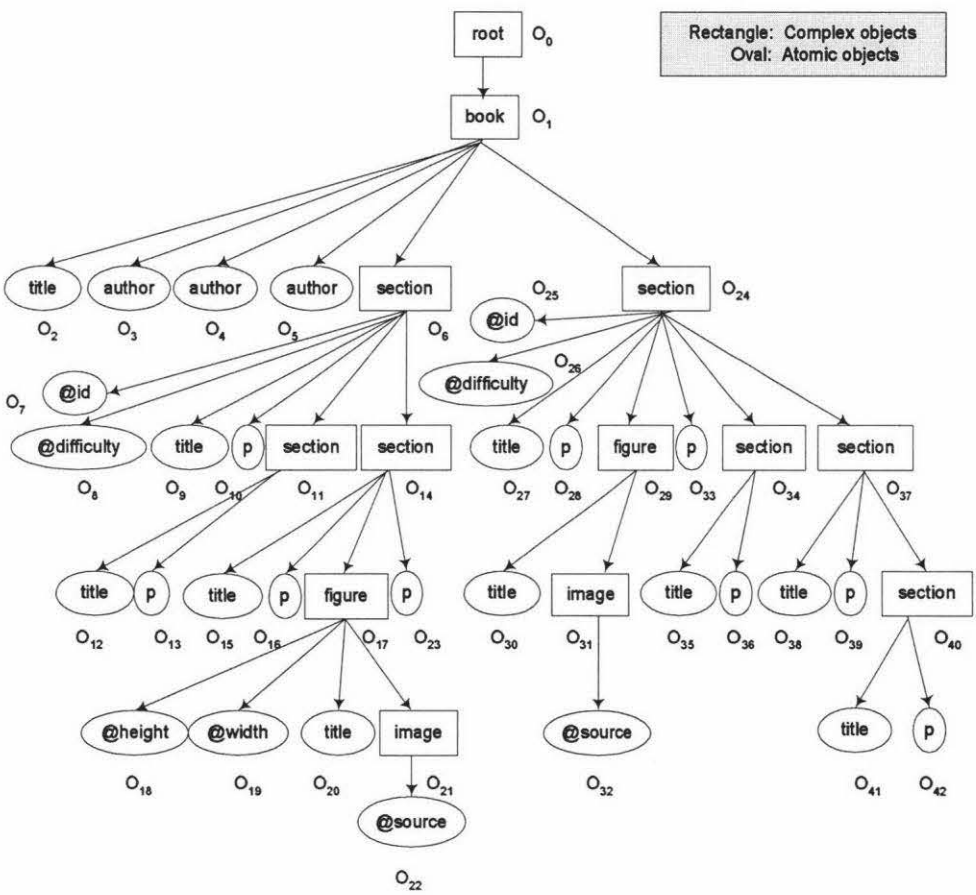


Figure 2.2: *An XML data graph*

The fundamental idea is, for a set of objects collected into a class, to obtain class schemata by generalizing their object value types with optional superscript characters $\{?, +, *\}$ [70], and then treating their object values appropriately as class tuples over the specified class schema. One thing we should be aware of is that this approach builds on the fact that each object can only belong to one class in the SXO model since class names have been explicitly encoded into object expressions.

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 an 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 $\{?, +, *\}$ has 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;
$edom(A^+) = \mathcal{P}(dom(A)) - \{\emptyset\}$, where $\mathcal{P}$ denotes powersets.

Similar to the relational data model, there are two alternatives for specifying class schemata and tuples: anonymous and named perspectives.

Definition 2.2.2. (Class Schema)

(i) An *anonymous class schema* $\dot{s}_c$ of a class with name c consists of a finite set of distinct implicit attribute names with superscript characters $\{A_1^{f_1}, ..., A_n^{f_n}\}$, where $f_i \in \{1, ?, +, *\} (1 \leq i \leq n)$, having fixed order given by the bijective function g: $\{1, ..., n\} \rightarrow \{A_1^{f_1}, ..., A_n^{f_n}\}$ and an assignment of domains $\{1, ..., n\} \rightarrow \bigcup_{1 \leq i \leq n} edom(g(i))$ such that $i \mapsto edom(g(i))$.

(ii) A *named class schema* s_c of a class with name c consists of a finite set of distinct explicit attribute names with superscript characters $\{A_1^{f_1}, ..., A_n^{f_n}\}$, where $f_i \in \{1, ?, +, *\} (1 \leq i \leq n)$, and 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 $A_i^{f_i} \mapsto edom(A_i^{f_i})$.

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

Definition 2.2.3. (Class Tuple)

(i) A *class tuple* t over an anonymous class schema $\dot{s}_c$ 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 $v_i \in edom(g(i))$ for $1 \leq i \leq n$.

(ii) A *class tuple* t over a named class schema s_c 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 2.2.4. Consider the class with name *section* in the XML document book.xml. In this case we obtain

- $\dot{s}_{section} = [\ ,\ ,\ ,\ ,\ ,\ ]$ with a set of implicit class attributes having fixed order $@id^?$, $@difficult^?, title, p^+, figure^?, section^*$, and
  $s_{section} = [@id^?, @difficult^?, title, p^+, figure^?, section^*]$.
- *A set of objects in the class is:* $\{o_6, o_{11}, o_{14}, o_{24}, o_{34}, o_{37}, o_{40}\}$.

The class tuples over $\dot{s}_{section}$ can be represented from the anonymous perspective.
$[i_7, i_8, i_9, \{i_{10}\}, \lambda, \{i_{11}, i_{14}\}]$;
$[\lambda, \lambda, i_{12}, \{i_{13}\}, \lambda, \emptyset]$;
$[\lambda, \lambda, i_{15}, \{i_{16}, i_{23}\}, i_{17}, \emptyset]$;
$[i_{25}, i_{26}, i_{27}, \{i_{28}, i_{33}\}, i_{29}, \{i_{34}, i_{37}\}]$;
$[\lambda, \lambda, i_{35}, \{i_{36}\}, \lambda, \emptyset]$;
$[\lambda, \lambda, i_{38}, \{i_{39}\}, \lambda, i_{40}]$;
$[\lambda, \lambda, i_{41}, \{i_{42}\}, \lambda, \emptyset]$.

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

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. For convenience, it suffices to restrict discussion of class schemata and tuples to complex classes in the following content if no specified demonstration provided.

Now the relations between individual object value types and a class schema are discussed to avoid confusion. Let $s = [A_1^{f_1}, ..., A_n^{f_n}]$ be a class schema and $\tau(o) = [B_1, ..., B_m]$ be an object value type for object o, then the object value type $\tau(o)$ is said to belong to the class schema s, denoted by $\tau(o) \sqsubseteq s$, if and only if the following conditions are satisfied.

- $n \geq m$;
- for each $B_i \in \tau(o)$, there exists some $A_j = B_i$ and $A_j^{f_j} \in [A_1^{f_1}, ..., A_n^{f_n}]$ such that

  (1) B_i is a set type, iff $A_j^{f_j} \in \{A_j^*, A_j^+\}$;

  (2) B_i is a non-set type, iff $A_j^{f_j} \in \{A_j^1, A_j^?\}$.

It is quite straightforward to see that all object value types of objects in a class should belong to the specific class schema of that class.

We end this section by introducing subsumption relations between object values and class tuples developed at the instance level.

Definition 2.2.4. (Subsumption Relation) Let $u = [A_1^{f_1} : t_1, ..., A_n^{f_n} : t_n]$ be a class tuple and $u' = [B_1 : d_1, ..., B_m : d_m]$ be an object value, where $m \geq n$, then u' is said to subsume u, denoted by $u' \succeq u$ if and only if, for each $A_i^{f_i} : t_i (i \in [1, n])$, one of the following conditions is satisfied.

- for $f_i = 1$, there is exactly one $d_j (j \in [1, m])$ over u' satisfying $A_i = B_j$ and $t_i = d_j$;
- for $f_i =?$, there is exactly one $d_j (j \in [1, m])$ over u' satisfying $A_i = B_j$ and $t_i = d_j$ when $t_i \neq \lambda$, or no $d_j (j \in [1, m])$ over u' satisfying $A_i = B_j$ when $t_i = \lambda$;
- for $f_i = +$, there is a non-empty set $\{d_{j_1}, ..., d_{j_n}\} (j_1, ..., j_n \in [1, m])$ over u' satisfying $A_i = B_j$ for $j \in \{j_1, ..., j_n\}$ and $t_i = \{d_{j_1}, ..., d_{j_n}\}$;

- for $f_i = *$, there is a non-empty set $\{d_{j_1}, ..., d_{j_n}\}(j_1, ..., j_n \in [1, m])$ over u' satisfying $A_i = B_j$ for $j \in \{j_1, ..., j_n\}$ and $t_i = \{d_{j_1}, ..., d_{j_n}\}$ when $t_i \neq \emptyset$, or no $d_j(j \in [1, m])$ over u' satisfying $A_i = B_j$ when $t_i = \emptyset$.

Furthermore, the subsumption relations can be easily extended to capture the relations between two class tuples, or two relational tuples. For instance, given two class tuples $u_1 = [B_1^{g_1} : d_1, ..., B_m^{g_m} : d_m]$ and $u_2 = [A_1^{f_1} : t_1, ..., A_n^{f_n} : t_n]$, $u_1 \succeq u_2$ if and only if, for each $A_i^{f_i} : t_i \in u_2$ $(i \in [1, n])$, the condition $A_i^{f_i} : t_i \in u_1$ must be satisfied. Alternatively, given two relational tuples $u_1 = [B_1 : d_1, ..., B_m : d_m]$ and $u_2 = [A_1 : t_1, ..., A_n : t_n]$, $u_1 \succeq u_2$ if and only if, for each $A_i : t_i \in u_2$ $(i \in [1, n])$, the condition $A_i : t_i \in u_1$ must be satisfied.

The underlying reason to develop the subsumption relations between object values and class tuples is to describe the flexible expressions associated with a class, especially in case that only partial schema information are obtained with respect to a class. Both languages presented in Chapter 3 and 5, respectively, shall use this concept to deal with their class constructs.

Remark 2.2.1. Classes play a crucial role in the SXO model. More accurately, although a class name is associated with a collection of object identifiers, the corresponding class schema has a capability to describe the heterogenous structures encoding of objects through a set of class tuples. Indeed, a class schema provides a general structural specification for object values of objects grouped in a class by allowing a variance in class attributes with operational characters. Furthermore, treating attributes in object types as class names or atomic types as appropriate leads to the fact that schemata and instances are mixed together into object expressions within the SXO model, completely different from the strict separation of schemata and instances in the relational data model.

2.2.3 Database Schemata

In the SXO model, objects are the only means of representing XML databases, and classes are employed to describe all objects. Thus, we treat class schemata as the sole building block for XML database schemata, a counterpart of relation schemata in the relational data model.

First of all, we adopt a notion of XML schema graph to describe XML database schemata.

Definition 2.2.5. (XML Schema Graph) Let S denote a finite set of class schemata. An XML schema graph G over S is a rooted and directed graph such that

- $V_G = \bigcup_{s_c \in S}(c \cup cattr(c))$, where s_c is a class schema for the class with name c, $cattr(c)$ is a set of attributes $\{A | A \in \mathbb{C}, A^f \in s_c\}$.

- $(c, A) \in E_G$ for each $s_c \in S$ and each class attribute $A^f \in s_c$, where $A \in \mathbb{C}$. Further, the edge (c, A) is labelled with f iff $f \in \{+, *, ?\}$.

An XML schema graph is *complete* if and only if, for each class schema in the graph, all class schemata corresponding to class names of its class attributes are also in this XML schema graph.

To be more precise, an algorithm called *Schema Reconstruction* is presented to show how an XML schema graph can be constructed from a finite set of class schemata. The principle of the algorithm is to start from any class schema, and then build an XML schema graph via a method called *AddClassSchema* on the basis of a set of class schemata S. Indeed, this algorithm provides a simple but efficient approach to construct an XML schema graph.

Algorithm: Schema Reconstruction
Input: A finite set of class schemata S for complex classes.
Output: An XML schema graph G.

Begin

Method `AddClassSchema (in:` $s_c \in S$`)`
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
IF $S_A \notin S$
THEN add a vertex labelled A
ENDIF;
add an edge e from c to A
IF $f \in \{+, *, ?\}$

THEN label the edge e with f
ENDIF;
ENDIF;
ENDFOR;

Main body
WHILE $S \neq \emptyset$
Get $s_c \in S$;
Do `addClassSchema`(s_c);
$S = S - \{s_c\}$
ENDWHILE.

End

An XML database schema is a collection of XML schema graphs.

Example 2.2.5. The Figure 2.3 shows the XML schema graph for the XML document book.xml. The following five complex class schemata can be obtained using a named perspective.

- $s_{root} = [book];$
  $s_{book} = [title, author^{+}, section^{+}];$
  $s_{section} = [@id^{?}, @difficult^{?}, title, p^{+}, figure^{?}, section^{*}];$
  $s_{figure} = [@height^{?}, @width^{?}, title, image];$
  $s_{image} = [@source].$

Comparing the XML data graph with the XML schema graph shown in Figures 2.2 and 2.3, respectively, it is quite straightforward to see that the XML schema graph is a homomorphic image of the XML data graph.

Definition 2.2.6. (Validity) Let G_d and G_s be an XML data graph and an XML schema graph respectively. Then, G_d is *valid* with respect to G_s iff the following conditions are satisfied.

(i) there exists a surjective graph homomorphism from G_d to G_s that preserves labels of vertices;

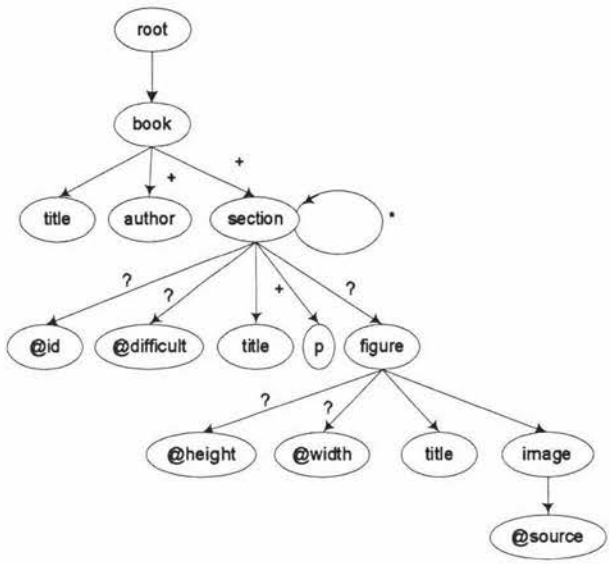


Figure 2.3: An XML schema graph

(ii) G_d satisfies all the constraints imposed by labels of edges in G_s.

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 $\mathbb{S}$.

Although the notions of schema and instance are separate in the preceding discussion regarding XML databases, they essentially differ from a counterpart in relational databases. The point is that schemata and instances are independent in relational databases, whereas schemata are dependent on instances in XML databases. More precisely, schemata can be inferred from instances in the SXO model by using the algorithm of schema reconstruction and a set of class schemata generalizing individual object value types of objects within an XML database.

2.3 The Semantics of Objects

In the SXO model, an object can be interpreted by different semantics according to identification. For instance, consider an object $car(i_c, [model : "Nissan", owner : "Mike", description : i_d])$. We can identify this object by its unique object identifier i_c, or by using a set of semantic descriptions such as $[model : "Nissan", owner : "Mike", description : i_d]$. In accordance

with these representations for an object, objects are considered to be in identifier semantics and value semantics, respectively.

Definition 2.3.1. (Object in Identifier Semantics) Given an object $o = c(i, u)$, then in identifier semantics there is an injective function $z : O \rightarrow \mathcal{I}$ from objects to identifiers, i.e. for any $i_1, i_2 \in \mathcal{I}$, where $z(o_1) = i_1$ and $z(o_2) = i_2$. If $i_1 = i_2$, then $o_1 = o_2$ holds.

Definition 2.3.2. (Object in Value Semantics) Given an object $o = c(i, u)$ with $u = [A_1 : t_1, ..., A_n : t_n]$, then in value semantics there is an injective mapping z between objects and complex values, i.e. for $o \mapsto [A_1 : t_1, ..., A_n : t_n]$ we have the condition that if $z(o_1) = z(o_2)$ and $\eta(o_1) = \eta(o_2) \neq \emptyset$, then $o_1 = o_2$ holds.

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.

Having two semantics for objects provides a flexible and powerful interpretation for XML data. In fact, it implies that the notion of object can be rich enough to support higher-order semantics since object values are normalized into first order semantics abstraction that can be reflected by object identifiers. We shall introduce two languages XIQL and XML calculus in the following chapters, and it is easy to see strong expressibility for data manipulation languages using object identity as primitives.

2.4 Object Connectivity

In this section, connectivity among objects shall be investigated from several aspects: dominant relation, class hierarchy and object order. Our intention is to capture more features of objects and to depict the relationships between objects in a precise manner.

2.4.1 Dominant Relation

There are a lot of approaches used in the literature [55, 33, 4, 61] to investigate the relationships among objects, such as generalization, classification, aggregation, etc. Here, we are more interested in capturing dominant relations between objects, and treat each object as associating other objects in its object family.

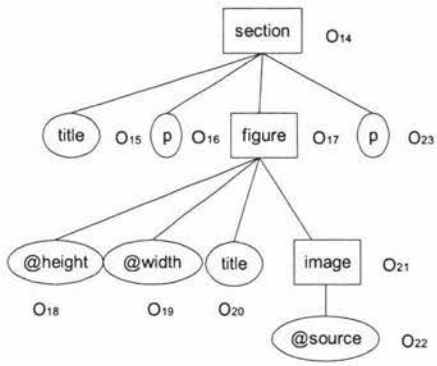


Figure 2.4: An object family graph

Definition 2.4.1. (Dominant Relation) Given objects o_1, o_2 and o_3 in an XML data graph, objects o_1 and o_2 have a dominant relation, denoted by $o_2 \measuredangle o_1$, if and only if

(i) $\delta(o_2) \in \eta(o_1)$, or

(ii) there exists object o_3 such that $o_2 \measuredangle o_3$ and $o_3 \measuredangle o_1$.

For the expression $o_2 \measuredangle o_1$, we also say that o_1 *dominates* o_2. A dominant relation between objects can be easily represented by a graph. For this reason, we define a notion of *object family graph* depicting all objects dominated by an object.

Definition 2.4.2. (Object Family Graph) Let o_d be an object, then the *object family* of o_d is $O^{\measuredangle}$, the finite set of all objects dominated by o_d. The object family graph G is an XML data graph over $O^{\measuredangle}$ rooted at the vertex labelled $\sigma(o_d)$.

Obviously, an object family graph is a rooted subgraph of an XML data graph.

Example 2.4.1. Let us consider object o_{14} in Figure 2.2, where $\delta(o_{15}) = i_{15}$, $\delta(o_{16}) = i_{16}$, $\delta(o_{17}) = i_{17}$, $\delta(o_{23}) = i_{23}$ and $\eta(o_{14}) = \{i_{15}, i_{16}, i_{17}, i_{23}\}$. This means that objects $o_{15}, o_{16}, o_{17}, o_{23}$ are dominated by o_{14} in accordance with condition (i) of Definition 2.4.1. Objects $o_{18}, o_{19}, o_{20}, o_{21}, o_{22}$ are dominated by o_{14} in accordance with condition (ii). Therefore, the object family for o_{14} is a set of objects $\{o_{15}, ..., o_{23}\}$, depicted by the object family graph shown in Figure 2.4.

2.4.2 Class Hierarchy

To show that classes in an XML graph constitute a hierarchical structure, we first need to define the subclass relations between classes. Let K be a set of class names in an XML graph.

Definition 2.4.3. (Subclass Relation) Given $c_1, c_2 \in K$, then c_1 is a subclass of c_2, denoted by $c_1 \leq c_2$, by using the follows rules inductively.

(i) $c_1 \leq c_1$;

(ii) $\perp \leq c_1$, where $\perp$ denotes the base class;

(iii) $c_1 \leq c_2$, where $c_2 = \sigma(o)$, $c_1 \in \tau(o)$ 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 SXO model can be easily obtained.

Theorem 2.4.1. *The partially ordered set* $(K \cup \{\perp\}, \leq)$ *is a complete lattice.*

Proof. Obviously, the subclass relation over $K \cup \{\perp\}$ is reflexive, transitive and antisymmetric, i.e. a partial order. For an XML graph, since the graph is rooted, there must exist a class that is a superclass for all classes in K. Similarly, the class $\perp$ with no objects defines a subclass of any other class. Furthermore, due to rules (i), (ii) and (iii) in Definition 2.4.3, any pair of classes c_1 and c_2 can have a least common superclass and a greatest common subclass in K. □

Example 2.4.2. The diagram for the subclass relation over the classes in the XML data graph shown in Figure 2.2 is presented in Figure 2.5.

The class *book* 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 (iii) of Definition 2.4.3. Similarly, we can obtain that the subclasses of class *section* are classes *figure*, *title*, *p*, *@difficult* and *@id*. Inductively using the rules of Definition 2.4.3, the diagram can be obtained as in Figure 2.5.

This kind of diagram is also called a class hierarchy for an XML data graph.

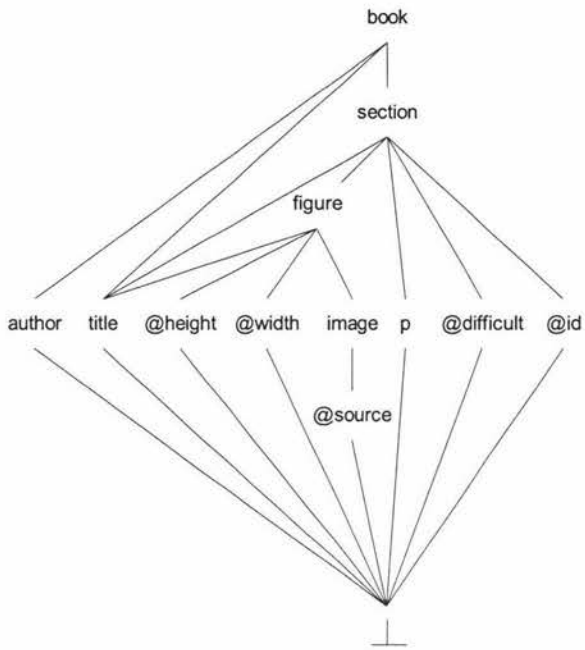


Figure 2.5: A class hierarchy

2.4.3 Object Order

In a more precise representation, XML data are treated as ordered trees. Although order is not an issue receiving significant attention in most situations, order should be taken into consideration as a distinguished feature of XML data.

Instead of set semantics for a collection of objects in unordered trees, objects are regarded as a sequence in a collection with list semantics due to order significance. However, for a sequence of objects $[o_1, ..., o_n]$, we can use a succinct expression $\{[i, o_i] : 1 \leq i \leq n\}$, where i is the order of object o_i, to convert list semantics into set semantics. One observation about this expression is that it is similar to assigning an object identifier to each object in a certain sequence. Therefore, the first impression is that object order and object identifers are at least similar in representations with respect to objects.

We know that object identifiers are invisible for users, and managed by the system. Individual object identifiers themselves are meaningless, as stated in [7], only interrelationships among object identifiers matter. What is the case for object orders? Surprisingly, we find out that they have the almost same feature; that the individual concrete orders need not be

provided to users, and only the relativity among them is 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 SXO model. An extra accompanying functionality with object identifiers being order primitives is that object identifiers can be involved in computations.

Example 2.4.3. Assume that there are two object $o_1 = (i_1, u_1)$ 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 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.

A benefit of our approach is that the order concern is naturally solved by object identifiers adopting a dual role such that object orders can be considered whenever needed, and ignored whenever we are not interested in them.

2.5 Another Representation of Objects

Although the SXO model is an object-based data model, it is quite easy to show how this model could be naturally linked to a value-based data model. In this section, we discuss a connection between the SXO model and a value-based data model which have different representations associated with each object. To fulfill this purpose, a new notion of object-structured value needs to be developed.

Before further formalization, the distinction between the notions of object value and object-structured value needs to be clarified. Each object o is always associated with an object value, denoted as $\widehat{o}$, and an object-structured value, denoted as $\widehat{\widehat{o}}$. The definition of object value having a flat form $c(A_1 : t_1, ..., A_n : t_n)$ was presented in Section 2.1, while the notion of object-structured value will be defined by a nested value encoding atomic values into structures in this section. Object values and object-structured values coincide for atomic objects.

Definition 2.5.1. (Object-Structured Value) Let o be an object with object value $\widehat{o} = c(A_1 : t_1, ..., A_n : t_n)$, then the associated object-structured value $\widehat{\widehat{o}}$ can be obtained by the following rule:

- Starting with the object value $\widehat{o}$, continuously replace object identifiers in the expression by the corresponding object values if objects are complex objects, or atomic values if objects are atomic objects, until no more object identifiers appear in the expression. The final result is the object-structured value $\widehat{\widehat{o}}$.

Alternatively, object-structured values can be represented by graphs, which are equivalent to expressions provided in the above definition. Let us look at an illustrative example.

Example 2.5.1. Consider o_{14} in the XML document book.xml, we obtain

- $\widehat{o}_{14} = [title : i_{15}, p : i_{16}, figure : i_{17}, p : i_{23}]$
- $\widehat{\widehat{o}}_{14} = [title : \text{"}Web\ Data\ and\ the\ Two\ Cultures\text{"},$
  $p : \text{"}T3\text{"},$
  $figure : [@height : \text{"}400\text{"},$
  $@width : \text{"}400\text{"},$
  $title : \text{"}Tranditional\ client/server\ architecture\text{"},$
  $image : [@source : \text{"}csarch.gif\text{"}]]$
  $p : \text{"}T4\text{"}]$

The corresponding graph representation for $\widehat{\widehat{o}}_{14}$ is shown in Figure 2.6.

Correspondingly, the notion of object-structured value type can be defined analogous to object value type. The only difference between them is that object value type is type of depth one, whereas object-structured value type is type of depth one or more.

Definition 2.5.2. (Object-Structured Value Type) Given an object o with object-structured value $\widehat{\widehat{o}}$, then there exists a function τ' mapping from the object o to object-structured value type $\tau'(o)$ containing a set of distinct nested attribute names in value $\widehat{\widehat{o}}$ obtained by applying the following rules.

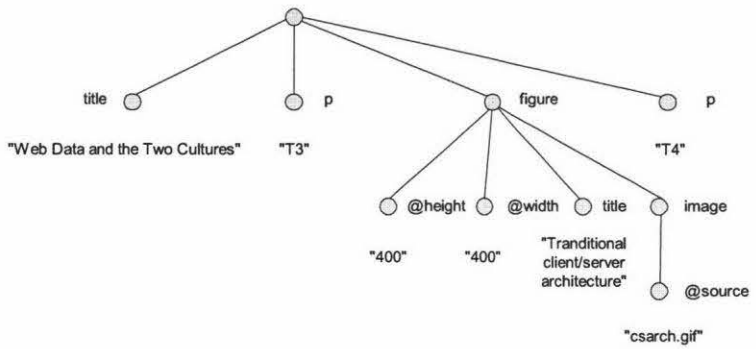


Figure 2.6: An object-structured value

- Starting with $\widehat{\widehat{o}}$, remove all atomic values and object identifers from the expression so that only attributes are left after that. In each tuple constructor, if more than one nested attributes are the exactly same, then they are merged into one nested attribute in a set type. The final expression represents the object value type $\tau'(o)$ corresponding to $\widehat{\widehat{o}}$.

To illustrate the difference between object value type and object-structured value type, both are given for the object appeared in the preceding example.

Example 2.5.2. Consider o_{14} in the XML document book.xml again. For this object, the associated object value type $\tau(o)$ and object-structured value type $\tau'(o)$ can be obtained such that

- $\tau(o_{14}) = [title, \{p\}, figure]$
- $\tau'(o_{14}) = [title, \{p\}, figure : [@height, @width, title, image : [@source]]]$

Remark 2.5.1. Obviously, when object identifiers are removed out of object values by replacing them with corresponding atomic values, objects have tree representations. Therefore, it is easy to see that objects in the SXO model correspond to subtrees in the value-based model. Nevertheless, objects have more flexibility than subtrees with respect to expressions since objects can be interpreted under both identifer semantics and value semantics, and subtree representations can be regarded as an extension which is a mixture of identifier semantics and value semantics.

Chapter 3

XML Identity Query Language: A Framework

In this chapter, a logic programming query language for XML will be investigated, which we call XML Identity Query Language (XIQL). In general, XIQL is developed with the following key characteristics:

1. the language is built on the Semantic XML Object Model (SXO Model);
2. it is motivated by the idea of IQL [7] to generate object identifiers during query evaluation;
3. function symbols are incorporated into terms to handle aggregation functions, sequence and schema query, which usually are not permitted in Datalog-like languages;
4. set and non-set variables are tackled in an unified and natural fashion so that it is unnecessary to distinguish them syntactically in the language.

This chapter aims at presenting a fundamental framework of XIQL, while some advanced issues are left to be further explored in Chapter 4.

3.1 Syntax

Since XIQL is established on the semantic data model as a theoretical foundation, the notations of the semantic data model and first-order logic will be used to formalize this language. Assume that the alphabet of XIQL is defined by a quintuple $\mathcal{H}=(\Sigma_u, \Sigma_x, \Sigma_c, \Sigma_f, \Sigma_R)$, where Σ_u,Σ_x,Σ_c,Σ_f and Σ_R are the pairwise disjoint sets of constants, variables, class names, function and relation symbols, respectively.

Terms: The *set of terms* is defined by the following rules:

- each *type term* τ is either a class name c or a type variable $\&x$;
- each constant u of type τ is a term of type τ, and each variable x of type τ is a term of type τ;
- for a term t of non-set type τ, $\tau : t$ is a term of type τ, while for a term t of set type $\{\tau\}$, $\tau : t$ is a term of type $\{\tau\}$;
- for a term t of non-set type τ_1, $\tau_2 \hookrightarrow t$ is a term of type τ_2, while for a term t of set type $\{\tau_1\}$, $\tau_2 \hookrightarrow t$ is a term of type $\{\tau_2\}$;
- each object o in a class c with identifier i, then
  - i is a term of type c,
  - $\widehat{i}$ is a term of type $\tau(o)$, where $\tau(o)$ is the object value type defined in Definition 2.1.5,
  - $\widehat{\widehat{i}}$ is a term of type $\tau'(o)$, where $\tau'(o)$ is the object-structured type defined in Definition 2.5.2;
- each relation R with terms $t_1, ..., t_n$ of type $\tau_1, ..., \tau_n$ as attributes is a term of type $\{[\tau_1, ..., \tau_n]\}$;
- each class c is a term of type $\{c\}$;
- each function f with terms $t_1, ..., t_n$ as arguments is a term of base type, where base types refer to atomic types defined in Section 2.1;
- for terms t_1, t_2 of type τ_1 and τ_2, respectively, $t_1.\, t_2$ is a term of type τ_2;
- for terms t_1, t_2 of type τ_1 and τ_2, respectively, $t_1 \vee t_2$ is a term of type $\tau \in \{\tau_1, \tau_2\}$;
- for $t_1, ..., t_n$ terms of type τ, $\{t_1, ..., t_n\}$ is a term of type $\{\tau\}$;
- for $t_1, ..., t_n$ terms of type $\tau_1, ..., \tau_n$, $[t_1, ..., t_n]$ is a term of type $[\tau_1, ..., \tau_n]$.

Note that $\hookrightarrow$ is an explicit typing operator developed to enhance the flexibility of types on terms. The term $\tau_2 \hookrightarrow t$ means that the term t is typed to the type τ_2 or $\{\tau_2\}$ in accordance with the underlying database, regardless of the original type of t. Similarly, the type of the term $\tau : t$ is decided by the type of term t in the underlying database. As for term $t_1.\, t_2$, it refers to a path expression, which will be discussed later in details. In addition, the terms $t_1 \vee t_2$, $\{t_1, ..., t_n\}$ and $[t_1, ..., t_n]$ represent union, set and tuple constructors as usual.

Literals: The *literals* consist of positive literals and negative literals. Let t_1 and t_2 be terms, then

- the positive literals are $t_1(t_2)$, $t_1 = t_2$, $t_1 \in t_2$ and $t_1.t_2$;
- the negative literals are $\neg t_1(t_2)$, $t_1 \neq t_2$, $t_1 \notin t_2$ and $\neg t_1.t_2$.

The *atoms* are positive literals. For the ease of expression, three atoms are presented in particular.

1. relation atoms: $R(t_1, ..., t_k)$;
2. value atoms: $\widehat{i}(t_1, ..., t_k)$;
3. class atoms: $c(i, [t_1, ..., t_k])$ or $c(i)$.

Rules: A *rule* r is an expression of the form

$$L_0 \leftarrow L_1, ..., L_n$$

where $n \geq 0$, L_0 is an atom and $L_1, ..., L_n$ are literals. For convenience, the literal on the left-hand side is denoted by $head(r)$; whereas the literals on the right-hand side are denoted by $body(r)$.

In terms of atoms placed in $head(r)$, there are three kinds of rules are defined in the language.

-Relation rules:

$R(t_1, ..., t_k) \leftarrow L_1, ..., L_n$, where $R(t_1, ..., t_k)$ is a relation atom, and $t_1, ..., t_k$ are terms of variables or constants. Further, variables can occur only in head(r), only in $body(r)$, or both in $head(r)$ and $body(r)$;

-Value assignment rules:

$\widehat{i}(t_1, ..., t_k) \leftarrow L_1, ..., L_n$, where $\widehat{i}(t_1, ..., t_k)$ is a value atom, and $t_1, ..., t_k$ are terms of variables, constants or $\tau \hookrightarrow t$. Further, all variables must occur both in $head(r)$ and in $body(r)$;

-Class rules:

$c(i, [t_1, ..., t_k]) \leftarrow L_1, ..., L_n$, where $c(i, [t_1, ..., t_k])$ is a class atom, and $t_1, ..., t_k$ are terms of $\tau : u$, $\tau : x$ or $\tau \hookrightarrow t$. Further, variable i only occurs in head(r) while other variables occur both in $head(r)$ and in $body(r)$.

All variables occurring in $head(r)$ that do not appear in $body(r)$ represent identifiers. In addition, to guarantee safety, each rule must be range-restricted (the notion of range-restricted will be defined later in Section 3.2.5.).

Programs: A *program* Γ consists of a nonempty finite set of rules such that $r_1.r_2.....r_n (n \geq 1)$, where $r_i (i \in [1, n])$ is a rule.

Queries: A *query* Q is a sequence of programs $\Gamma_1; ...; \Gamma_n (n \geq 1)$, where $\Gamma_i (i \in [1, n])$ is a program.

Remark 3.1.1. Compared to the syntax of IQL, a logic programming approach proposed in [7], we adopt a similar style for defining the language XIQL. However, some important features distinguish XIQL from IQL by providing more functionality. Syntactically, these are as follows. First, sets are treated as "first-class citizens" since we do not use distinguished symbols for set and non-set variables in the formalism. Secondly, type terms are developed to enable a capability for querying over types. Furthermore, object-structured values, denoted $\widehat{\widehat{i}}$, are included into terms as well to enrich the expressive power of the language. Finally, an additional kind of rule with a class literal in $head(r)$ allows that object identifiers can be invented with explicit types in the language.

3.2 Semantics

The goal of this section is to explore the semantics of XIQL from multiple aspects. Besides some traditional issues covered in the semantics of database manipulation languages, such as valuation, satisfaction, semantics of queries, fixpoint semantics and safe queries, some special features embodied on XML data, including optional semantics and underlying type binding, will be additionally discussed in Section 3.2.3.

3.2.1 Valuations and Satisfaction

Since the core of XIQL comes from IQL, in which objects are built in stages, incomplete objects might also exist in the intermediate procedure of XIQL queries. Thus, a valuation θ is a partial function defined on a database instance I mapping individual variables to appropriate values from the underlying domain $xdom(I)$ of database instance I, a set $\mathbb{C}$ of class names, or a set O_{new} of new object identifiers, which are created during querying. For these new object identifiers, values are assigned to them in a weak manner [7].

The usual rules for XIQL valuation are as follows.

- For type variables $\&x$, $\theta\&x = c$, where $c \in \Sigma_c$;

- For variables x, x' that occur in a relation rule r: $R(t_1, ..., t_k) \leftarrow L_1, ..., L_n$,

  - if x occurs in $body(r)$, then $\theta x \in xdom(I)$;
  - if x only occurs in $head(r)$, then
    - (1) $\theta x \in xdom(I)$ iff $[\theta x_1, ..., \theta x_m] \preceq R(t_1, ..., t_k)$ holds, where variables $x_1, ..., x_m$ of types $\tau_1, ..., \tau_m$, respectively, are both in $head(r)$ and in $body(r)$;
    - (2) otherwise, $\theta x \notin xdom(I)$ and $\theta x \in O_{new}$.
  - if both x and x' occur only in $head(r)$, then $\theta x \neq \theta x'$;
  - if x occurs in $head(r)$ and x' occurs in $head(r')$ and $r \neq r'$, then $\theta x \neq \theta x'$.

- A variable x occurs in a value assignment rule r, then $\theta x \in xdom(I)$;

- For a variable x that occurs in a class rule r: $c(i, [t_1, ..., t_k]) \leftarrow L_1, ..., L_n$,

  - if x occurs in a place of identifier in $head(r)$, then
    - (1) $\theta x \in xdom(I)$ iff $[\theta t_1, ..., \theta t_n] = \pi(o)$ holds, where o is some object such that $o \in xdom(I)$, and π is a value project function for objects defined in Section 2.2;
    - (2) otherwise, $\theta x \notin xdom(I)$ and $\theta x \in O_{new}$.
  - otherwise, $\theta x \in xdom(I)$;

- $\theta(t_1 \vee t_2) = \theta t_1 \cup \theta t_2$;

- $\theta(t_1 \,.\, t_2) = \theta t_1 \,.\, \theta t_2$;

- $\theta\{t_1, ..., t_n\} = \{\theta t_1, ..., \theta t_n\} (n \geq 0)$;
- $\theta[t_1, ..., t_n] = [\theta t_1, ..., \theta t_n] (n \geq 0)$.

Now we define the notion of satisfaction on database instances. Let I be a database instance, S be a database schema, θ be a valuation on I and F be a formula. Then $(I, \theta) \vDash_S F$ is written to express that I satisfies the formula F over S with the valuation θ provided that one of the following holds:

- $(I, \theta) \vDash_S R(t_1, ..., t_k)$ iff $[\theta t_1, ..., \theta t_k] \in R$, where R is a relation in I;
- $(I, \theta) \vDash_S c(i, [t_1, ..., t_k])$ iff $[\theta t_1, ..., \theta t_k] \preceq \pi(o)$, $\theta i = \delta(o)$ and $\sigma(o) = c$, where π, δ and σ are value, identifier and name projection functions, respectively, defined in Section 2.2, o is an object in I and $\preceq$ indicates a subsumption relation defined in Definition 2.2.4;
- $(I, \theta) \vDash_S c(i)$ iff $\theta i = \delta(o)$ and $\sigma(o) = c$, where σ and δ are name and identifier projection functions, respectively, defined in Section 2.2, and o is an object in I;
- $(I, \theta) \vDash_S \widehat{i}(t_1, ..., t_k)$ iff $[\theta t_1, ..., \theta t_k] = \pi(o)$, and $\theta i = \delta(o)$, where π and δ are value and identifier projection functions, respectively, defined in Section 2.2, and o is an object in I;
- $(I, \theta) \vDash_S f(t_1, ..., t_n)$ iff $f(\theta t_1, ..., \theta t_n)$ of type BOOL is true.
- $(I, \theta) \vDash_S t_1 \in t_2$ iff $\theta t_1 \in \theta t_2$, and $(I, \theta) \vDash t_1 \notin t_2$, iff $\theta t_1 \notin \theta t_2$;
- $(I, \theta) \vDash_S t_1 \; op \; t_2$, where $op \in \{<, \leq, >, \geq, =, \neq\}$, iff $\theta t_1 \; op \; \theta t_2$;
- $(I, \theta) \vDash_S t_1 \; . \; t_2$ iff $\theta t_2 \measuredangle \theta t_1$;
- $(I, \theta) \vDash_S \neg \, t_2(t_1)$ iff $(I, \theta) \nvDash t_2(t_1)$.

In fact, $c(i)$ is just a variant expression of $c(i, [t_1, ..., t_k])$ when $[t_1, ..., t_k]$ is not a concern and can thus be always satisfied in any case.

Remark 3.2.1. In XIQL, the valuation and satisfaction are slightly complicated by object creation mechanism implicated in three kinds of rules. However, we believe that it is worthwhile since a powerful capability for object creation can be easily provided by using these rules in a natural fashion.

3.2.2 Semantics of Queries

On the basis of the SXO model, we begin with exploring the relationship between instances and schemata of XIQL.

Given an XML database D, alternatively, instances and schemata discussed in Chapter 2 can also be expressed as follows.

- A *schema* S over D consists of a finite set C of class names.

- An *instance* I over schema S is a finite set O of objects that is a triple (σ, δ, π), where σ, δ and π are name, identifier and value projection functions for O, respectively. Further, for any object $o \in O$, there must exist some $c \in C$ such that $\tau(o) \sqsubseteq s_c$ ($\sqsubseteq$ is defined in Section 2.2.2).

Obviously, we can observe that a schema can be treated as a special part of instance since class names have been encoded into object expressions. In fact, different from IQL, which is built on a clear separation of the notions of instance and schema so that a number of typing restrictions are imposed on IQL via type checking, XIQL provides a powerful and flexible type creation and query mechanism to make sure that types are incorporated into the language without strong limitations.

Since a schema and an instance can be considered as finite sets of classes and objects respectively, we need to introduce several concepts here in order to further formalize the semantics of queries.

- Given two schemata S_1 and S_2, $S_1 \subseteq S_2$ if and only if, for all $c_1 \in S_1$, there exists some $c_2 \in S_2$.

- Given an instance I with its schema S, and a schema S_1 such that $S_1 \subseteq S$, then $I[S_1]$ is defined to be an instance with schema S_1 after projecting I on S_1 such that, for objects in I, only the objects within classes in S_1 can exist in $I[S_1]$.

Following the same line of IQL [7], the semantics of a program Γ in XIQL could be considered as a binary relation $\gamma \subseteq \mathit{instance}(S_{in}) \times \mathit{instance}(S_{out})$, where S_{in} and S_{out} are input and output schemata defined as follows in our work.

Definition 3.2.1. (Extensional and Intensional Classes) Given a program Γ, classes only occurring in the body of the rules represent extensional classes, denoted C_{edb}, whereas classes occurring in the head of some rule represent intensional classes, denoted C_{idb}. Then, $S_{in} = \bigcup C_{edb}$ and $S_{out} = \bigcup C_{idb} \cup \bigcup C_{edb}$.

Correspondingly, if we denote $instance(S_{in})$ and $instance(S_{out})$ by a pair of (I_1, I_2), then $I_1 = I[S_{in}]$ and $I_2 = J[S_{out}]$, where I and J are database instances before and after implementing a program, respectively, $I[S_{in}]$ represents projecting I on S_{in} and $J[S_{out}]$ represents projecting J on S_{out} as defined above.

Since a query is a sequence of programs, thus, intuitively, the semantics of a query can be considered as a sequence of binary relations such that $\gamma_1; ...; \gamma_n$, where query Q consists of n programs that represent binary relations $\gamma_1; ...; \gamma_n$ respectively. In fact, $\gamma_1; ...; \gamma_n$ can be regarded as a binary relation as well since the output instance and schema of γ_k are always the input instance and schema of γ_{k+1}. Therefore, the semantics of queries can also be considered as a binary relation in our work.

One thing needs to be paid extra attention here. XIQL queries are closed under the SXO model so that both input and output schemata with respect to a particular query stem from the SXO model. This is the reason why relations are considered in the syntax of XIQL queries, but ignored in the semantics of XIQL queries since only objects exist in input and output instances. More precisely, relations are only regarded as an auxiliary expression to facilitate the language based on the following reasons.

- Values of a set of objects can be treated as relations when their object identifiers are not concerned;
- The semantics of relations provides a capability to eliminate some duplicates as stated in [7, 65];
- The language can be simplified with respect to the operational computation.

3.2.3 Union and Intersection Types

In this section, we introduce union and intersection types, and show how they are related to some features of the language. In XIQL, the main results regarding union and interaction types are these:

1. we observe that union types provide an efficient and natural approach to express optional semantics since optional semantics as an important feature encoded in XML data has to be dealt with in XIQL;

2. the underlying binding of types within objects can be interpreted as the semantics of interaction types through an appropriate equivalence.

We begin with union types. Recall that the union of types is developed in the language as usual, let τ_1, τ_2 be two types, then the union of types τ_1 and τ_2 (notation: $\tau_1 \vee \tau_2$) is interpreted as $\| \tau_1 \vee \tau_2 \| = \| \tau_1 \| \cup \| \tau_2 \|$.

Based on the above interpretation, the following equivalent expressions hold.

$$[\tau_1 : x_1 \vee \tau_2 : x_2, \tau_3 : x_3] = [\tau_1 : x_1, \tau_3 : x_3] \cup [\tau_2 : x_2, \tau_3 : x_3]. \quad (1)$$
$$[\tau_1 \vee \tau_2 : x, \tau_3 : x_3] = [\tau_1 : x, \tau_3 : x_3] \cup [\tau_2 : x, \tau_3 : x_3]. \quad (2)$$

The difference between expressions (1) and (2) consists in the fact that one variable x is bound to the union of types τ_1 and τ_2 in expression (2), whereas two different variables x_1 and x_2 are bound to types τ_1 and τ_2 individually in expression (1). However, to handle the optional semantics with union types, a trivial type needs to be further defined.

Definition 3.2.2. $\oslash$ is a trivial type that means no specified type, and it can be interpreted as $\| \oslash \| = \lambda$, where λ means that no specified value exists.

The following example is provided for the purpose of showing how union types can facilitate the optional semantics in XIQL.

Example 3.2.1. Consider the XML document book.xml, and a query that will list all sections with their titles, together with the id attribute if any.

$$R(A, i, t) \leftarrow section(s, [@id \vee \oslash : i, title : t]). \quad (1)$$
$$\widehat{A}(i, t) \leftarrow R(A, i, t); \quad (2)$$
$$result(r, [section \hookrightarrow A]) \leftarrow R(A, i, t). \quad (3)$$

The rule (i) can be rewritten as $R(A, i, t) \leftarrow section(s, [@id : i, title : t]) \cup section(s, [\oslash : i, title : t])$. Thus, it is quite easy to see that i can be valuated either as an identifer of the object typed $@id$ or as λ, since variable i is bound to the union of types $@id$ and $\oslash$. The results of this query are shown in Figure 3.1.

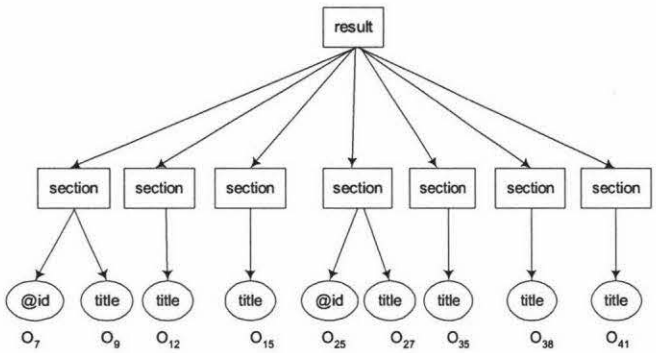


Figure 3.1: The query results of Example 3.2.1

Now, we turn to discuss interaction types. Recall that, in Section 3.2.1, we define that $(I, \theta) \vDash c(i, [t_1, ..., t_k])$ if and only if $[\theta t_1, ..., \theta t_k] \preceq \pi(o)$, and $\sigma(o) = c$. Indeed, this expression implies the following equivalence regarding the type interpretation of $[\tau_1, ..., \tau_n]$, which is the type of $[t_1, ..., t_n]$ as defined in terms of the language.

$$\| [\tau_1, ..., \tau_n] \| = \{[\tau_1 : v_1, ..., \tau_n : v_n, \tau_{n+1} : v_{n+1}, ..., \tau_m : v_m] | \text{ for some } \tau_{n+1}, ..., \tau_m (m \geq n), \text{ which are distinct from } \tau_1, ..., \tau_n\}.$$

Departing from some similar equivalences appearing in the literature [16, 7], which focus on the semantics of type inheritance, this equivalence is used for a different purpose in our work to capture the underlying type binding of object literals. On this basis, some interesting equivalent expressions with respect to the intersection of types can be obtained as follows.

$$[\tau_1 : x_1, \tau_2 : x_2] \wedge [\tau_1 : x_1, \tau_3 : x_3] = [\tau_1 : x_1, \tau_2 : x_2, \tau_3 : x_3]. \quad (1)$$

$$[\tau_1 : x_1 \wedge \tau_2 : x_2, \tau_3 : x_3] = [\tau_1 : x_1, \tau_2 : x_2, \tau_3 : x_3]. \quad (2)$$

Finally, let us look at a simple example to show that the intersection of types is implicitly present in the binding of object types.

Example 3.2.2. Consider again the XML document book.xml. We define a query to list the titles of all sections that have figures. Two solutions are provided as follows, and they are equivalent.

$$result(i, [title : t]) \leftarrow section(s, [title : t]), section(s, [figure : f]). \quad (1)$$

$$result(i, [title : t]) \leftarrow section(s, [title : t, figure : f]). \quad (2)$$

3.2.4 Inflationary Fixpoint Operator

In this section, we shall investigate the semantics of XIQL queries from a computational perspective using a inflationary fixpoint operator, essentially in the spirit of IQL [7].

First of all, the notion of an immediate consequence operator is employed in XIQL by building upon the semantics of programs discussed in Section 3.2.2.

Definition 3.2.3. (Immediate Consequence Operator) Let Γ be a program and I_0 be an input database instance, an immediate consequence operator μ_Γ on Γ over I_0 is a binary relation γ^1=$instance_{in} \times instance_{out}$ by executing all rules of Γ once. Further, the following condition holds.

- $instance_{out} = instance_{in} \cup \triangle O$, where $\triangle O$ is a finite set of objects created during the executing all rules of Γ once.

Given a program Γ and an input database instance I_0, thus the inflationary fixpoint operator of Γ over I_0 is also a binary relation $\gamma^* = (J_1, J_n)$ on instances such that a finite set instances J_2,...,J_{n-1} can be computed in sequence satisfying the conditions

(1) $\gamma^i = \{(J_1, J_{i+1}) | \exists_{J_i}(J_1, J_i) \in \gamma^{i-1} \wedge (J_i, J_{i+1}) \in \gamma^1\}$, where $i \in [2, n-1]$ and $J_1 \subseteq J_2 \subseteq$ ..... $\subseteq J_n$ holds;

(2) $\gamma^* = \gamma^{n-1}$ if and only if $\gamma^{n-1} = \gamma^n$.

It means that a fixpoint J_n of program Γ is reached in the above sequence where $\gamma^* = \gamma^{n-1}$ if no more change occurred on instances after $n-1$ steps of the immediate consequence operator. Furthermore, the fixpoint semantics defined above clearly shows an inflationary feature that objects can be created into a database, but can not be deleted or updated within a database.

For a query with a sequence of programs, the programs are evaluated sequentially to compute inflationary fixpoints.

3.2.5 Safe Queries

In this section safety issues of XIQL queries will be discussed, especially for queries involving object creation and negation. Since the underlying universe might be infinite, answers to queries can not be guaranteed to be finite in general, some restrictions need to be imposed on XIQL.

Our idea proceeds in two steps. As a first step, we start with a definition of a range-restricted query to get desired queries in finite underlying domain interpretation. As a second step, we focus on nonterminating computations caused by recursion with object creation [7, 2] by means of examining the possible recursion occurring in a program.

From the finiteness point of view, literals in a query can be categorized into three kinds.

- *Safe literals* that mean variables inside have finite domain, such as: $t_1(t_2)$, $t_1.t_2$;
- *Unsafe literals* that mean variables inside have infinite domain, such as: $\neg t_1(t_2)$ and $\neg t_1.t_2$;
- *Fuzzy literals* that mean variables inside may have finite domain, and also may have infinite domain, such as: $t_1 = t_2$, $t_1 \in t_2$, $t_1 \neq t_2$, $t_1 \notin t_2$.

The concept of range-restricted similar to that in [7] is developed as follows.

Definition 3.2.4. Let Q be a query, r be a rule and v be a variable, then

(i) A variable v is *range-restricted* if and only if one of the following conditions is satisfied.

- a variable v occurs in at least one safe literal in $body(r)$, or
- a variable v occurs in a class literal of $body(r)$, in which the variable in place of identifier is range-restricted.

(ii) A rule r is *range-restricted* if and only if all of the following conditions are satisfied.

- all variables occurring in $body(r)$ are range-restricted, and
- only safe literals occur in $head(r)$.

(iii) A query Q is *range-restricted* if and only if all rules in Q are range-restricted.

A simple example involving negation is provided here to demonstrate a range-restricted query.

Example 3.2.3. Consider the XML document book.xml, for which the following query lists the sections that contain figures but have no attribute element.

$$\begin{aligned} R(i_s) \leftarrow\ & section(i_s, [figure : f]), && (1)\\ & \neg section(i_s, [@id : i_i]),\\ & \neg section(i_s, [@difficult : i_d]);\\ result(i, [section : i_s]) \leftarrow\ & R(i_s). && (2) \end{aligned}$$

Since section objects only have two attributes: @id and @difficult, each object that contains either of them is removed from the final result. The query is range-restricted because it satisfies the conditions presented in the above definition. Alternatively, we can encode union types into the rule (1) in the query.

$$\begin{aligned} R(i_s) \leftarrow\ & section(i_s, [figure : f]), && (1)\\ & \neg section(i_s, [@id : i_i \vee @difficult : i_d]); \end{aligned}$$

Although range-restricted queries are defined, they might produce a potentially non-terminating loop due to a recursive invention of object identifiers. To avoid this problem, a concept of "recursion-freedom" is defined to control invention in [6, 7]. Alternatively, to handle this problem, objects can only created as a function of input objects and not of new output created objects as proposed in [2]. In our opinion, the first approach seems more interesting than the second approach since the second one might lead to some restriction on the expressive power of languages.

In Abiteboul and Kanellakis's work [6, 7], a recursion-free program Γ is defined under a strict technical restriction that the leftmost symbol of each rule in Γ must be a relation name to simplify the presentation. Along the lines of recursion-free programs, XIQL provides a simpler and more general approach to identify recursion that might exist in a program.

Definition 3.2.5. Let Γ be a program and let Σ_r be a set of rules involving object creation that include all relation rules and class rules in Γ. Γ is recursion-free if there is no cycle in the object-creation graph. An object-creation graph is a directed graph defined as follows.

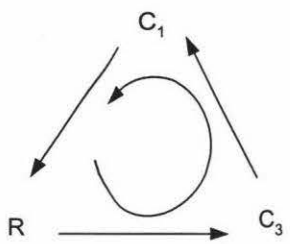


Figure 3.2: The object-creation graph for Example 3.2.4

For each rule r in Σ_r, if a class or a relation in $body(r)$ has at least one variable occurring in $head(r)$, then

- add two nodes n_1 and n_2, which denote relation or class names in $body(r)$ and $head(r)$ respectively, if they still do not exist in the graph, and
- add an edge from nodes n_1 to n_2.

Example 3.2.4. Let us look at an example to illustrate how to get an object-creation graph.

$$R(w, x, y) \leftarrow C_1(i, [A_1 : x, A_2 : y, A_3 : z]). \quad (1)$$
$$\widehat{w}(x, y) \leftarrow R(w, x, y); \quad (2)$$
$$C_3(i, [A_4 : w]) \leftarrow R(w, x, y), C_2(e, [A_1 : x, A_5 : k]); \quad (3)$$
$$C_1(i, [A_1 : o, A_2 : w, A_3 : z]) \leftarrow C_3(o, [A_4, w]). \quad (4)$$

In this example, $\Sigma_r = \{r_1, r_3, r_4\}$ since only rules (1), (3) and (4) involve object creation. The associated object-creation graph is shown in Figure 3.2. Clearly, there is a cycle in the graph, so the program is not recursion-free.

Compared with the approach in [6, 7] our improvement consists of two things. The first one is that not only relations but also objects are considered in $head(r)$. This technical restriction imposed on defining IQL recursion-freedom is released in XIQL. Second, the object-creation graph can be obtained in a simpler manner.

Definition 3.2.6. (Safe Queries) A query Q is safe if and only if Q is range-restricted and recursion-free.

a_1	a_2	a_3
b_1	b_2	b_3
c_1	c_2	c_3
d_1	d_2	d_3

o_1	a_1	a_2	a_3
o_2	b_1	b_2	b_3
o_3	c_1	c_2	c_3
o_4	d_1	d_2	d_3

Table 3.1: Tuple-new statement with ϕ(left) and R(right)

Remark 3.2.2. Observing from the above discussion, two main possibilities to cause unsafe XIQL queries are negation and recursion of object creation. Hence, the concepts of range-restricted and recursion-free are developed for this purpose. It is sufficient to obtain safe queries if they are both range-restricted and recursion-free.

3.3 Object Creation Mechanism

Object creation is a crucial mechanism that is required for languages in object-based models. Intensive research efforts [7, 32, 65, 68, 10] on object-creating query languages have been done to handle this feature.

Although object creation approaches are greatly different in syntax and semantics over languages, most of them adopt a style of oid invention similar to skolem function techniques [50, 42, 43, 40] in essence. Furthermore, they are considered to be expanded in two directions: tuple objects and set objects. Here, we borrow the notions of tuple-new statement and set-new statement from [69] since we believe that they generalize most approaches adopted in object-creating query languages.

A *tuple-new statement* can be expressed as $R :=$ **tuple-new** ϕ, where R is a $k+1$-ary relation name and ϕ is a k-ary first-order expression. Let I be a database instance and $\phi(I) = \{t_1, ..., t_n\}$, then $R = (\{t_1\} \times \{o_1\}) \cup ... \cup (\{t_n\} \times \{o_n\})$, where $o_1, ..., o_n$ are n different objects not in $|I|$.

A *set-new statement* can be expressed as $R :=$ **set-new** ϕ, where R a binary relation name and ϕ is a k-ary first-order expression. Let I be an database instance and $\phi(I) = \{(x_1, t_1), ..., (x_n, t_n)\}$, then $R = \{(x_1, o_1), ..., (x_n, o_n)\}$, where $o_1, ..., o_n$ are objects not in $|I|$ satisfying $o_i = o_j$ if and only if $\{y|(x_i, y) \in \phi(I)\} = \{y|(x_j, y) \in \phi(I)\}$.

a_1	b_1	b_2
a_2	b_1	b_2
c_1	d_1	d_2
c_2	d_1	d_2
e_1	e_2	e_2

o_1	a_1
o_1	a_2
o_2	c_1
o_2	c_2
o_3	e_1

Table 3.2: Set-new statement with ϕ(left) and R(right)

a_1	a_2	a_3
b_1	b_2	b_3
c_1	c_2	c_3
d_1	d_2	d_3

a_1	a_2	a_3
b_1	b_2	b_3
c_1	c_2	c_3
d_1	d_2	d_3
o_1	o_2	o_2

Table 3.3: SingletonSet-new statement with ϕ(left) and R(right)

It has been clearly shown that, in Figure 3.1, one object is created for each row in a relation by using **tuple-new** statement, and in Figure 3.2, one object is created for part of a column after grouping in a relation. Since new objects in a class are always placed in a column of a relation encoded with some common properties, it might be of interest to create a new singleton object representing a set of objects in a desired column as shown in Figure 3.3.

It has been simulated in [69] that a singleton object can be created from scratch by means of zero-ary first-order expressions. The following program yields a unary relation that contains one new object identifier.

$$R :=\textbf{tuple-new}\ \{()|\ \textbf{true}\ \}$$

Let us turn to examine how a set of singleton objects can be created in XIQL. In terms of Figure 3.3, three new singleton objects A, B and C, which have identifiers o_1, o_2 and o_3, respectively, can be created in the following manner.

$$\begin{aligned} R_1(A,B,C) &\leftarrow . \\ \widehat{A}(x) &\leftarrow \phi(x,y,z), R_1(A,B,C). \\ \widehat{B}(y) &\leftarrow \phi(x,y,z), R_1(A,B,C). \end{aligned}$$

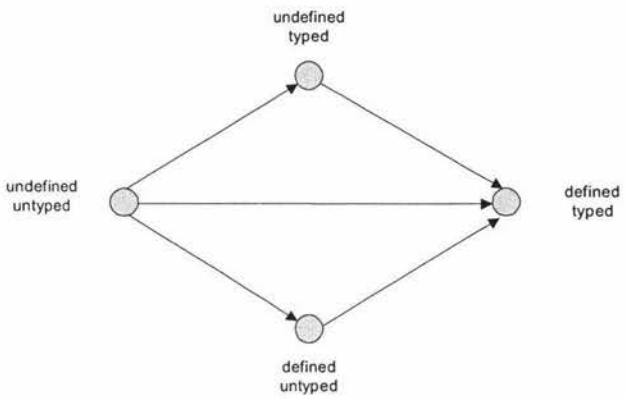


Figure 3.3: The statuses of object

$$\widehat{C}(z) \leftarrow \phi(x, y, z), R_1(A, B, C).$$

After executing this program, $\widehat{A} = \{a_1, b_1, c_1, d_1\}$, $\widehat{B} = \{a_2, b_2, c_2, d_2\}$ and $\widehat{C} = \{a_3, b_3, c_3, d_3\}$.

For clarity, let us see one concrete example built on the running database as follows.

Example 3.3.1. Consider the XML document in book.xml, and a query that produces a single object *result* containing sections that have two attributes $@id$ and $@difficulty$, along with the authors.

$$\begin{aligned} R(S, A) &\leftarrow . \\ \widehat{S}(i_s) &\leftarrow section(i_s, [@id : i, @difficulty : d]), R(S, A). \\ \widehat{A}(i_a) &\leftarrow author(i_a), R(S, A); \\ result(i, [section : \widehat{S}, author : \widehat{A}]) &\leftarrow R(S, A). \end{aligned}$$

In this example, we can get: $\widehat{S} = \{i_6, i_{24}\}$ and $\widehat{A} = \{i_3, i_4, i_5\}$, so an object in the class *result* with value $[section : \{i_6, i_{24}\}, author : \{i_3, i_4, i_5\}]$ is created.

In XIQL, a pure object identifier could be invented without specified object type and object value. However, at the intermediate stages, each pure object identifier must be defined with value assignment rule of a form "$\widehat{i}(t_1, ..., t_k) \leftarrow L_1, ...L_n$", or typed with typing operator "$\hookrightarrow$" when necessary so that it represents a complete object in the final result. The four statuses that might occur on an object during an XIQL program are shown in Figure 3.3.

The example to illustrate object statuses is presented for illustration.

Example 3.3.2. List books published by Addison_Wesley after 1991 in the XML document bib.xml, including their year and title.

$$R(A, i_b, y, t) \leftarrow book(i_b, [publisher : p, @year : y, title : t]), \widehat{y} > 1991, \quad (1)$$
$$\widehat{p} = \text{“}Addison\_Wesley\text{”}.$$
$$\widehat{A}(y, t) \leftarrow R(A, i_b, y, t); \quad (2)$$
$$result(i, [book \hookrightarrow A]) \leftarrow R(A, i_b, y, t). \quad (3)$$

For each book published by Addison_Wesley after 1991, there is a new object identifier invented in A. The invented objects are both untyped and undefined in rule (1). Then, they are defined by assigning year and title as values in rule (2). In the end, objects created in A are typed to *book*, and a singleton object is created with *book* as value in the final result.

Comparing **tuple-new** and **set-new** statements with three kinds of rules developed in XIQL (see Section 3.1), it is quite straightforward to find out the following results:

- **tuple-new** statements can be expressed in XIQL with relation and class rules;
- **set-new** statements can be expressed in XIQL with relation rules and value assignment rules;
- class and value assignment rules can explicitly specify types of objects.

3.4 Key Characteristics and Examples

In this section, some examples are provided to illustrate other key characteristics of XIQL as an XML query language. To keep clear expressions, each subsection only focuses on a specific language feature that is illustrated with one or more examples based on the running example.

3.4.1 Nest and Unnest

Nest and unnest are crucial operations within most complex value data models, especially, in the context of XML data. Although sets are regarded as “first class citizens” in XIQL, nest and unnest operations can also be easily realized by using a set operator $\in$, which is

associated with a first order variable and a second order variable. In general, nesting is more complicated than unnesting so that it needs to be simulated with invented oids, as indicated in [6, 7]. The details can be illustrated by the following example.

Example 3.4.1. For each author of books in the XML document bib.xml, list the author's name and the titles of all books by that author, grouped inside a "result" element.

$$
\begin{aligned}
R_1(A,B,C,\widehat{v}) &\leftarrow book(i_b,[author:u]), && (1)\\
&\quad\ v \in u.\\
\widehat{B}(\widehat{v}) &\leftarrow R_1(A,B,C,\widehat{v}). && (2)\\
\widehat{C}(t) &\leftarrow R_1(A,B,C,\widehat{v}), && (3)\\
&\quad\ book(i_b,[author:u,title:t]),\\
&\quad\ v \in u.\\
\widehat{A}(B,\widehat{C}) &\leftarrow R_1(A,B,C,\widehat{v}); && (4)\\
R_2(D) &\leftarrow . && (5)\\
\widehat{D}(A) &\leftarrow R_1(A,B,C,\widehat{v}), R_2(D); && (6)\\
Ans(i,[book \hookrightarrow \widehat{D}]) &\leftarrow R_2(D). && (7)
\end{aligned}
$$

In rule (1), all the authors are unnested into the relation R_1 with the use of the set operator $\in$. Then, three new object identifiers are created for each author in relation R_1, and duplicate authors that have the same name can be eliminated in this case. In the sequel, rule (2) assigns author names as values to new identifiers in B, Rule (3) nests the titles of all books with respect to each distinct author into the value of new identifiers in C. To get the desired query result, B and $\widehat{C}$ are assigned as values to object identifiers in A in rule (4). Finally, a singleton object is created, and assigned a set of object identifiers in A as its value in rule (5) and (6). Rule (7) creates a *result* object and type A to *book*.

One thing we should be aware of in this example, is that author objects are duplicates in the type of depth one so that duplicates can be eliminated in the above program. However, when author objects have duplicates of more than one depth type, the situation will become much more complicated to eliminate duplicates. We shall go into the details regarding this issue in Chapter 4.

3.4.2 Structural Recursion

XML data might involve cyclic structures at schema level, as shown in Figure 2.3. For an XML query language, there are at least two possible scenarios that need restructuring of hierarchic structures.

- Input database schema is acyclic, whereas output database schema is encoded with cyclicity. For instance, a query that converts the "flat" representation into a hierarchic representation
- Both input and output database schemata contain the encoding of cyclicity. For instance, a query that preserves original hierarchy.

In fact, XIQL provides a flexible and concise approach to handle structural recursion, even in the case that objects have wide variation in structure. We provide one example to describe how XIQL works in the second scenario.

Example 3.4.2. Let us see the XML document book.xml. List all the sections and their titles. Preserve the original hierarchy if possible.

$$
\begin{aligned}
R_1(X, Y, i_s) &\leftarrow \mathit{section}(i_s). && (1)\\
\widehat{Y}(Z) &\leftarrow R_1(X, Y, i_1), && (2)\\
&\quad\ R_1(Z, W, i_2),\\
&\quad\ \mathit{section}(i_1, [\mathit{section} : s]),\\
&\quad\ i_2 \in s.\\
\widehat{X}(t, \mathit{section} \hookrightarrow \widehat{Y}) &\leftarrow R_1(X, Y, i_s), && (3)\\
&\quad\ \mathit{section}(i_s, [\mathit{title} : t, \mathit{section} : s]);\\
R_2(D) &\leftarrow . && (4)\\
\widehat{D}(X) &\leftarrow R_1(X, Y, i_s), && (5)\\
&\quad\ \mathit{book}(i_b, [\mathit{section} : s]),\\
&\quad\ i_s \in s, R_2(D);\\
\mathit{result}(i, [\mathit{section} \hookrightarrow \widehat{D}]) &\leftarrow R_2(D). && (6)
\end{aligned}
$$

In rule (1), a pair of object identifiers in X and Y are created for each section. X is to represent a new section object in the final query result; whereas Y is to represent a set of new section objects corresponding to each new section object in X. More intuitively, the

purpose of Y is to capture the original hierarchy among sections. Then, Y is defined to have a set of object identifiers of new section objects as appropriate by rule (2). Furthermore, Y is typed to section, and then X is defined to contain title and a set of sections in rule (3). Finally, X is typed to *section*, and the top-level sections are rooted at a singleton object *result*.

3.4.3 Path Expressions

Path expressions have been accepted as an intuitive and concise approach to traverse complicated data structures either in object query processing or in semi-structured query processing. Besides basic syntactical notations, more research efforts are put on semantic implication of path expressions, e.g. The well-known XPath [71] goes through XML data by specifying a set of axes encoding the desired navigation of the tree structure (child, parent, descendant, ancestor, self and etc). The paper [45] distinguishes set relationships with multi-valued path expressions. PathLog [31] extends path expressions to two dimensions: composition of scalar methods and set-valued methods, and the associated object properties so that more semantics could be imposed on path expressions.

However, in our work, the basic idea is to simplify the syntax of path expressions to the most degree so that an unified path expression can be obtained on the basis of dominant relations between objects discussed in Section 2.4.1.

Definition 3.4.1. Let $O = \{O_1, O_2, ..., O_{n-1}, O_n\}(n \geq 1)$ be a set of objects, then the expression $O_1.O_2...O_{n-1}.O_n$ represents that O_{i+1} is in the object family of $O_i(1 \leq i \leq n)$ such that $O_n \measuredangle O_{n-1}$ ... $O_2 \measuredangle O_1$ holds.

Example 3.4.3. Look at the XML document book.xml again, list all titles of figures used in some section of the book "Data on the web".

$$
\begin{aligned}
R_1(t_2) &\leftarrow book(b, [title : t_1]).section(s).figure(f, [title : t_2]), \\
&\quad \widehat{t_1} = \text{“}Data\ on\ the\ web\text{”}; \\
R_2(D) &\leftarrow . \\
\widehat{D}(t_2) &\leftarrow R_1(t_2), R_2(D); \\
result(i, [title \hookrightarrow \widehat{D}]) &\leftarrow R_2(D).
\end{aligned}
$$

To get the final result, the dominant relations: $o_1 \measuredangle o_2 \measuredangle o_3$, where $o_1 \in \theta(figure(f, [title : t_2]))$, $o_2 \in \theta(section(s))$ and $o_3 \in \theta(book(b, [title : t_1]))$, must hold over the database. In terms of the running database, two objects in the class *result* with values $[title : i_{20}]$ and $[title : i_{30}]$, respectively, are created in the results.

3.4.4 Schema Query

As one of the most different features from the relational data model, schema and content data are mixed together within an XML database. The capability of querying schema information is considered to be a necessary requirement for any XML query language.

XIQL realizes schema query by means of a function called *match*, which can match a particular variable with regular expressions. One point should be noted that variables beginning with a symbol "&" are used to distinguish from constants syntactically.

Regular expressions are supported in XIQL to describe desired patterns. Limited to space, it is not necessary to give a full discussion of regular expressions used in XIQL here. We only introduce the expression "%" that matches zero or more occurrences of the character immediately preceding.

Example 3.4.4. Consider the XML document book.xml, find books in which the name of some element ends with the string "or". For such book, return the title and that element.

$$\begin{aligned} R_1(Y, t, i_x) &\leftarrow book(i_b, [title : t]).\&X(i_x), match(\&X, \text{“\%or”}). \\ \widehat{Y}(t, i_x) &\leftarrow R_1(Y, t, i_x); \\ result(i, [book \hookrightarrow Y]) &\leftarrow R_1(Y, t, i_x). \end{aligned}$$

$\&X$ is a variable to represent the class name of some object, which ends with the string "or". Function *match* is used to guarantee the matching between them. In addition, this program describes some object in class $\&X$ is a descendant of *book* object by using the path expression $book(i_b, [title : t]).\&X(i_x)$. In the end, Y is defined with title and that element as its value and typed to *book*.

3.4.5 Join Operation

Join is an important operation in the relational data model. Indeed, XIQL can easily simulate the join operation, except that unified variables may lead to slightly different programming between variables in set or non-set semantics. On the basis of the XML document bib.xml in the running example database, two comparable examples are provided for illustration.

Join on Set Variables

First, let us see how to deal with the join operation on set variables in XIQL.

Example 3.4.5. Let us have a look at the XML document bib.xml, and find pairs of books that have different titles but the same set of authors (possibly in a different order).

$$
\begin{aligned}
R(C, t_1, t_2) \leftarrow\ & book(i_1, [title : t_1, author : u_1]), && (1)\\
& book(i_2, [title : t_2, author : u_2]),\\
& \widehat{t_1} \neq \widehat{t_2},\\
& \widehat{u_1} = \widehat{u_2}.\\
\widehat{C}(t_1, t_2) \leftarrow\ & R(C, t_1, t_2); && (2)\\
result(r, [book\_pair \hookrightarrow C]) \leftarrow\ & R(C, t_1, t_2). && (3)
\end{aligned}
$$

Due to no discrimination between set and non-set variables in XIQL expressions, $\widehat{u_1} = \widehat{u_2}$ is equal to set comparison when more than one author is involved in books. $\widehat{u_1} = \widehat{u_2}$ holds if and only if values of all objects in $\widehat{u_1}$ exactly equal values of all objects in $\widehat{u_2}$. Finally, C is defined to contain the titles of each pair of books, and typed to *book_pair*.

Join on Non-set Variables

In the case that the join operation will work on some objects within non-set variables, how will XIQL do? For the sake of contrast, the preceding example has been changed a bit to illustrate the join operation on non-set variables.

Example 3.4.6. Consider the XML document bib.xml again. Find pairs of books that have different titles and at least one same author.

$$
\begin{aligned}
R(C, t_1, t_2) \leftarrow\ & book(i_1, [title : t_1, author : v_1]), && (1)\\
& book(i_2, [title : t_2, author : v_2]),\\
& \widehat{t_1} \neq \widehat{t_2},
\end{aligned}
$$

$$
\begin{aligned}
& u_1 \in v_1, \\
& u_2 \in v_2, \\
& \widehat{u_1} = \widehat{u_2}. \\
\widehat{C}(t_1, t_2) \leftarrow\ & R(C, t_1, t_2); && (2) \\
result(i, [book\_pair \hookrightarrow C]) \leftarrow\ & R(C, t_1, t_2). && (3)
\end{aligned}
$$

Compared with the program of the preceding example, the only difference consists in rule (1), in which " $\in$ " operators are used in the latter example to get some object out of the set variables, and then build the join operation on individual objects as usual. Thus, it has been shown that XIQL provides a very flexible approach to switch between set and non-set variables when non-set variables are of particular interest.

3.4.6 Sequences

From the document point of view, sequence of content is a significant aspect within an structured XML document. As discussed in chapter 2, object identifiers function as order primitives, hence, a powerful function $position(oid_1, |oid_2|)$ can be developed for handling sequence on the object identifer base.

- $position(oid_1, |oid_2|)$ returns an integer number that indicates a order of the object with identifer oid_1, with respect to a specific range. The parameter oid_2 is optional. If oid_2 is provided, then the function returns the relative order of the object with identifer oid_1 with respect to the object with identifer oid_2, otherwise, it returns the global order of the object with identifer oid_1 in the current XML document.

Example 3.4.7. Let us see the XML document book.xml. What are the first two image sources to be used for the book titled "Data on the Web"?

$$
\begin{aligned}
result(r, [@source \hookrightarrow x]) \leftarrow\ & book(i_b, [title : t]).image(i_i, [@source : x]), \\
& \widehat{t} = \text{"}Data\ on\ the\ Web\text{"}, \\
& position(i_i) \geq 2.
\end{aligned}
$$

In this example, the first two images within the book indicate the global order requirement for image objects, hence, $position(i_i) \geq 2$ is used. To be more clear with more complicated cases, let us look at one more example.

Example 3.4.8. Consider the XML document book.xml again. In the book titled "Data on the Web", what happened between the first p and the second p in the fourth section?

$$\begin{aligned} result(r, [sth \hookrightarrow z]) \leftarrow\ & book(i_b, [title : t]).section(i_s, [p : x_1, \&y : z, p : x_2]), \\ & \widehat{t} = \text{"}Data\ on\ the\ Web\text{"}, \\ & position(i_s) = 4, \\ & position(x_1, i_s) = 1, \\ & position(x_2, i_s) = 2, \\ & x_1 < z < x_2. \end{aligned}$$

The p objects are constrained for the relative order with respect to some *section* object, while that *section* object is required with a global order. To ensure that the objects between two p objects can be retrieved, we use their object identifiers for comparing their document orders.

3.4.7 Function Symbols

So far, we have already presented some functions used for the purpose of sequence and schema query. Considering that XIQL should be able to express for aggregation functions as they are expressed in most XML query languages, function symbols are necessary to be developed in XIQL.

Similar to relational query languages, aggregation functions can facilitate retrieving of XML data in some cases, such as: $max()$, $min()$, $count()$ and etc. One example regarding the $count()$ function is provided in the following:

Example 3.4.9. How many top-level sections are in the XML document book.xml?

$$result(r, [topSectionCount \hookrightarrow count(i_s)]) \leftarrow book(i_b, [section : i_s]).$$

Remark 3.4.1. Function symbols in XIQL are not allowed to represent intricate data structures, such as lists and trees. Hence, different from most logic programmings, although function symbols are provided in XIQL, queries always have finite models.

Chapter 4

Further Issues on XML Identifier Query Language

In this chapter, duplicates and copies, which are well-known issues in the area of object-creating query languages, will be studied for the case of XML Identifier Query Language (XIQL). It is shown that XIQL faces the opposite problem to IQL. IQL can eliminate duplicates, but is not complete up to copy elimination; whereas XIQL can eliminate copies, however, an additional approach needs to be employed to handle duplicates when necessary. Naturally, this leads to the notion of hybrid-deterministic transformation developed in our work. Finally, we examine the completeness of XIQL in connection with the classical determinate transformation semantics from the theoretical perspective.

4.1 Duplicate Elimination

Undoubtedly, object identifiers play a crucial role in object-based databases. However, the functions of object identifiers is like a two-sided coin. On one side, object identifiers can facilitate data sharing and infinite data structure manipulation, such as rational trees. On the other side, object identifiers encapsulate the underlying value and structure into an abstract symbol so that objects might be considered as duplicates in the sense of the underlying information they are carrying. Consequently, the duplicate issue in object-based databases has been widely investigated in the literature [65, 13, 66]. In this section, the task is to explore what sort of duplicates might appear in XIQL by extending the traditional concepts to a setting of XML data.

4.1.1 Structured Value Duplicates

Since object identifers are just a means of abstraction to express particular complex values within a database, thus, the link between object identifiers and underlying complex values is our main interest. To clarify the background, we begin with a brief introduction to the research on duplicates and a discussion on how the classical concepts of duplicates could be handled in XIQL.

In general, most research efforts on duplicates refer to duplicates in types at depth one. According to type constructors, basically, duplicates could be split into two kinds: set duplicates and tuple duplicates. Since tuple duplicates could be easily dealt with by incorporating the relational semantics into languages, set duplicates present the major research problem in this area. There are some well-known solutions proposed for solving this problem. The most influential one is to add an explicit primitive into a language, such as the powerset operation [47] or the abstraction operation [32]. Others comprise non-deterministic choice [74], linearly-ordered databases [36], or equality axioms [43].

It has been shown in [65] that IQL can express abstraction so that duplicate elimination can be achieved in IQL. Analogously, because XIQL follows the spirit of IQL with respect to dereferencing and assignment to objects, and thus can simulate set object creation over grouping, XIQL can also express abstraction, and therefore achieve both tuple and set duplicate elimination as well.

However, we cannot yet conclude that duplicates are not an issue for XIQL, because unfortunately, if we consider types of more than depth one in complex values represented by object identifiers, structured value duplicates may occur in XIQL. We first introduce how two objects can be regarded as being V-equivalent as follows.

From a graph point of view, two objects can be defined to be V-equivalent on the basis of the notion of object-structured value that we defined in Section 2.5.

Definition 4.1.1. (V-equivalent) Let o_1, o_2 be two objects, then o_1 and o_2 are considered to be *V-equivalent*, denoted by $\widehat{\widehat{o}}_1 = \widehat{\widehat{o}}_2$, if and only if the following condition is satisfied.

- $\widehat{\widehat{o}}_1$ is isomorphic to $\widehat{\widehat{o}}_2$.

Objects are said to be *structured value duplicates* if they are V-equivalent.

Remark 4.1.1. The departure from the classical object-based database systems implies that issues regarding duplicates in the context of XML databases embrace some new features. On the one hand, duplicates including set and tuple duplicates are no longer the case for XIQL. On the other hand, due to the fact that the document order is significant in XML databases, the uniqueness constraint for each class in object-oriented databases [59, 65], which implies that two different objects in the same class have a different associated value, has been released. As a result, this leads to structured value duplicates being a problem specific to XML databases.

4.1.2 A Main Result

To obtain a better understanding for structured value duplicates, an illustrative example is provided before presenting a result with respect to structured value duplicate eliminations in XIQL.

Recall from Example 3.4.1 that author objects are modelled as atomic objects carrying authors' names directly so that the desired query can be written easily. Now, to demonstrate how structured value duplicates arise from using object identifiers, we slightly change the XML document bib.xml to produce more complicated structured values associating with author objects in the bib2.xml, shown in Figure 4.1.

Because the presence of structured value duplicates is our main interest, the first attempt to write down a query serves for the purpose of figuring out structured value duplicates existing in the result.

Example 4.1.1. For each author in the XML document bib2.xml, list the author's name and the titles of all books by that author.

$$
\begin{aligned}
R_1(A, B, v, t) \leftarrow\ & book(i_b, [author : u, title : t]),\\
& v \in u.\\
\widehat{B}(t') \leftarrow\ & R_1(A, B, v, t),\\
& author(v, [last : l_1, first : f_2]),\\
& R_1(A', B', v', t'),\\
& author(v', [last : l_2, first : f_2]),\\
& \widehat{l_1} = \widehat{l_2},
\end{aligned}
$$

```
<bib2>
  <book year=1994>
    <title>"TCP/IP Illustrated"</title>
    <author>
       <name><last>"Stevens"</last><first>"W."</first></name>
       <age>45</age>
    </author>
    <publisher>"Addison-Wesley"</publisher>
    <price> 65.95</price>
  </book>
  <book year=1992>
    <title>"Advanced Programming in the Unix environment"</title>
    <author>
       <name><last>"Stevens"</last><first>"W."</first></name>
       <age>45</age>
    </author>
    <publisher>"Addison-Wesley"</publisher>
    <price>65.95</price>
  </book>
  <book year=2000>
    <title>"Data on the Web"</title>
    <author>
       <name><last>"Abiteboul"</last><first>"Serge"</first></name>
       <age>41</age>
    </author>
    <author>
       <name><last>"Buneman"</last><first>"Peter"</first></name>
       <age>36</age>
    </author>
    <author>
       <name><last>"Suciu"</last><first>"Dan"</first></name>
       <age>38</age>
    </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>
    <price>129.95</price>
  </book>
</bib2>
```

Figure 4.1: bib2.xml

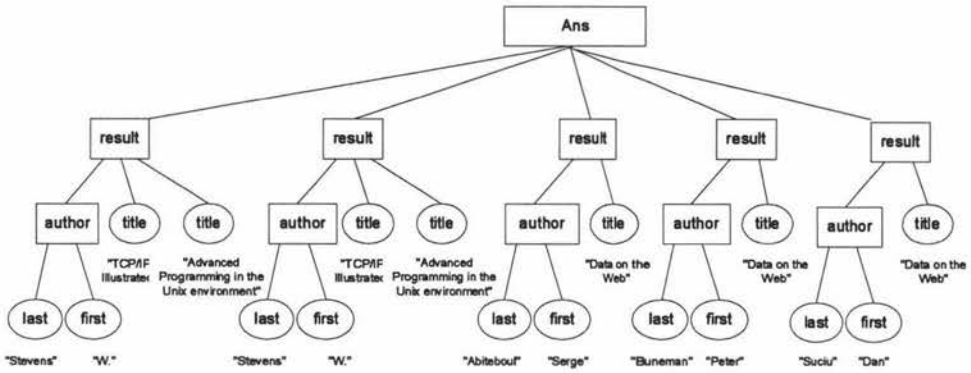


Figure 4.2: Duplicate results

$$\widehat{f}_1 = \widehat{f}_2.$$
$$\widehat{A}(v, \widehat{B}) \leftarrow R_1(A, B, v, t);$$
$$R_2(C) \leftarrow .$$
$$\widehat{C}(A) \leftarrow R_1(A, B, v, t);$$
$$Ans(i, [result \hookrightarrow \widehat{C}]) \leftarrow R_2(C).$$

The result of the query above is shown in Figure 4.2. Obviously, two of the *result* objects regarding the author named "W. Stevens" are structured value duplicates. The underlying reason is that multiple author objects that encode exactly the same information into the same structure, and only differentiate in object identifiers, originally exist in the input database (we assume that they represent the same person in this case).

However, XIQL is powerful enough to handle this problem. The solutions can be given in terms of two situations. One is that only partial object structured values are of interest, whereas another one is to require complete object-structured values to be compared. We will discuss both of them based on the previous example.

In the first case, we assume that the author's name, including *first* and *last* name, is sufficient to identify each concrete author in the real world, then in this case the query can adopt the dominant relation between objects to capture the particular components of structured value of interest, which might be nested into an object with arbitrary depth.

$$R_1(A, B, C, \widehat{f}, \widehat{l}) \leftarrow book(i_b, [author : u]),$$
$$v \in u,$$

$$
\begin{aligned}
& author(v).first(f), \\
& author(v).last(l). \\
\widehat{B}(\widehat{f},\widehat{l}) \leftarrow\ & R_1(A,B,C,\widehat{f},\widehat{l}). \\
\widehat{C}(t) \leftarrow\ & R_1(A,B,C,\widehat{f},\widehat{l}), \\
& book(i_b,[author:u,title:t]), \\
& v \in u, \\
& author(v).first(f), \\
& author(v).last(l). \\
\widehat{A}(B,\widehat{C}) \leftarrow\ & R_1(A,B,C,\widehat{f},\widehat{l}); \\
R_2(D) \leftarrow\ & . \\
\widehat{D}(A) \leftarrow\ & R_1(A,B,C,\widehat{f},\widehat{l}); \\
Ans(i,[result \hookrightarrow \widehat{D}]) \leftarrow\ & R_2(D).
\end{aligned}
$$

In the second case, the whole structured values of *author* objects need to be considered so that the underlying information must be compared to be exactly the same. To handle this situation, we need to introduce non-deterministic semantics as an extra approach for object structured values, so that the arbitrary choice can be done on object structured values of authors during consecutive instantiations in this case. The rewritten query in the second case is provided as follows. For formal semantics, we shall discuss in Section 4.3.2.

$$
\begin{aligned}
R_1(A,C,\widehat{\widehat{v}}) \leftarrow\ & book(i_b,[author:u]), \\
& v \in u. \\
\widehat{C}(t) \leftarrow\ & R_1(A,C,\widehat{\widehat{v}}), \\
& book(i_b,[author:u,title:t]), \\
& v \in u. \\
\widehat{A}(\widehat{\widehat{v}},\widehat{C}) \leftarrow\ & R_1(A,C,\widehat{\widehat{v}}); \\
R_2(D) \leftarrow\ & . \\
\widehat{D}(A) \leftarrow\ & R_1(A,C,\widehat{\widehat{v}}); \\
Ans(i,[result \hookrightarrow \widehat{D}]) \leftarrow\ & R_2(D).
\end{aligned}
$$

Finally, a result regarding structured value duplicates in XIQL can be obtained straightforwardly based on the preceding discussion.

Theorem 4.1.1. *Structured value duplicates can be eliminated in XIQL queries.*

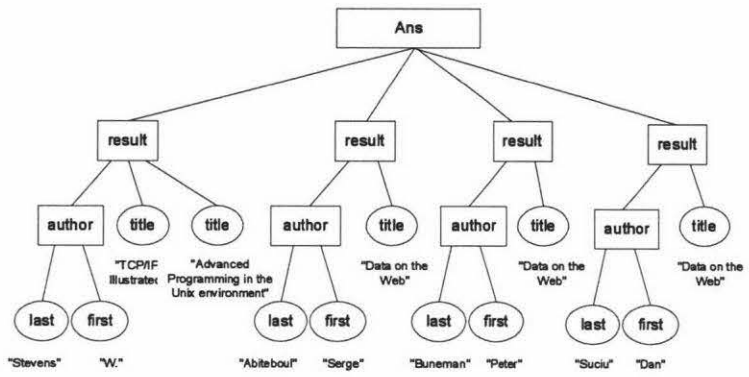


Figure 4.3: Expected results

Remark 4.1.2. In conclusion, XIQL stays in the same line as IQL to eliminate tuple and set duplicates that are the classical perspective regarding duplicates. Furthermore, in case that structured value duplicates are of specific concern to XIQL, either object domains can be used to retrieve particular atomic values of interest or non-deterministic semantics can be adopted to choose one out of multiple structured value duplicates.

4.2 Copy Elimination

The notion of copies was originally introduced in [6] in the context of IQL with a proof that IQL is complete up to copy elimination. After that, a number of research results [65, 67, 69, 27] have been achieved with respect to the relationship between completeness of object-creating query languages and copy elimination. Since XIQL has a close relationship with IQL, I believe that it is quite necessary to conduct an investigation on copy elimination in XIQL.

To be clear, we first review the definition of copies used in [65] as follows.

Given a query $Q(I, J)$, $\overline{Q}$ is said to be *equivalent to Q up to copies* iff $Q(I, J)$ implies $\overline{Q}(I, \overline{J})$, or conversely, $\overline{Q}(I, \overline{J})$ implies $Q(I, J)$ for some $\overline{J}$ with copies of J such that, there are $J_1, ..., J_n$ over S_{out} satisfying the following conditions:

- $f(J_i) = J_i$, where f be a permutation function of the universe of object identities in I;
- J_i and J_j are disjoint outside S_{in} $(i \neq j)$;

- $\overline{J} = J_1 \cup ... \cup J_n$.

In fact, the reason for producing output instances with copies lies in the non-determinism arising from the object creation feature incorporated in languages. More specifically, non-determinism is caused by the use of object identity, which is only an abstract representation associated with concrete structured value, so that a arbitrary choice for object identifers during object creation might lead to copies for the same structured value. However, as pointed out in [7], the object identifiers in the output instance do not matter, nevertheless, their interrelationships do matter.

We outlined in Section 3.3, that the approach adopted in XIQL to create objects is essentially similar to skolem functions. To be more precise, we rewrite the rules of object creation mechanism (see Section 3.2.1) with explicit skolem functions in expressions as follows. Both of these are equivalent with respect to the effects on query results.

(i) Let r be a relation rule of the form

$$R(x_1, ..., x_n, x_{n+1}...., x_m) \leftarrow L_1, ..., L_k,$$

where $x_1, ..., x_n$ are variables only appearing in $head(r)$, and $x_{n+1}, ..., x_m$ are variables appearing both in $head(r)$ and in $body(r)$. Then the rewriting rule with skolem functions has the form

$$R(f_1(x_{n+1}...., x_m), ..., f_n(x_{n+1}...., x_m), x_{n+1}...., x_m) \leftarrow L_1, ..., L_k,$$

where $x_i = f_i(x_{n+1}...., x_m)(n \geq i \geq 1)$, and $f_1, ..., f_n$ are skolem functions.

(ii) Let r be a class rule of the form

$$C(i, [x_1, ..., x_n]) \leftarrow L_1, ..., L_k,$$

where i is a variable only appearing in $head(r)$, and $x_1, ..., x_n$ are variables appearing both in $head(r)$ and in $body(r)$. Then the rewriting rule with skolem functions takes the form

$$C(f(x_1...., x_n), [x_1...., x_n]) \leftarrow L_1, ..., L_k,$$

where $i = f(x_1...., x_n)$, and f is a skolem function.

(iii) For a program Γ containing skolem functions $f_1, ..., f_n$

$$\bigwedge_{1 \leq i < j \leq n} f_i \neq f_j$$

holds.

Since variables that express new objects in a program are always associated with distinct skolem functions, each new object identifier is created within a variable as a function for a set of specified arguments. Thus, it suffices to guarantee that new object identifers are determined in a query, although their concrete symbols are not important, indeed. We notice that a similar result has been shown in [68], in which object creation is interpreted as the construction of function terms over the existing objects.

We are now ready to get a result regarding copy elimination in XIQL.

Theorem 4.2.1. *Copies can be eliminated in XIQL queries.*

Proof. The proof is quite straightforward. Since new object identifiers are always fixed to a specific function and a set of particular arguments, XIQL queries create specific object identifiers in the final results as well. Although object identifiers are abstract and hidden, they can not be chosen arbitrarily in XIQL. Expressed with the words of the definition for copies, it is not possible to produce $\overline{J}$ with copies of J for any XIQL query. □

4.3 Database Transformations

Before leaving the discussion about XIQL, we want to examine this language from a different theoretical angle. We start by introducing determinate transformation [6], then show that XIQL queries can be interpreted using a notion of hybrid-determinism. Finally, we are interested in investigating whether or not XIQL, as an object-creating XML query language, can be complete with respect to determinate transformations.

4.3.1 Determinate Transformations

From a classical database computability point of view, a query can be considered to be a binary relation as a transformation between input instances and output instances. Recall that the semantics of XIQL queries has been formalized in Section 3.2.2 using this standpoint. Although the traditional database applications are considered to be deterministic transformations, both deterministic and non-deterministic transformations have been discussed in the literature [9, 10]. For a query language, deterministic semantics or non-deterministic semantics could make different influences on final results, which has been clearly shown by the example provided in [9].

With the development of object-creating query languages, non-deterministic implication under object creation were widely investigated in [11, 68], and received significant research attention. On this basis, the notion of determinate transformation is developed in [6], which is well-defined on how non-deterministic features of object creation can be incorporated into a "deterministic-effect" transformation so that the requirement for genericity can be achieved in the context of domain-generating query languages.

For the sake of expression convenience, we present the notion of determinate transformation [7, 65] in the following.

A query Q on a database instance I is said to be a *determinate transformation* $\gamma_Q \subseteq I_1 \times I_2$ (see Definition 3.2.1) if and only if the following conditions are satisfied.

(i) for $(I_1, I_2) \in \gamma_Q$, $|I_1| \subseteq |I_2|$ holds;

(ii) for $(I_1, I_2) \in \gamma_Q$ and $(I_1, I_2') \in \gamma_Q$, there is an isomorphic mapping h such that $h(I_2) = I_2'$ holds;

(iii) for $(I_1, I_2) \in \gamma_Q$, if there is an isomorphic mapping h on I_1, then $(h(I_1), h(I_2)) \in \gamma_Q$ holds.

Regarding the above three conditions, condition (i) is to guarantee that objects can be added into a database, and conversely, they are not allowed to be deleted from a database. The reason for this is that determinate transformations can not be preserved under composition when deletion operations involve [68]. As for condition (ii), it implies that two output instances are identical up to renaming of object identifiers. Finally, the purpose of condition (iii) is to achieve genericity of a language regardless of the permutation of the universe.

4.3.2 Hybrid-deterministic Transformations

Recall that Example 4.1.1 has shown how structured value duplicates can be eliminated in XIQL syntactically. Indeed, the underlying consideration is as follows:

Assume that there are two different objects o_1 and o_2 in a database instance, and $\widehat{\widehat{o_1}} = \widehat{\widehat{o_2}}$ holds. Can we say that they reflect the same object in the real world?

The answer is uncertain. They might be the same object, and they also might be two different objects that have the same properties. Hence, we need a capability provided by the language to flexibly express both of these intentions.

In order to express this intuition, the notion of hybrid-determinism is developed to interpret XIQL queries. Simply, hybrid-determinism means the semantics of determinism and non-determinism could be mixed up to serve for XIQL queries when necessary. The use of hybrid-determinism greatly enhance the flexibility for choosing how to implement valuations of individual variables in order to achieve a particular purpose of a query. In essence, the purpose of hybrid-determinism is to provide a capability to solve structured value duplicate problems caused by object identifiers.

To clarify hybrid-deterministic semantics used in XIQL, the formal definition is presented as follows.

Definition 4.3.1. (Hybrid-determinism) Let $Q(I, J)$ be a query on a database instance. Then $Q(I, J)$ is *hybrid-deterministic* iff

(i) If structured value variables $\widehat{\widehat{x}}_1, ..., \widehat{\widehat{x}}_n$ appear in $head(r)$ of value assignment and class rules, then Q can yield more than one different outputs $J_1, ..., J_n (n \geq 1)$. Moreover, the valuations $\widehat{\widehat{v}}_1$,...,$\widehat{\widehat{v}}_k$ for variables $\widehat{\widehat{x}}_i$ are identical if there is a pairwise automorphism mapping h among them such that $h(\widehat{\widehat{v}}_i) = \widehat{\widehat{v}}_j$ $(1 \leq i < j \leq m)$ holds.

(ii) Otherwise, Q yields only one output J.

Here we should be aware of the fact that values for atomic objects can be considered as structured values since they have the same expression. Therefore, we assume that the above definition is based on a rewriting rule: replace all value variables $\widehat{x}$ for atomic objects with structured value variables $\widehat{\widehat{x}}$ in XIQL queries.

4.3.3 Completeness

Since the purpose of determinate transformations is to assess the expressiveness of object-creating query languages in the context of IQL, XIQL is developed in close relationship to IQL. Thus, it would be of interest to know, whether or not XIQL is complete for determinate transformations.

First, let us begin with the result with respect to the completeness of IQL. Abiteboul and Kanellakis's work [7] has shown that IQL is complete up to copy elimination. However, for XIQL our previous discussion has clearly shown that copies do not exist in the result of XIQL queries. Thus, copy elimination is not a problem for XIQL.

To show completeness of XIQL, we can consider the conditions listed in the definition of determinate transformation (see Section 4.3.1) one by one in connection with XIQL. Condition (i) can be obviously satisfied by XIQL, since only objects can be added into output database of XIQL queries. Condition (ii) actually has been enforced by the hybrid-deterministic semantics of XIQL queries. As for condition (iii), atomic elements do not need to be interpreted in XIQL.

Therefore, the following result can be obtained.

Theorem 4.3.1. *XIQL is complete with respect to determinate transformation semantics.*

Remark 4.3.1. Although the notion of determinate transformation is initially motivated by non-deterministic transformations caused by object creation, it can also be applied to capture non-deterministic transformations produced by structured value duplicates in the context of XML data. To sum up, XIQL is a simple but powerful logic programming language that extends the techniques of IQL to the context of XML databases, thus enriching IQL by some additional approaches to handle the unique features of XML data.

Chapter 5

The XML Calculus Language

In this chapter, we propose a novel logic-based query language for XML, called XML calculus, which is a counterpart of relational calculus defined for the relational data model. Instead of first-order logic used in relational calculus, XML calculus shall employ higher-order logics to investigate a natural approach to specifying declarative queries over XML data. The main idea is to extend the relevant techniques used for the relational model to the context of XML data, and exploit the capabilities of a language with higher-order notions.

There are several aspects covered in the content of this section. We first define a fundamental type system as a basis for formalizing higher-order logics incorporated into the language since atomic types and object types defined in Chapter 2 are only able to handle first-order individuals. Then, two higher-order constructs are discussed to explore how higher-order notions can be obtained under the defined type system. After that, a formal presentation of the syntax and semantics used in XML calculus is given, and a logical reflection addressing the relationship between first-order semantics and higher-order semantics interpreted on the higher-order syntax of XML calculus will be examined accompanied by some equality issues. Finally, examples of several interesting issues which should be addressed by desired XML query languages are discussed.

5.1 Preliminaries

To facilitate the formalization, we still use the notations $\mathbb{B}$ and $\mathbb{C}$ to refer to the sets of atomic types and class names, as defined in Chapter 2. Furthermore, the notations $\mathbb{R}$ and $\mathcal{X}$ are adopted to denote countably infinite sets of set names and variables respectively. In addition, we assume that $\mathbb{C}$ and $\mathbb{R}$ are disjoint.

5.1.1 A Fundamental Type System

We begin by introducing a fundamental type system to describe the syntax of a higher-order logic used in the language. The type system of XML calculus is defined by

$$\tau = \oslash \mid b \mid \tau_c \mid l_0 : \tau_0 \mid l_x : \tau_x \mid \{\tau\} \mid [\tau_1, ..., \tau_n] \mid \tau_1 \vee \tau_2$$

where $n \geq 0$. In fact, the type system consists of three parts: individual types, labelled types and type constructors.

1. The individual types comprise a trivial type $\oslash$ which means the existence of type is trivial, all atomic types $b \in \mathbb{B}$, and all class types τ_c with class name $c \in \mathbb{C}$.

2. There are two kinds of labelled types. The type $l_0 : \tau_0$ represents a type τ_0 with a label l_0 such that $l_0 \in \mathbb{C} \cup \mathbb{B}$. The type $l_x : \tau_x$ represents a type variable τ_x occurring with a label variable l_x such that $l_x \in \mathcal{X}$.

3. Furthermore, three type constructors are defined comprising a tuple constructor [ ], a set constructor $\{\}$ and a union constructor $\vee$.

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 $b \in \mathbb{B}$, as defined in Section 2.1;

- $tdom(\tau_c) = \{\delta(o_i) \mid c \text{ is a complex class name}\} \cup \{\pi(o_i) \mid c \text{ is an atomic class name}\}$, where δ and π are identifier and value projection functions defined in Section 2.2;

- $tdom(l_0 : \tau_0) = \{l_0 : v \mid l_0 \in \mathbb{C} \cup \mathbb{B}, v \in tdom(\tau_0)\}$;

- $tdom(l_x : \tau_x) = \{c : v \mid c \in \mathbb{C}, v \in tdom(\tau)$, where v is of type $\tau\}$;

- $tdom(\{\tau\}) = \{\{v_1, ..., v_m\} | m \geq 1, v_i \in tdom(\tau), 1 \leq i \leq m\}$;

- $tdom([\tau_1, ..., \tau_n]) = \{[v_1, ..., v_n] | v_i \in tdom(\tau_i), 1 \leq i \leq n\}$;

- $tdom(\tau_1 \vee \tau_2) = tdom(\tau_1) \cup tdom(\tau_2)$.

Note that the purpose of introducing the trivial type $\oslash$ is to enhance the flexibility of the language when handling optionality of XML data at the type level. Since set and non-set variables are treated in a unified manner, thus, λ in $tdom(\oslash)$ of XML calculus corresponds to both $\emptyset$ and λ appeared in class tuples discussed in Chapter 2 for expression convenience in the language. Furthermore, 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, type $l_x : \tau_x$ is introduced into the type system for this reason. However, to clearly distinguish label variables from other variables and class names, we use the notation $\&x$ referring to a label variable in particular.

Clearly, a type can be obtained recursively from other types and type constructors, thus, recursive types and higher-order types might occur in this type system. To describe higher-order features, the notion of order with respect to type is developed.

Definition 5.1.1. (Type Order) For each type t, the order of t can be obtained by the following rules.

(i) the order of types $\oslash$ and b is 0;

(ii) the order of $[\tau_1, ..., \tau_n]$ is $k+1$, where k is the maximal order of types $\tau_1, ..., \tau_n$;

(iii) the order of $\{\tau\}$ is $k+1$, where k is the order of type τ;

(iv) the order of $\tau_1 \vee \tau_2$ is k, where k is the higher order of types τ_1 and τ_2;

(v) the order of type τ_c is $k+1$ if $\tau_c = [\tau_1, ..., \tau_n]$, where k is the maximal order of types $\tau_1, ..., \tau_n$, and 0 otherwise;

(vi) the order of type $l_0 : \tau_0$ is $k+1$ if $\tau_0 = [\tau_1, ..., \tau_n]$, where k is the maximal order of types $\tau_1, ..., \tau_n$, and 0 otherwise;

(vii) the order of type $l_x : \tau_x$ is $k+1$ if $\tau_x = [\tau_1, ..., \tau_n]$, where k is the maximal order of types $\tau_1, ..., \tau_n$, and 0 otherwise.

For convenience, terms and formulae can also be 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 the formula.

5.1.2 Higher-order Constructs

The XML calculus language is built upon an object base using a higher-order logic, and thus two basic higher-order constructs are employed for representing objects in the language: class construct and set construct.

Definition 5.1.2. (Class Construct) Given a class with name $c \in \mathbb{C}$ and schema $s_c = \{A_1^{f_1}, ..., A_m^{f_m}\}$, then a *class construct* over class schema s_c has the form $L : x(L_1 : x_1, ..., L_n : x_n)$ with $L_j \in \mathcal{X} \cup \mathbb{C} \cup \mathbb{B}$, and each x_j is a variable of type τ_j $(j = 1, \ldots, n)$, which can be obtained in accordance with the following rules.

(i) In case $L_j \in \mathcal{X}$, then $\tau_j = \tau_x$;

(ii) In case $L_j \in \mathbb{C} \cup \mathbb{B}$ and $L_j = A_i$, where $A_i^{f_i} \in attr(s_c)$, then

- for $f_i = 1$, $\tau_j = \tau_{A_i}$;
- for $f_i = ?$, $\tau_j = \tau_{A_i} \vee \oslash$;
- for $f_i = +$, $\tau_j = \{\tau_{A_i}\}$;
- for $f_i = *$, $\tau_j = \{\tau_{A_i}\} \vee \oslash$.

In addition, x is a variable of type $\tau_c = [L_1 : \tau_1, ..., L_n : \tau_n]$ and $L \in \mathcal{X} \cup \mathbb{C}$.

Instead of the notion of class schema used in the SXO model, class types are incorporated into the type system as discussed before. They indeed have the same effect as class schemata with respect to capturing the features of XML data. Informally, the following simple example illustrates how class schemata in the SXO model can be expressed by class types as appropriate.

Example 5.1.1. Recall the class schemata of the XML document book.xml shown in Example 2.2.5. They can be expressed with class types τ_{root}, τ_{book}, $\tau_{section}$, τ_{figure} and τ_{image} respectively as follows.

- $\tau_{root} = [book : \tau_{book}]$;
- $\tau_{book} = [title : \tau_{title}, author : \{\tau_{author}\}, section : \{\tau_{section}\}]$;
- $\tau_{section} = [@id : \tau_{@id} \vee \oslash, @difficult : \tau_{@difficult} \vee \oslash, title : \tau_{title}, p : \{\tau_p\},$
  $figure : \tau_{figure} \vee \oslash, section : \{\tau_{section}\} \vee \oslash]$;

- $\tau_{figure} = [@height : \tau_{@height} \vee \oslash, @width : \tau_{@width} \vee \oslash, title : \tau_{title}, image : \tau_{image}]$;
- $\tau_{image} = [@source : \tau_{@source}]$.

In the following, a variation $c(L_1 : x_1, ..., L_n : x_n)$ is used as a specific representation of class construct $c : c(L_1 : x_1, ..., L_n : x_n)$. In this case, c is both a class name and a variable of type τ_c. This abbreviation will not lead to ambiguity, as we assumed that the sets of class names and set names are disjoint. To get a better understanding for this, let us see one example.

Example 5.1.2. Assume that there are two class constructs:

(1) $book : b(title : t, author : a, price : p)$, and

(2) $book(title : t, author : a, price : p)$,

where t, a and p are variables of types τ_{title}, $\{\tau_{author}\}$ and τ_{price} respectively, then we have $\tau_{book} = [title : \tau_{title}, author : \{\tau_{author}\}, price : \tau_{price}]$ for both class constructs. However, b is a variable of type τ_{book} in class construct (1), whereas $book$ is a variable of type τ_{book} in class construct (2).

Definition 5.1.3. (Set Construct) Suppose $r \in \mathbb{R}$ is a set name and x is a variable of type τ. A *set construct* over r, denoted as $r(x)$, is a finite set of values of type τ. Further, r is a variable of type $\{\tau\}$.

In fact, the definition of set construct coincides with a unary relation used in the relational data model in the sense that $r(x)$ can be treated as a unary relation r that has only one attribute corresponding to x.

Both class and set constructs play an important role in generating higher-order variables since the orders of class and set types increase by applying constructs. More specifically, in a class construct $L : x(L_1 : x_1, ..., L_n : x_n)$, x_0 can be regarded as a variable of order one higher than the maximal order among variables in the expression $L_1 : x_1, ..., L_n : x_n$. The same holds for other variations of class constructs and for set constructs. Note that each label L or $L_i (i \in [1, n])$ might occur as a label variable $\&x$ as defined in the type system.

There are two essential distinctions between class constructs and set constructs:

- Class constructs support the tuple type constructor, whereas set constructs support the set type constructor.
- Class constructs use at least one label-variable pair as the basic components, whereas set constructs only use a single variable as its component.

Therefore, class constructs and set constructs serve different purposes in the language. Class constructs can be used for capturing objects that are the aggregation of other objects, while set constructs provide a necessary means for handling transformation between non-set or set objects.

5.2 Syntax

In this section, we formalize the syntax of the XML calculus language in a style similar to the relational calculus [5]. However, as will be seen, the two languages differ from each other in many aspects.

5.2.1 Terms and Variables

Variables in XML calculus have flexible and rich interpretations in the sense that class names might be regarded as variables as well. For clarity, both term and variable are defined together.

Definition 5.2.1. (Terms and Variables) The *terms* together with their associated types in the language can be defined inductively with the following rules. Furthermore, each term t defines a set of *variables* $var(t)$ occurring in t.

(i) each variable x of type τ is a term of type τ and $var(x) = \{x\}$;

(ii) each constant u of type τ is a term of type τ, and and $var(u) = \emptyset$;

(iii) each pair $L : t$ with term t of type τ is a term of type $L : \tau$, and the variables occurring in it are variables occurring in t or L, i.e. $var(L : t) = var(L) \cup var(t)$;

(iv) each class construct $L_0 : t_0(L_1 : t_1, ..., L_n : t_n)$ with terms $t_0, ..., t_n$ of types $\tau_0, ..., \tau_n$ respectively, is a term of type $\tau_0 = [L_1 : \tau_1, ..., L_n : \tau_n]$, and the variables occurring in it are variables occurring in $L_0, ..., L_n$ and $t_0, ..., t_n$, i.e. $var(L_0 : t_0(L_1 : t_1, ..., L_n : t_n)) = \bigcup_{i=0}^{n}(var(L_i) \cup var(t_i))$;

(v) each set construct $r(t)$ with term t of type τ is a term of type $\{\tau\}$, and the variables occurring in it are r and the variables in t, i.e. $var(r(t)) = \{r\} \cup var(t)$;

(vi) each path expression $t_1 \,.\, t_2$ with terms t_1, t_2 of type τ_1 and τ_2, respectively, is a term of type τ_2, and the variables occurring in it are variables occurring in t_1 or t_2, i.e. $var(t_1 \,.\, t_2) = var(t_1) \cup var(t_2)$.

A term is *ground* if and only if no variable occurs in it. For example, *1* and *title* : "*Databases*" are ground terms.

For convenience, we use $\#t$ in the following to refer to the type of term t.

5.2.2 Well-Formed Formulae

With a set of logic symbols $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \exists, \forall$ corresponding to negation, disjunction, conjunction, implication, equivalence, existential quantification and universal quantification, respectively, the proposed XML calculus language can be represented under a precise logic interpretation. Note that the presence of the symbols $\in$ and $\subseteq$ is not allowed in our work since their interpretations for the standard set membership and inclusion relation can be simulated by particular higher-order logic formulae. For instance, $y \in x$ could be simulated by the expression $x(y)$ and $y \subseteq x$ could be simulated by the expression $\forall_z . y(z) \Rightarrow x(z)$.

We start from the notion of atomic formula, which provides a necessary basis for formulating well-formed formulae later on. Given an XML database schema $\mathbb{S}$, then the set of atomic formulae over $\mathbb{S}$ (for short: *atoms*) consists of four kinds of formulae:

- *predicate formulae*: include class predicates and set predicates.

  (1) $L_0 : t_0(L_n : t_1, ..., L_n : t_n)$, where $L_0 : t_0, ..., L_n : t_n$ are terms over $\mathbb{S}$, is a class predicate;

  (2) $r(t)$, where t is a term over $\mathbb{S}$ and r is a set variable, is a set predicate;

- *path formulae*: expressed by $t_1 \,.\, t_2$, where t_1 and t_2 are terms over $\mathbb{S}$;

- *pair formulae*: expressed by $L : t$, where $L : t$ is a term over $\mathbb{S}$;

- *comparison formulae*: expressed by t_1 *cop* t_2, where t_1 and t_2 are terms of some common type over $\mathbb{S}$, and *cop* represents comparison operators such as $=$, $\neq$, $<$, $>$, $\leq$ and $\geq$.

It is easy to see that some atoms including predicate formulae, path formulae and pair formulae are also terms defined in the last section.

Definition 5.2.2. (Well-Formed Formulae) The set of well-formed formulae of the XML calculus over $\mathbb{S}$, denoted as wff, can be defined inductively with the following rules.

(i) each atomic formula is a wff;

(ii) φ *bop* ψ, where φ and ψ are well-formed formulae over $\mathbb{S}$, and *bop* is a logical symbol amongst $\vee, \wedge, \Rightarrow$ and $\Leftrightarrow$, is a wff;

(iii) $\neg\psi$, where ψ is a well-formed formula over $\mathbb{S}$, is a wff;

(iv) $\exists_x\psi$ or $\forall_x\psi$, where ψ is a well-formed formula over $\mathbb{S}$ and x is a variable occurring in terms in ψ, is a wff.

As with relational calculus, some shortcuts are used in the XML calculus language:

- $\exists_{x_1,x_2,...,x_n}\varphi$ abbreviates successive existential quantification $\exists_{x_1}\exists_{x_2}...\exists_{x_n}\varphi$;
- $\forall_{x_1,x_2,...,x_n}\varphi$ abbreviates successive universal quantification $\forall_{x_1}\forall_{x_2}...\forall_{x_n}\varphi$.

5.2.3 Free and Bound Variables

In this section, we consider how to identify free and bound variables among all variables appearing in some formula of the XML calculus language. The notations $fr(\varphi)$ and $bo(\varphi)$ are used to denote free and bound variables, respectively, in a formula φ. Analogous to relational calculus, free variables can be obtained in accordance with the following rules inductively.

- for an atomic formula φ, which is also a term, including a predicate formula, a path formula or a pair formula, $fr(\varphi) = var(\varphi)$;

- for a comparison formula φ *cop* ψ, which is an atom but not a term, $fr(\varphi \ cop \ \psi) = var(\varphi) \cup var(\psi)$;
- for a formula $\neg\varphi$, $fr(\neg\varphi) = fr(\varphi)$;
- for a formula $\varphi \vee \psi$, $fr(\varphi \vee \psi) = fr(\varphi) \cup fr(\psi)$;
- for a formula $\varphi \wedge \psi$, $fr(\varphi \wedge \psi) = fr(\varphi) \cup fr(\psi)$;
- for a formula $\varphi \Rightarrow \psi$, $fr(\varphi \Rightarrow \psi) = fr(\varphi) \cup fr(\psi)$;
- for a formula $\varphi \Leftrightarrow \psi$, $fr(\varphi \Leftrightarrow \psi) = fr(\varphi) \cup fr(\psi)$;
- for a formula $\exists_x \psi$, $fr(\exists_x \psi) = fr(\psi) - \{x\}$;
- for a formula $\forall_x \psi$, $fr(\forall_x \psi) = fr(\psi) - \{x\}$.

A variable appearing with a quantifier is called a *bound* variable. It may happen that a variable is free and bound in a formula at the same time. However, bound variables can be renamed without changing the semantics. We may therefore achieve that all variables in a formula are either bound or free, but not both.

Example 5.2.1. Given a formula $\psi = \forall_{n,d.} author : a(name : n, address : d) \wedge \exists_{a.} book(title : t, author : a) \wedge author : a(name : n)$, then, $fr(\psi) = \{a, book, t\}$, and $bo(\psi) = \{n, d, a\}$. Obviously, the variable a is both a free variable and a bound variable in the formula ψ.

Through variables renaming, the formula ψ can also be expressed by the formula $\varphi = \forall_{n,d.} author : a_1(name : n, address : d) \wedge \exists_{a_2.} book(title : t, author : a_2) \wedge author : a_2(name : n)$ without changing the semantics.

In fact, the approach of XML calculus can be seen as an extension to the approach of relational calculus by allowing further quantification of higher-order variables as well as first-order variables.

5.3 Semantics

Now we turn to discuss the semantics of terms and formulae interpreted in the XML calculus language.

5.3.1 Interpretation

For the sake of expression convenience, we use the notations $DOM = \bigcup_{i \in n} tdom(\tau_i) \cup \mathbb{C}$ to denote the universal domain consisting of all domains and class names. Given a database schema $\mathbb{S}$, an interpretation to terms and formulae can be defined as usual in logic.

Definition 5.3.1. (Interpretation) An interpretation ω of an XML calculus formula over an XML database schema $\mathbb{S}$ consists of a database $\mathcal{DB}$ over $\mathbb{S}$ and a valuation function ν: $\mathcal{X} \rightarrow DOM$, where $\nu(x) \in tdom(\tau)$ if $x \in \mathcal{X}$ and the type of x is τ, or $\nu(\&x) \in \mathbb{C}$ if $\&x \in \mathcal{X}$. Furthermore, the following rules must be satisfied.

- for a term t of type τ, $\omega(t) \in tdom(\tau)$. In particular, $\omega(t) = \nu(t)$ if t is a variable, and $\omega(t) = t$ if t is a constant;
- for a pair formula, $\omega(L : t) = T$ iff $\omega(L) \in \mathbb{C}$ and $\omega(t) \in \mathcal{DB}$ holds, and F otherwise;
- for a class predicate formula, $\omega(L : x(t_1, ..., t_n)) = T$ iff $[\omega(t_1), ..., \omega(t_n)] \preceq u$ for an object with identifier $\omega(x) \in \mathcal{DB}$ and value $u \in \mathcal{DB}$, and F otherwise (note that $\preceq$ is a subsumption relation defined in Definition 2.2.4);
- for a set predicate formula, $\omega(r(t)) = T$ iff $\omega(t) \in \omega(r)$ holds, and F otherwise;
- for a path formula, $\omega(t_1.t_2) = T$ iff $\omega(t_2) \measuredangle \omega(t_1)$ holds, and F otherwise;
- for a comparison formula, $\omega(t_1 \ cop \ t_2) = T$ iff $\omega(t_1) \ cop \ \omega(t_2)$ holds, and F otherwise;
- for a negation, $\omega(\neg\psi) = T$ iff $\omega(\psi) = F$ holds, and F otherwise;
- for a disjunction, $\omega(\varphi \vee \psi) = T$ iff $\omega(\varphi) = T$ or $\omega(\psi) = T$ holds, and F otherwise;
- for a conjunction, $\omega(\varphi \wedge \psi) = T$ iff $\omega(\varphi) = \omega(\psi) = T$ holds, and F otherwise;
- for an implication, $\omega(\varphi \Rightarrow \psi) = T$ iff $\omega(\varphi) = F$ or $\omega(\psi) = T$ holds, and F otherwise;
- for an equivalence, $\omega(\varphi \Leftrightarrow \psi) = T$ iff both $\omega(\varphi)$ and $\omega(\psi)$ are T, or both $\omega(\varphi)$ and $\omega(\psi)$ are F;
- for an existential quantification, $\omega(\exists_x \psi) = T$ where x is of type τ iff some replacement of x in ψ by a valuation $\nu(x) \in tdom(\tau)$ resulting in a formula φ with $\omega(\varphi) = T$, and F otherwise;

- for a universal quantification, $\omega(\forall_x \psi) = T$ where x with some labels $L_1, ..., L_k$ within ψ iff each replacement of x in ψ by a valuation $\nu(x) \in \bigcup_{1 \leq i \leq k} rdom(\omega(L_i))$, where $rdom(\omega(L_i))$ is a restricted domain for x labelled by L_i in some class construct, as defined below, resulting in a formula φ with $\omega(\varphi) = T$, and F otherwise.

Definition 5.3.2. (Restricted Domain) Given a class with name c and schema s_c, and a set T_c of class tuples over s_c. Then, each variable x_i ($i \in [1, m]$) occurring in a class construct $L : x(L_1 : x_1, ..., L_m : x_m)$ over s_c associates a restricted domain $rdom(\omega(L_i))$ such that $rdom(\omega(L_i)) = \{v | [\omega(L_i) : v] \preceq t$, where $t \in T_c\}$. For variable x of type $\tau_{\omega(L)}$, $rdom(\omega(L)) = tdom(\tau_{\omega(L)})$.

If a variable x associating with multiple restricted domains $rdom(\omega(L_1)), ..., rdom(\omega(L_k))$ within the scope of a universal qualification $\forall_x$, then variable x associates the union of them such that $\bigcup_{1 \leq i \leq k} rdom(\omega(L_i))$.

Example 5.3.1. Consider the XML document book.xml. Assume that we have a formula $\varphi_1 = \forall_{x,y,z}.section(@id : x, title : y, p : z)$, in which $section(@id : x, title : y, p : z)$ is a class construct over the class schema $s_{section}$ shown in Example 2.2.4. Thus, the following results can be obtained.

- $x \in rdom(@id)$, where $rdom(@id) = \{\lambda, i_7, i_{25}\}$;
- $y \in rdom(title)$, where $rdom(title) = \{i_9, i_{12}, i_{15}, i_{27}, i_{35}, i_{38}, i_{41}\}$;
- $z \in rdom(p)$, where $rdom(p) = \{\{i_{10}\}, \{i_{13}\}, \{i_{16}, i_{23}\}, \{i_{28}, i_{33}\}, \{i_{36}\}, \{i_{39}\}, \{i_{42}\}\}$.

Let us see another formula $\varphi_2 = \forall_y.(section(@id : x, title : y, p : z) \wedge figure(title : y, image : w))$, then, the restricted domain of variable y is

- $y \in \bigcup rdom(title)$, where $\bigcup rdom(title) = \{i_9, i_{12}, i_{15}, i_{27}, i_{35}, i_{38}, i_{41}\} \cup \{i_{20}, i_{30}\}$
  $= \{i_9, i_{12}, i_{15}, i_{27}, i_{35}, i_{38}, i_{41}, i_{20}, i_{30}\}$.

Note that labels in different class constructs are always considered as different labels even they have the same name. For instance, in the above example, *title* in *section(@id:x,title:y,p:z)* and *title* in *figure(title:y,image:w)* are two different labels associating with variable y.

5.3.2 Query Mapping

Since XML calculus is a purely declarative language, XML queries can be written down in a very elegant and natural way. The basic form of an XML query is a set expression as defined below.

Definition 5.3.3. (XML Query) Each query Q in the XML calculus language has the form

$$Q = \{(x_1, ..., x_n) | \varphi\},$$

with variables $x_i \in \mathcal{X}$ of type τ_i, a wff φ and $fr(\varphi) = \{x_1, ..., x_n\}$. The answer schema of Q, denoted by $ans(Q)$, is a set of XML schema graphs rooted in objects belonging to $\{\omega(x_1), ..., \omega(x_n)\}$ correspondingly, where $\omega(x_i) \in tdom(\tau_i)(1 \leq i \leq n)$.

For an XML database $\mathcal{DB}$ with database schema $\mathbb{S}$, each query in the XML calculus language is always associated with an input-schema $in(Q) = \mathbb{S}$ and an output-schema $out(Q) = ans(Q)$. In addition, the query mapping $q(Q)$ is defined by

$$q(Q)(\mathcal{DB}) = \{G_{\nu(x_1)}, ..., G_{\nu(x_n)} | \omega(\varphi) = T\}$$

where $G_{\nu(x_1)}, ..., G_{\nu(x_n)}$ be object family graphs corresponding to objects $\nu(x_1), ..., \nu(x_n)$ respectively.

5.3.3 Object Creation

In the XML calculus language new objects can be created during querying in accordance with the logic. For this we have to make the assumption to have disjoint sets of intensional and extensional class types. This is important for the expressive power of the language up to object creation.

Given a query with associated input and output schemata: $in(Q)$ and $out(Q)$, respectively, then the intensional and extensional class types, denoted by $\mathcal{T}_{int}$ and $\mathcal{T}_{ext}$, respectively, can be obtained by the following rules.

- $\mathcal{T}_{ext} = \{\tau_c | \ \tau_c \in in(Q)\}$;

- $\mathcal{T}_{int} = \{\tau_c |\ \tau_c \in out(Q), \tau_c \notin in(Q)\}$.

Obviously, observing from the above definition, $\mathcal{T}_{int} \cap \mathcal{T}_{ext} = \emptyset$ holds. With intensional and extensional class types as a basis, new objects can be created in a database by the following means.

(i) only objects within classes that have intensional types can be created in an XML database;

(ii) object value types of created objects must coincide with class types such that $\tau(o) \sqsupseteq \tau_c$, where $\tau(o)$ is an object value type of object o and τ_c is a class type with class name c.

To better see the mechanism of object creation in XML calculus, the following simple example is provided, in which some *books* objects are created in the database.

Example 5.3.2. Let us consider to list books in the XML document bib1.xml, which were published by Addison-Wesley after 1991, including their year and title.

- $\{books | \forall_{y,t.} \exists_{book.} book(publisher : \text{“}Addison - Wesley\text{”}, @year : y, title : t) \wedge y > 1991$
  $\Rightarrow books(@year : y, title : t)\}$

Clearly, in this query, $\mathcal{T}_{ext} = \{\tau_{book}\}$ and $\mathcal{T}_{int} = \{\tau_{books}\}$. All objects in the class name *books* are new objects created under the class type $\tau_{books} = [@year : \tau_{@year}, title : \tau_{title}]$.

5.3.4 Active Domain Semantics

Although an XML database under the SXO model consists of a finite set of objects, the infinite universe might possibly produce infinite answers for XML calculus queries, as the case in relation calculus. To guarantee both finiteness of query answers and termination of computation, several approaches have been developed in the literature for the relational calculus to address this problem such as safe queries [5], domain independence [1] or active domains [49, 35]. Indeed, these solutions are provided by essentially restricting either syntax or semantics of the language. Nevertheless, it has been proven that they all have the equivalent expressive power [38].

The approach employed in XML calculus to deal with safety issues is to restrict the semantics of the language using active domains. Roughly speaking, the idea is to associate variable assignments with the active domains specific to some database instance, while the underlying domain is still allowed to be infinite. Therefore, in the XML calculus language, a set of active domains is given, which represents a set of objects or constants occurring within a database instance.

Definition 5.3.4. (Active Domain) Given a query Q on an XML database $\mathcal{DB}$, then let $adom(Q)$ be the finite set of constants occurring in the query Q, and $adom(\mathcal{DB})$ be the finite sets of constants and objects occurring in the database $\mathfrak{S}$. The active domains, denoted by $Adom$, consist of $adom(Q)$ and $adom(\mathcal{DB})$ such that $Adom = adom(Q) \cup adom(\mathcal{DB})$.

Under the active domain semantics, a valuation needs to be revisited as "in a database $\mathcal{DB}$, a valuation ν on variable x of type τ in a query Q is of the form $\nu(x) \in tdom(\tau) \cap (adom(Q) \cup adom(\mathcal{DB}))$. This change makes sure that valuations can be limited within the active domains.

5.4 Logical Reflection

It has been pointed out in [13] that higher-order logics have many attractive features, and most of research efforts [22, 42] seem to fall into the same direction to develop the syntax and semantics of higher-order languages by interpreting a high-order syntax with a first-order semantics. This approach enriches the languages by extending the first-order syntax to more natural and powerful expressions in higher-order logics.

Recall that in the SXO model a complete lattice of classes has been proven to be imposed on objects within an object data graph. This result provides a straightforward thought to consider a first-order semantics with object abstraction instead of a higher-order semantics with complex structures. The essential bridge between these two approaches is the type system that has been defined in Section 5.1.1. On this basis, the XML calculus language is developed with a higher-order syntax under a first-order semantics.

In this section, our discussion with respect to the notion of higher-order logics will build upon the classification provided in [21]. A logic is syntactically first-order if its language is

first-order; it is syntactically higher-order otherwise. A system is semantically first-order if the meaning assigned to its constructs by interpreting them in structure is first-order; it is semantically higher-order otherwise.

So far, although the XML calculus language has been formulated to have a higher-order syntax interpreted by a first-order semantics, we are interested in exploiting the relationship between the following two different logic interpretations:

- first-order semantics imposed on higher-order syntax;
- higher-order semantics imposed on higher-order syntax.

To simplify the work, the relevant formal definitions with respect to a higher-order semantics will be omitted since our interest is only on informally comparing and contrasting effects under a first-order semantics and a higher-order semantics imposed on a same syntax of the language.

First of all, let us consider the first situation, which is the defined semantics in XML calculus. Although higher-order variables and quantifiers over such variables are allowed, higher-order variables are interpreted as in the first-order logic, which could only be objects in their domains (represented by object identifiers). One thing should be noted that not only objects but also atomic values can occur in first-order variables of a formula in XML calculus.

Example 5.4.1. Consider the following formula on the XML document book.xml whose XML data graph has been shown in Figure 2.2.

$$\exists_{b,x,y}.book : b(title : x, section : y) \wedge \forall_{w}.(y(w) \Leftrightarrow \exists_{z,t.}(section : w(@id : z, title : t) \wedge z \neq \lambda))$$

Because $\tau_{section} = [@id : \tau_{@id} \vee \oslash, title : \tau_{title}]$ and $\tau_{book} = [tite : \tau_{title}, section : \{\tau_{section}\}]$ hold for the formula above, the variables with their orders can be obtained such that

- 1^{st} order variables: x with $\#x = \tau_{title}$, z with $\#z = \tau_{@id} \vee \oslash$ and t with $\#t = \tau_{title}$;
- 2^{nd} order variables: w with $\#w = \tau_{section}$;
- 3^{rd} order variables: y with $\#y = \{\tau_{section}\} \vee \oslash$;

- 4^{th} order variables: b with $\#b = \tau_{book}$.

In addition, for the variables beyond first-order their valuations are as follows:

- $\nu(w)$=i_6,i_{25};
- $\nu(y)$=$\{i_6, i_{24}\}$;
- $\nu(b)$=i_1.

From these valuations, it is quite straightforward to see that the higher-order variables w, y and b in this example are interpreted as individual object identifiers rather than higher-order semantic structures in XML calculus.

Now, we examine what results this formula will yield, if higher-order variables in the language are interpreted by a higher-order semantics instead of a first-order semantics. The valuations for variables w, y and b can be obtained as follows. (We assume that there is a valuation function ν' defined for higher-order semantics.)

- $\nu'(w)$=*section* : [@*id* : "*intro*", *title* : "*Introduction*"], *section* : [@*id* : "*syntax*", *title* : "*A Syntax For Data*"];
- $\nu'(y)$=$\{$*section* : [@*id* : "*intro*", *title* : "*Introduction*"], *section* : [@*id* : "*syntax*", *title* : "*A Syntax For Data*"]$\}$;
- $\nu'(b)$=*book* : [*title* : "*Data on the Web*", $\{$*section* : [@*id* : "*intro*", *title* : "*Introduction*"], *section* : [@*id* : "*syntax*", *title* : "*A Syntax For Data*"]$\}$].

More intuitively, the graphs corresponding to the above valuations of variables w, y and b are presented in Figure 5.1.

If comparing the object family graphs of objects in valuations of variables w, y and b under first-order semantics with the graphs presented in Figure 5.1, it is not difficult to obverse that the graphs under higher-order semantics are subgraphs of the graphs under first-order semantics. Nevertheless, they can still be regarded as the same expressive power

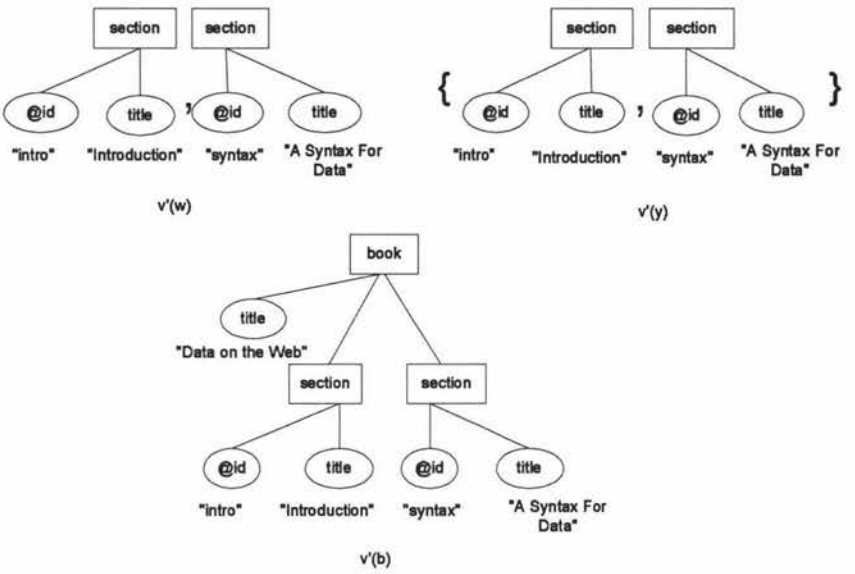


Figure 5.1: Valuation for higher-order variables

because data of interest have been identified by both of them, and a restricting function on types of variables can be easily defined to make sure that these two logic interpretations are the same.

Remark 5.4.1. On the basis of the above example some interesting conclusions can be drawn. First, a first-order semantics for the higher-order syntax can be more elegantly interpreted under an object-based data model, while a higher-order semantics for the higher-order syntax is more suitable for being interpreted under a value-based data model. Moreover, whatever semantics is used to encode the higher-order syntax, there is no loss of expressive power of the language. However, the benefit gained by encoding first-order semantics on higher-order syntax under the object paradigm is to greatly reduce the complexity of the language by providing a simpler and intuitive representation for data.

In the end, we explain in more detail about some issues regarding the equality theory that applies to our work. There are two different facets: equality between the domain of objects and the domain of graphs, and equality between class predicates.

The relationships between first-order and higher-order semantics can be well represented by two different semantics associated with an object in the domain. Recall that each object

has two expressions corresponding to the identifier and value semantics in Section 2.3, these two expressions may lead to an indistinguishable conversion between a first-order semantics and higher-order semantics since it is quite natural for us to treat $i = [A_1 : t_1, ..., A_n : t_n]$ as an extensionality axiom for each object o with identifier i and value $[A_1 : t_1, ..., A_n : t_n]$ in a class c. Furthermore, this extensionality axiom can be extended to the domain of objects and the domain of graphs such that $i = G_i$, where G_i be the object family graph of an object with identifier i.

As for an equality between class predicates, it can be considered as explicitly equated by their class names. For instance, given two class constructs $q = c_1 : x(l_1 : x_1, ..., l_n : x_n)$ and $p = c_2 : y(l'_1 : y_1, ..., l'_m : y_m)$, $q = p$ holds if and only if $c_1 = c_2$, no matter what is inside these constructs. The underlying reason for this equation is that both of q and p are variant expressions of a class with name c_1 or c_2, and type $[L_1 : \tau_1, ..., L_k : \tau_k]$ such that

- $q = [l_1 : x_1, ..., l_n : x_n, l_{n+1} : x_{n+1}, ..., l_k : x_k]$ with $\omega(q) = [\omega(l_1 : x_1), ..., \omega(l_n : x_n), ok, ..., ok]$, where $\{\omega(l_i)\}_{i\in[1,k]} = \{L_j\}_{j\in[1,k]}$;
- $p = [l'_1 : y_1, ..., l'_m : y_m, l'_{m+1} : y_{m+1}, ..., l'_k : y_k]$ with $\omega(p) = [\omega(l'_1 : y_1), ..., \omega(l'_m : y_m), ok, ..., ok]$, where $\{\omega(l'_i)\}_{i\in[1,k]} = \{L_j\}_{j\in[1,k]}$.

Here, ok is a special value referring to T, and it means that the corresponding valuation is always true, since we are not concerned about the details.

5.5 Simulation

The task of this section is to simulate two issues that are relevant for XML:

1. control mechanisms such as recursive structures, if-then-else, composition, and nest and unnest operations, etc;
2. distinguished XML data characteristics such as irregular expressions, object order, schema query queries, etc.

5.5.1 Recursive structures

Unlike relational calculus XML calculus can be enhanced with the fixed-point logic to reflect inductive objects. Here, we discuss one example regarding objects of recursive structure in an XML document.

Example 5.5.1. Consider the XML document book.xml (see Figure A.2) and its XML data graph (see Figure 2.2). Look at a query to find all the sections in book.xml. For such sections, return $section'$ objects with the section titles, while preserving their original hierarchy.

We can obtain the following query

- $Q = \{section' | \forall_{x,y.}((\exists_{section.} section(title : x, section : y) \wedge y = \lambda)$
$\vee \exists_{z.}(\exists_{section.} section(title : x, section : z) \wedge$
$\forall_{s.}(y(s) \Leftrightarrow \forall_{w.}(section : z(title : w) \Rightarrow section' : s(title : w)))))$
$\Leftrightarrow section'(title : x, section' : y)\}$

to get the result such that

$$ans(Q) = \{i_{1011}, i_{1014}, i_{1034}, i_{1040}, i_{1006}, i_{1037}, i_{1024}\}$$

Tables 5.1-5.3 shows the evaluation result at each iteration, and Figure 5.2 presents the graph representations for objects crested in each iteration. In fact, this example also illustrates the approach used in XML calculus queries to handle optionality feature, since variable y can be identified by the λ value when necessary.

5.5.2 Nest and Unnest

As a well-known problem encountered in the development of languages under the object-based paradigm nest/unnest operations are mainly handled by two kinds of approaches. One is to use the ideas underlying the group-by construct of relational languages as e.g. adopted in LDL [64]. It has been proven in [14] that this solution is too restrictive, because the rules involving group-by constructs need to be implemented under a satisfied interpretation for the case of negation, otherwise it is ill-defined for reasons presented in [13]. Another approach is to represent a set of objects by a set object, which has these objects as its value [44, 3].

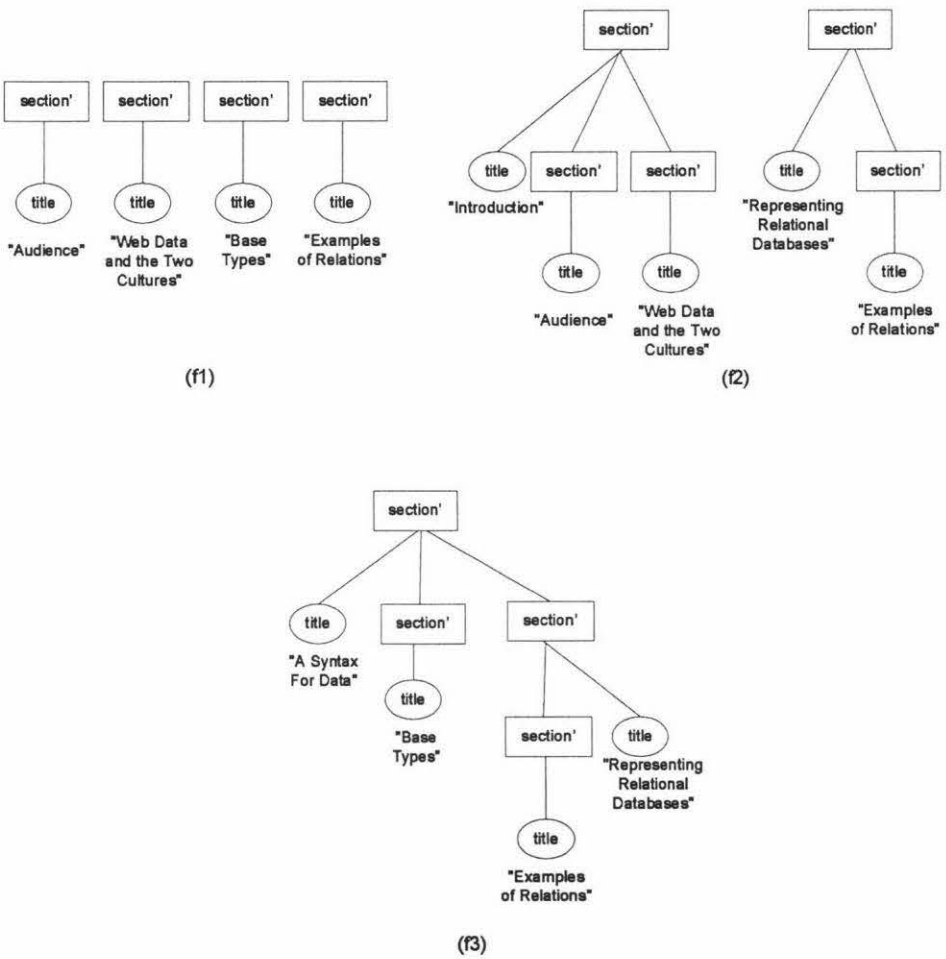


Figure 5.2: Represented as a graph

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

Table 5.1: *section'* objects in f_1

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}

Table 5.2: *section'* objects in f_2

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}

Table 5.3: *section'* objects in f_3

Then, some logical rules need to be taken to guarantee which objects are included in a set object.

The XML calculus stays in the line of argument of the second approach to deal with the problem of nesting and unnesting. For this the set construct is crucial to manipulate objects in a set by providing a type conversion from τ to $\{\tau\}$, where τ is assumed to be the type of objects within the set. The following example shows how nest and unnest operations on objects can be realized in XML calculus.

Example 5.5.2. 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 *result* element.

- $\{ans | \exists_y . \forall_{result.} (\forall_{t,x.} (\exists_{book.} book(title : t, author : x)$

$$\Rightarrow \forall_{a.}(x(a) \Leftrightarrow result(title : t, author : a))) \Leftrightarrow y(result))$$

$$\Rightarrow ans(result : y)\}$$

By using $x(a)$, we can unnest a set of *author* objects in some book into many title-author pairs in *result* objects correspondingly. As for nesting, $y(result)$ is used to nest a set of *result* objects into y with type of $\{result\}$, then y is passed into a set *ans* object.

5.5.3 If-Then-Else

As usual in logic, it is a quite straightforward idea that the semantics of "If-then-else" can be simulated by the logical symbol "$\Rightarrow$" in XML calculus. To be more clear, an example is provided to illustrate this point.

Example 5.5.3. Look at the XML document bib.xml again. Now, for each book with an author, return a *booka* with the book title and its authors. For each book with an editor, return a *booke* with the book title and the editor's affiliation.

- $\{bib|\exists_{x,y.}(\forall_{booka.}(\forall_{t_1,a_1.}(\exists_{book.}book(title : t_1, author : a_1)$

  $\Rightarrow booka(title : t_1, author : a_1)) \Leftrightarrow x(booka))\vee$

  $\forall_{booke.}(\forall_{t_2,a_2.}(\exists_{book,e.}book(editor : e, title : t_2) \wedge editor : e(affiliation : a_2)$

  $\Rightarrow booke(title : t_2, affiliation : a_2)) \Leftrightarrow y(booke)))$

  $\Rightarrow bib(booka : x, booke : y)\}$

Obviously, the logic implicated in the above query could be explained as follows:

- If the book has an author, then produce an object *booka* that contains the book title and its authors;
- If the book has an editor, then produce an object *booke* that contains the book title and the editor's affiliation.

5.5.4 Class Names and Atomic Values

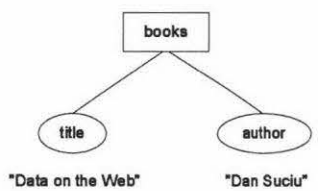
Although the notions of class name and atomic value seem quite irrelevant at the first look, they correspond to two significant portions of the text in an XML document: markup and character data, respectively, as specified in [70]. For this reason, they are discussed together in this section. Moreover, we assume that regular expressions introduced for XIQL in Section 3.4.4 also apply for the XML calculus language.

Example 5.5.4. Let us consider the XML document bib.xml, find books in which the name of some element ends with the string "or". For each such book, return the title and the qualifying element.

- $\{books | \forall_{t,z,\&x.}(\exists_{book.} book(title : t, \&x : z) \wedge \&x = \text{“}\%or\text{”}$

  $\Rightarrow books(title : t, \&x : z))\}$

In this example, $\tau_{book} = [title : \tau_{title}, \&x : \tau_x]$. As discussed in the preceding type system, τ_x is a type variable since the label variable $\&x$ indicates that the class name is unknown. After executing the query on the running database, the following result can be obtained.



To show how atomic values can be queried in XML databases, we slightly change the above question to find books with title of "Data on the Web", in which the name of some element ends with the string "or". For each such book, return the title and the qualifying element. Then, we can get the following query.

- $\{books | \forall_{z,\&x.}(\exists_{book.} book(title : \text{“}Data\ on\ the\ Web\text{”}, \&x : z) \wedge \&x = \text{“}\%or\text{”}$

  $\Rightarrow books(title : \text{“}Data\ on\ the\ Web\text{”}, \&x : z))\}$

5.5.5 Queries over Object Order

When object order is a concern for users, it is easy to tackle this problem by means of functions, which are in the same line with the approach used in XIQL. For this we define a function *postion*(*oid*), and roughly show how object order might be taken into consideration with using such a function through a simple example.

The function *position*(*oid*) with an object identifier *oid* as an argument returns the relative order of the object with identifer *oid* in the current set. Note that this *position*(*oid*) function is defined differently from the one used in XIQL in order to serve a different purpose in the language.

Example 5.5.5. 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.

- $\{ans|\exists_{z.}\forall_{book'.}(\forall_{t_1,x.}(\exists_{y.}(\exists_{book.}book(title:t_1, author:x) \wedge \exists_{a.}(x(a) \wedge position(a) > 2)$
  $\Rightarrow \forall_{b.}(x(b) \wedge position(b) \leq 2 \Leftrightarrow y(b)))$
  $\Rightarrow book'(title:t_1, author:y, et-al:\lambda))$
  $\vee\forall_{t_2,x.}(\exists_{book.}book(title:t_2, author:x) \wedge \forall_{a.}(x(a) \Rightarrow position(a) \leq 2)$
  $\Rightarrow book'(title:t_2, author:x)) \Leftrightarrow z(book'))$
  $\Rightarrow ans(book':z)\}$

5.5.6 Aggregation Functions

The XML calculus language has very strong expressive power so that even some aggregation functions can also be simulated, such as *min*(), which is widely used in most of query languages. Here, we provide one example to show how the semantics of *min*() in a particular context can be simulated by a proper formula in XML calculus.

Example 5.5.6. Consider the XML document prices.xml, find the minimum price for each book in this document, return in the form of a "minprice" element with the book title as its attribute and the minimum price as its element.

- $\{ans|\exists_y.\forall_{minprice.}(\forall_t.(\exists_{book.}book(title:t) \wedge \exists_{p_2.}(\forall_{p_1.}(\exists_{book.}book(title:t, price:x) \wedge x(p_1)$
  $\Rightarrow (x(p_2) \wedge p_1 \geq p_2)))$
  $\Rightarrow minprice(@title:t, price:p_2)) \Leftrightarrow y(minprice))$
  $\Rightarrow ans(minprice:y)\}$

In a similar manner, the function $max()$ can be simulated in the language.

5.5.7 Duplicate Elimination

In contrast to XIQL, the query regarding structured value duplicates discussed in Example 4.1.1 could be rewritten with the XML calculus language as follows. This solution is analogous to the query considered in the first case where we assume that two authors having the same first name and last name could be identified as being the same person.

Example 5.5.7. Consider the XML document bib2.xml (see Figure 4.1). For each author, list the author's name and the titles of all books by that author, grouped inside a "result" element.

- $\{ans|\exists_z.\forall_{result.}(\forall_{f,l.}(\exists_{y,a.}(\exists_{author.}author(first:f, last:l) \wedge$
  $\forall_t(\exists_{book,x.}book(author:x, title:t) \wedge author:x(first:f, last:l) \Leftrightarrow y(t))$
  $\Rightarrow result(name:a, title:y) \wedge name:a(first:f, last:l))) \Leftrightarrow z(result))$
  $\Rightarrow ans(result:z)\}$

Chapter 6

Related Work

Much of the related work has been presented with the relevant issues discussed in previous chapters. In this chapter, we give a brief introduction to the related research work in the fields of semantic data models, object-creating languages and pure declarative languages.

6.1 Semantic Data Models

Although there has been much debate about the origin of semantic data models, the development of semantic data models was initiated in the early 1970s. Since then, a good number of research works [26, 4, 33, 55, 61] have emerged in this field focusing on the investigation of solid semantical foundations of databases from a structural perspective. Currently, one of the most prominent semantic data models is the entity-relationship (ER) model [20, 19], which adopts a mature and diagrammatic technique to model complex data. Although an advantage of the ER model over the relational model is to incorporate more semantic information, it still lacks the ability to clearly state intended semantics such that different semantics can be applied to the same concept [62].

Among these numerous semantic data models, the higher-order entity-relationship model (HERM) [62] and the semantic object model (SOM) [46] attract our attention in particular. Considering that the clear semantics of objects can complement the weakness of HERM with respect to ambiguous semantics, we intend to extend HERM by employing the notion of semantic object to replace the notion of entity and relationship in the context of XML data, especially in the case that XML data can be naturally treated as a tree implied higher-order. Based on these intuitions, the semantic XML object model (SXO Model) is built up

in our work.

Two general approaches taken in constructing semantic models were specified in [34]. One is to interrelate objects by a means of using attributes, such as the Functional Data Model (FDM) [41, 60] and the Semantic Database Model [33]. Alternatively, the explicit type constructors can also be adopted to capture structures, for example, the ER models [20, 19]. The SXO model takes a similar approach to the former. Semantic structures are encoded into attributes of objects, meanwhile, attributes can also provide a linkage to subclass relations, so that a hierarchy can be obtained for object classes within an XML data graph.

To handle XML data, many data models specific to XML data have been proposed serving as a basis for developing XML query languages. As stated in [15], these data models can be divided into two categories in accordance with different interpretations for IDREFs: reference or string. The interpretation of IDREFS as references between elements might lead to a cyclic data model, whereas only containment relationships exist between nodes when IDREFs are treated as strings. It is also possible to consider both interpretations into one data model by providing multiple modes, such as LOREL [8], which has two modes: semantic and literal. In the SXO model, IDREFs are regarded as other attributes, thus, a tree structure of XML data is taken into consideration throughout this thesis.

6.2 Object-Creating Languages

Object-creating languages, as pointed out in [58], can be considered as one of three groups classified in the field of theoretical investigations for the object paradigm. The language IQL [6] has been widely recognized as being a significant framework in this area because it was the first one to extend some language-theoretic issues, such as completeness, to encompassing object creation. Since the emergence of IQL, many research efforts have been reported for object-creating languages [69, 65, 68, 67, 40, 37, 28]. This field has also been receiving increasing attention from database theory researchers.

To create objects in databases, IQL adopts a similar approach to skolem function techniques, which was first discussed in [50] from a logic-based perspective, and further developed in [44, 43, 39, 42, 40, 8, 29]. Nevertheless, these object-creating languages may depart from

each other by other factors during object creation, such as different approaches for inheritance used in F-logic [42] and IQL [6], the possibility of explicitly referencing object identifiers [8, 29]. In our work, the semantics of XIQL defines implicit skolem functions to create object identifiers which can be referenced by means of variables.

To consider new objects in query and update results, a completeness criterion was proposed by the notion of determinate transformation in IQL [6, 7]. Although IQL is a natural extension of languages in [10], which are complete with respect to [18], it is only complete for determinate transformation up to copy elimination. Therefore, to achieve completeness of the language, two interesting research streams are generated as follows.

One direction is to check whether the notion of determinate transformation is the most suitable criterion to evaluate the expressive power of object-creating languages. It was argued in [67, 65] that the notion of determinate transformation may not be the most appropriate one for object-creating languages, and they proposed an alternative notion of semi-determinism. However, although semi-determinism is an elegant generalization of determinacy, it is undecidable.

Another direction is to solve the problem of copy elimination by developing new constructs [7, 6, 28]. For example, a tractable construct called *reduce* is introduced in [28] based on the expanded notions of deep equality. However, the reduce construct remains global in nature. Finding a simple and local construct which yields a complete language is still an open problem.

A few rule-based query languages have been introduced for querying over XML data in the literature [51, 48]. Most of them either resembles the syntax of F-Logic, such as WebLog [48], or are XPath-based including Xpathlog [51]. In contrast, XIQL is a logic-based XML query language developed by studying an analogue to IQL in the context of an XML data model, on the basis of well-defined semantical analysis.

6.3 Pure Declarative Languages

Relational calculus was introduced by Edgar F. Codd [24, 25] as a pure declarative logical language in the relational model. There are two different but equivalent versions of relational

calculus: Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC). The major distinction between the two lies in the approach adopted for representation of variables; variables in TRC range over relations, while variables in DRC range over underlying domains associated to individual values. Compared with TRC, DRC is closer to traditional predicate logic with first-order syntax and semantics.

Numerous extensions of relational calculus have been developed under complex value data models in [5, 12, 57, 30]. A complex value calculus that can express the transitive closure of a binary relation with fixpoint semantics was explored in [5]. The calculus presented in [12] focuses on complex objects by defining an equivalence relation on complex objects. These languages might be based on the paradigm of beyond first-order logic.

In the literature, there has been very little work on the pure logic formalisms for XML query languages, especially extending the similar techniques used in relational calculus to higher-order logic languages built upon XML data models. To fill this gap, XML calculus, which is a novel higher-order logical language, is proposed in this thesis.

The notion of higher-order logic is often considered from Church's Simple Theory of Types [23]. A separation between syntax and semantics of higher-order logic has been identified by Chen's work [22, 21]. According to this opinion, the research efforts in the field of higher-order logics can be categorized into three directions. Some languages, such as LDL [14, 54], are developed with a first-order syntax and a higher-order semantics. Some languages are encoded with higher-order syntax and semantics, for example, COL [3] and λcalculus. Others might have a logic with a higher-order syntax but a first-order semantics, including F-logic [42] and HiLog [22, 21]. To simplify the language, but maintain the desirable higher-order features, XML calculus keeps in the line of languages having higher-order syntax and first-order semantics.

Toward realizing a 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 with the notion of typed λ-expression, and has been adopted in some languages, such as λProlog [53, 52]. Another approach that is used in XML calculus is to achieve the logic reflection by identifier and value semantics of objects, which associate two different aspects with each object. In this case, object identifiers function as first-order abstractions which encapsulate higher-order underlying structures.

Chapter 7

Conclusions

This thesis focuses on logic-based query languages for XML databases. A first area of investigation is an XML data model. We built up a specific XML data model on object bases, and showed that heterogeneous semantic structures can be encapsulated into individual values of objects, which are unified under flexible class schemata as class tuples. Furthermore, a complete lattice has been proven to exist among the classes of an XML data graph, thus an elegant class hierarchy can be constituted. In the object paradigm, we also observe that object identifers cannot only serve as the identity of objects, but also be order primitives.

On the basis of this data model two novel logic-based query languages have been proposed in the thesis with formal specifications of syntax and semantics. One is the XML identity query language (XIQL), in which object identifiers function as pure primitives of the language. Another one is the XML calculus, which is endowed with an elegant semantics based on higher-order logics. For a better understanding of the ideas several examples were provided to illustrate the key characteristics of languages. During our study, several important results have been obtained with respect to these languages.

- Union types could be adopted as an efficient and natural approach to address the problem of optionality appeared in XML query languages. Therefore, union types have been considered in the development of both XIQL and XML calculus.

- Although XIQL is an extension of IQL targeted to XML data, an approach by implicitly specifying a particular skolem function to each variable inventing object identifiers can help avoid copies generated in the queries results, thus copy elimination is not a concern for XIQL. However, the tree structure of XML data might raise the problem

of structured value duplicates in the query results. To address this issue, the notion of hybrid-determinism has been defined in the semantics of XIQL. Finally, XIQL has been proven to be complete with respect to determinate transformations.

– It has been shown that the XML calculus is a very natural language tailored for XML since a higher-order type system can be easily built upon the SXO model. There are two points we want to mention particularly. Firstly, the identifer and value semantics associated with objects provide the capability to shift the semantics of the language from higher-order to first-order without any loss of expressive power of the language. Secondly, by defining restricted domains for variables bounded with universal quantifications, the XML calculus can elegantly reflect the fixed-point semantics to incorporate recursion functionality. Therefore, it is possible to express recursive structures and transitive closure in the language.

Both proposed languages focus on setting up logical formalisms, which greatly facilitate the evaluation of the expressive power and completeness of XML query languages. However, limited by study time there are still some issues left out in this thesis, and it will be interesting to finish them in future research.

- Refine the XML calculus language and investigate more deeply the decidability of the approach to safety taken for this language.
- Investigate the expressive power and completeness of the XML calculus language in a formal way.
- Compare the expressiveness of several XML query languages such as XIQL, XML calculus and XQuery [73].

Appendix A

Running XML Database

```
<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>
    <price>65.95</price>
  </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>
    <price>129.95</price>
  </book>
</bib>
```

Figure A.1: bib.xml

```
<book>
 <title>"Data on the Web"</title>
 <author>"Serge Abiteboul"</author>
 <author>"Peter Buneman"</author>
 <author>"Dan Suciu"</author>
 <section id="intro" difficulty="easy" >
  <title>"Introduction"</title>
  <p>"T1"</p>
  <section>
   <title>"Audience"</title>
   <p>"T2" </p>
  </section>
  <section>
   <title>"Web Data and the Two Cultures"</title>
   <p>"T3"</p>
   <figure height="400" width="400">
    <title>"Traditional client/server architecture"</title>
    <image source="csarch.gif"/>
   </figure>
   <p>"T4"</p>
  </section>
 </section>
 <section id="syntax" difficulty="medium" >
  <title>"A Syntax For Data"</title>
  <p>"T5"</p>
  <figure height="200" width="500">
   <title>"Graph representations of structures"</title>
   <image source="graphs.gif"/>
  </figure>
  <p>"T6"</p>
  <section>
   <title>"Base Types"</title>
   <p>"T7"</p>
  </section>
  <section>
   <title>"Representing Relational Databases"</title>
   <p>"T8"</p>
   <section>
    <title>"Databases"</title>
    <p>"T9"</p>
   </section>
  </section>
 </section>
</book>
```

Figure A.2: book.xml

```
<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 A.3: prices.xml

Bibliography

[1] ABITEBOUL, S., AND BEERI, C. On the power of languages for the manipulation of complex objects. Tech. Rep. Institut National de la Recherche en Informatique et Automatique T, 1988.

[2] ABITEBOUL, S., CLUET, S., AND MILO, T. Correspondence and translation for heterogeneous data. *Theor. Comput. Sci. 275*, 1-2 (2002), 179–213.

[3] ABITEBOUL, S., AND GRUMBACH, S. COL: a logic-based language for complex objects. 347–374.

[4] ABITEBOUL, S., AND HULL, R. IFO: a formal semantic database model. *ACM Trans. Database Syst. 12*, 4 (1987), 525–565.

[5] ABITEBOUL, S., HULL, R., AND VIANU, V. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] ABITEBOUL, S., AND KANELLAKIS, P. C. Object identity as a query language primitive. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1989), ACM Press, pp. 159–173.

[7] ABITEBOUL, S., AND KANELLAKIS, P. C. Object identity as a query language primitive. *J. ACM 45*, 5 (1998), 798–842.

[8] ABITEBOUL, S., QUASS, D., MCHUGH, J., WIDOM, J., AND WIENER, J. L. The lorel query language for semistructured data. *Int. J. on Digital Libraries 1*, 1 (1997), 68–88.

[9] ABITEBOUL, S., AND VIANU, V. Procedural and declarative database update languages. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1988), ACM Press, pp. 240–250.

[10] ABITEBOUL, S., AND VIANU, V. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci. 43*, 1 (1991), 62–124.

[11] ABITEBOUL, S., AND VIANU, V. Non-determinism in logic-based languages. *Ann. Math. Artif. Intell. 3*, 2-4 (1991), 151–186.

[12] BANCILHON, F., AND KHOSHAFIAN, S. A calculus for complex objects. In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems* (New York, NY, USA, 1986), ACM Press, pp. 53–60.

[13] BEERI, C. A formal approach to object-oriented databases. *Data Knowl. Eng. 5*, 4 (1990), 353–382.

[14] BEERI, C., NAQVI, S., RAMAKRISHNAN, R., SHMUELI, O., AND TSUR, S. 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* (New York, NY, USA, 1987), ACM Press, pp. 21–37.

[15] BONIFATI, A., AND CERI, S. Comparative analysis of five xml query languages. *SIGMOD Rec. 29*, 1 (2000), 68–79.

[16] CARDELLI, L. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types* (New York, NY, USA, 1984), Springer-Verlag New York, Inc., pp. 51–67.

[17] CHAMBERLIN, D., ET AL. *XQuery from the experts: a guide to the W3C XML query language.* Addision-Wesley, 2003.

[18] CHANDRA, A., AND HAREL, D. Computable queries for relational database systems. *Journal of Computer and System Sciences 21*, 2 (1980), 156–178.

[19] CHEN, P. P. Er - a historical perspective and future directions. In *Proceedings of the 3rd Int. Conf. on Entity-Relationship Approach (ER'83)* (1983), C. G. Davis, S. Jajodia, P. A. Ng, and R. T. Yeh, Eds., North-Holland, pp. 71–77.

[20] CHEN, P. P.-S. The entity-relationship model: toward a unified view of data. *SIGIR Forum 10*, 3 (1975), 9–9.

[21] CHEN, W., KIFER, M., AND WARREN, D. S. HiLog: A first-order semantics for higher-order logic programming constructs. In *NACLP* (1989), pp. 1090–1114.

[22] CHEN, W., KIFER, M., AND WARREN, D. S. HILOG: A foundation for higher-order logic programming. *J. Log. Program. 15*, 3 (1993), 187–230.

[23] CHURCH, A. A formalulation of the simple theory of types. *Journal of Symbolic Logic 5*, 56-68 (1940).

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

[25] CODD, E. F. 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* (1972).

[26] CODD, E. F. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (1979), 397–434.

[27] DENNINGHOFF, K., AND VIANU, V. Database method schemas and object creation. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1993), ACM Press, pp. 265–275.

[28] DENNINGHOFF, K., AND VIANU, V. Database method schemas and object creation. In *PODS '93: Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1993), ACM Press, pp. 265–275.

[29] DEUTSCH, A., FERNANDEZ, M., FLORESCU, D., LEVY, A., AND SUCIU, D. A query language for XML. In *WWW,Xpath '99: Proceeding of the eighth international conference on World Wide Web* (New York, NY, USA, 1999), Elsevier North-Holland, Inc., pp. 1155–1169.

[30] ENDERTON, H. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[31] FROHN, J., LAUSEN, G., AND UPHOFF, H. Access to objects by path expressions and rules. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), Morgan Kaufmann Publishers Inc., pp. 273–284.

[32] GYSSENS, M., PAREDAENS, J., AND VAN GUCHT, D. A graph-oriented object database model. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1990), ACM Press, pp. 417–424.

[33] HAMMER, M., AND LEOD, D. M. Database description with SDM: a semantic database model. *ACM Trans. Database Syst. 6*, 3 (1981), 351–386.

[34] HULL, R., AND KING, R. Semantic database modeling: survey, applications, and research issues. *ACM Comput. Surv. 19*, 3 (1987), 201–260.

[35] HULL, R., AND SU, J. On the expressive power of database queries with intermediate types. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1988), ACM Press, pp. 39–51.

[36] HULL, R., AND SU, J. On accessing object-oriented databases: expressive power, complexity, and restrictions. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1989), ACM Press, pp. 147–158.

[37] HULL, R., AND SU, J. Untyped sets, invention, and computable queries. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1989), ACM Press, pp. 347–359.

[38] HULL, R., AND SU, J. Domain independence and the relational calculus. *Acta Informatica 31*, 6 (1994), 513–524.

[39] HULL, R., AND YOSHIKAWA, M. ILOG: declarative creation and manipulation of object identifiers. In *Proceedings of the sixteenth international conference on Very large databases* (San Francisco, CA, USA, 1990), Morgan Kaufmann Publishers Inc., pp. 455–468.

[40] HULL, R., AND YOSHIKAWA, M. On the equivalence of database restructurings involving object identifiers (extended abstract). In *PODS '91: Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1991), ACM Press, pp. 328–340.

[41] KERSCHBERG, L., AND PACHECO, J. A functional data base model. Tech. rep., Pontificia Univ. Catolica do Rio de Janeiro, Brazil.

[42] KIFER, M., AND LAUSEN, G. 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 (New York, NY, USA, 1989), ACM Press, pp. 134–146.

[43] KIFER, M., LAUSEN, G., AND WU, J. Logical foundations of object-oriented and frame-based languages. Tech. Rep. TR-90-003, 1, 1990.

[44] KIFER, M., AND WU, J. A logic for object-oriented logic programming (Maier's O-Logic revisited). In *PODS* (1989), pp. 379–393.

[45] KIM, J., HAN, T., AND LEE, S. K. Visualization of path expressions in a virtual object-oriented database query language. In *Database Systems for Advanced Applications, Proceedings of the Sixth International Conference on Database Systems for Advanced Applications (DASFAA), April 19-21, Hsinchu, Taiwan* (1999), A. L. P. Chen and F. H. Lochovsky, Eds., IEEE Computer Society, pp. 99–108.

[46] KROENKE, D. *Database Processing: Fundamentals, Design and Implementation*, seventh edition ed. Prentice Hall, Englewood, NJ, 1999.

[47] KUPER, G. M., AND VARDI, M. Y. The logical data model. *ACM Trans. Database Syst. 18*, 3 (1993), 379–413.

[48] LAKSHMANAN, L. V. S., SUBRAMANIAN, I. N., AND SADRI, F. A declarative language for querying and restructuring the web. In *RIDE '96: Proceedings of the 6th International Workshop on Research Issues in Data Engineering (RIDE '96) Interoperability of Nontraditional Database Systems* (Washington, DC, USA, 1996), IEEE Computer Society, p. 12.

[49] MAIER, D. *The Theory of Relational Databases.* Computer Science Press, 1983.

[50] MAIER, D. A logic for objects. In *Proc. of the Workshop on Foundations of Deductive Databases and Logic Programming* (Washington D.C., 1986), pp. 6–26.

[51] MAY, W. *A Logic-Based Approach to XML Data Integration.* Habilitation thesis, Universitt Freiburg, 2001.

[52] MILLER, D., AND NADATHUR, G. Higher-order logic programming. In *Proceedings on Third international conference on logic programming* (New York, NY, USA, 1986), Springer-Verlag New York, Inc., pp. 448–462.

[53] NADATHUR, G. *A higher-order logic as the basis for logic programming.* PhD thesis, Philadelphia, PA, USA, 1987.

[54] NAQVI, S., AND TSUR, S. *A logical language for data and knowledge bases.* Computer Science Press, Inc., New York, NY, USA, 1989.

[55] PECKHAM, J., AND MARYANSKI, F. Semantic data models. *ACM Comput. Surv. 20,* 3 (1988), 153–189.

[56] ROBIE, J., LAPP, J., AND SCHACH, D. XML query language (XQL). In *Proc. of the Query Languages workshop* (Cambridge, Mass, Dec. 1998).

[57] ROTH, M. A., KORTH, H. F., AND SILBERSCHATZ, A. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst. 13,* 4 (1988), 389–417.

[58] SCHEWE, K.-D. The type concept in OODB modelling and its logical implications. In *EJC* (1999), pp. 256–274.

[59] SCHEWE, K.-D., AND THALHEIM, B. Fundamental concepts of object oriented databases. *Acta Cybern. 11,* 1-2 (1993), 49–84.

[60] SHIPMAN, D. W. The functional data model and the data language DAPLEX. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1979), ACM Press, pp. 59–59.

[61] TER BEKKE, J. H. *Semantic data modeling,* first edition ed. Prentice Hall International, 1992.

[62] THALHEIM, B. *Entity-Relationship Modeling: Foundations of Database Technology.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[63] TRUJILLO, J., KING, G. A., AND PALOMAR, M. Semantic data modelling for databases, issues of modelling and teaching the paradigm. In *ISSEU'97: Proc. of International Simposium on Software Engineering in Universities* (Rovaniemi, Finlandia, 1997).

[64] TSUR, S., AND ZANIOLO, C. LDL: A logic-based data language. In *VLDB* (1986), pp. 33–41.

[65] VAN DEN BUSSCHE, J. *Formal Aspects of Object Identity in Database Manipulation.* PhD thesis, University of Antwerp, 1993.

[66] VAN DEN BUSSCHE, J., AND PAREDAENS, J. The expressive power structured values in pure OODB's (extended abstract). In *PODS '91: Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1991), ACM Press, pp. 291–299.

[67] VAN DEN BUSSCHE, J., AND VAN GUCHT, D. Semi-determinism (extended abstract). In *PODS '92: Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (New York, NY, USA, 1992), ACM Press, pp. 191–201.

[68] VAN DEN BUSSCHE, J., AND VAN GUCHT, D. Non-deterministic aspects of object-creating database transformations. In *Selected Papers from the Fourth International Workshop on Foundations of Models and Languages for Data and Objects* (London, UK, 1993), Springer-Verlag, pp. 3–16.

[69] VAN DEN BUSSCHE, J., VAN GUCHT, D., ANDRIES, M., AND GYSSENS, M. On the completeness of object-creating database transformation languages. *J. ACM 44*, 2 (1997), 272–319.

[70] WORLD WIDE WEB CONSORTIUM. Extensible markup language (XML) 1.0 (third edition). W3C Recommendation, february 2004. Editors: Tim Bray and Jean Paoli and C. M. Sperberg-McQueen and Eve Maler and François Yergeau.

[71] WORLD WIDE WEB CONSORTIUM. XML path language (XPath) 2.0. W3C Working Draft, september 2005. Editors: Anders Berglund and Scott Boag and Don Chamberlin and Mary F. Fernández and Michael Kay and Jonathan Robie and Jérôme Siméon.

[72] WORLD WIDE WEB CONSORTIUM. XML query use cases. W3C Working Draft, september 2005. Editors: Don Chamberlin and Peter Fankhauser and Daniela Florescu and Massimo Marchiori and Jonathan Robie.

[73] WORLD WIDE WEB CONSORTIUM. XQuery 1.0: An XML query language. W3C Working Draft, april 2005. Editors: Scott Boag and Don Chamberlin and Mary F. Fernández and Daniela Florescu and Jonathan Robie and Jérôme Siméon.

[74] ZANIOLO, C. Object identity and inheritance in deductive databases–an evolutionary approach. In *Proc. 1st Intl. Conf. on Deductive and Object-Oriented Databases* (Kyoto, Japan, 1989), Elsevier Science Publishers, pp. 2–19.