International Journal of Database Theory and Application
Vol.7, No.5 (2014), pp.171-186
http://dx.doi.org/10.14257/ijdta.2014.7.5.13

A Formal Description of XML Tree Pattern Query for XQuery Language

Husheng Liao, Xiaoqing Li and Hang Su

Beijing University of Technology, Beijing, China

{liaohs@, li_xiaoqing@emails.,suhang@}bjut.edu.cn

Abstract

In order to express tree pattern query in query plan and take advantage of formal method to analyze its behavioral characteristics, this paper present a formal description of tree pattern query based on functional language and denotational semantics. This description major focuses on behavior of a tree pattern query on matching against an eXtensible Markup Language (XML) document tree. First, we introduce a formal definition for a kind of extended generalized tree pattern (GTP++). Then we present a functional tree pattern description language (XTPL) for GTP++ and give its complete denotational semantics based on a novel data structure, named WTree, which efficiently organizes this typical XML data query results and provides flexible data access method. In the end, we present the formal semantics of identifying tree pattern from path expressions. By using formal methods, the semantics of tree pattern query is consistent and analyzable. As the core operation of XML query, this formal description can provide an initial step for analyzing the correctness of XML queries, and improves the reliability and robustness of query processing methods.

Keywords: *XML; tree pattern query; XTPL; denotational semantics; XQuery*

1. Introduction

EXtensible Markup Language (XML) has become the de facto standard for information exchange and sharing among various applications on the internet. Due to the increasingly widely used, how to improve XML query efficiency has become an important issue in the field of XML data processing in recent years. However, being different from the relational data, XML is a kind of semi-structured data, the query processing and optimization for XML also has its own particularity. The most prominent feature is the tree pattern query (TPQ), also called twig query. This kind of tree-shaped query against tree-structured XML data appears widely in XML query requests described by XML query languages such as XPath and XQuery, and is considered to be the core operation of XML data query. TPQ can fully reflect the characteristic of semi-structured data processing.

In order to standardize the XML data query and processing, W3C has developed XQuery as standard XML data query language. Since XQuery is a kind of XML data query language as well as functional programming language. High-performance implementation of XQuery needs to use query optimization methods provided by XML query algebra, also needs to use efficient holistic twig matching algorithm. Tree patterns are graphical re-presentations of queries over data trees [1]. Although graphical tree pattern can intuitively reflect XML query

ISSN: 2005-4270 IJDTA

requirements, it cannot adapt to the flexibility and the complexity of the XQuery language since lack of precise semantics and effective results organization. Therefore, there are several challenges on integrating tree pattern query into algebraic frameworks. Firstly, the form of tree pattern query request is tree-shaped, and the query results are generated by pattern matching. It is difficult to de-scribe tree pattern query and results reference using function calls in query plan. Secondly, there is much more to XQuery evaluation than a simple tree pattern matching. Sub-queries results can be organized by various ways and appeared at different positions simultaneously in XQuery expressions. However, it is difficult to access sub-queries results from unfolded multiple sequences generated by nested FLWOR clauses. Therefore, both the evaluation strategies and results organization of tree pattern query should be supported by query plan description language. To solve these issues, in this paper, we propose a kind of XML tree pattern description language, named XTPL, and give its complete denotational semantics to analyze behavioral characteristics of tree pattern query [2]. The main contributions of the paper are as follows:

(1) In order to contain more XQuery query semantics into high-performance tree pattern query, we introduce an extended XML tree pattern, called GTP++, which extends GTP in [3] with logic node AND, OR, NOT, wildcard and various predicates, and is able to express the query in nested FLWOR expressions;

(2)We develop a novel data structure, named WTree, which effectively organizes the tree pattern query result and provides flexible data access method. With this data structure, intermediate result can be avoided unfolding into multiple sequences with duplicate XML elements, thereby save memory overhead. Moreover, any sub-query results in a tree pattern can be accessed arbitrarily;

(3) Based on WTree structure, taking generalized list as data model, we present the functional XML tree pattern description language, named XTPL, and give its complete denotational semantics. As intermediate language used for the realization of tree pattern queries, XTPL not only has strong ability to abstract description, but also has simple grammar structure and clear semantic which facilitate to program analysis. XTPL denotational semantics provide formal definition for tree pattern queries, and make it possible to use formal method to analyze behavioral characteristics of tree pattern queries. Meanwhile, it contributes to verify correctness of XML queries, and improve the reliability and robustness of query processing method. More importantly, it can be seamless integrated into FXQL and realized in the framework in [4];

(4) To utilize tree pattern query for effectively realization of XQuery, it is inevitable to analysis the query plan and identify tree pattern from it. Due to limited space, this paper gives the denotational semantics of rewriting path expression in XQuery to tree pattern in XTPL, which is the core part of tree pattern identifying.

The remainder of this paper is organized as follows. Section II introduces the related work and Section III describes GTP++ and WTree structure. Then we present XTPL abstract syntax and denotational semantics in detail in Section IV. In Section V, we discuss the Identify-ing rules for GTP++ and Section VI concludes the paper.

2. Related Work

With XML becoming a ubiquitous language for data interoperability purposes in various domains, efficiently querying XML data is a critical issue. This has lead to the design of algebraic frameworks based on tree-shaped patterns akin to the tree-structured data model of XML [1]. Both XML tree pattern matching and query algebraic have been widely studied in recent years.

On the one hand, as the core operation of XML query processing. The representation of TPQ and its matching algorithm have been widely studied in the past ten years. TAX in [5] first introduces the notion of pattern tree in which edges specify the structural constraints between XML nodes, including ancestor-descendant (AD) relationship and parent-child (PC) relationship. There has been much work towards algorithms for matching such pattern against XML data efficiently, such as two-phase algorithms in [6][7] and one-phase algorithms in [8][9]. Bruno et al. in [6] proposed the first holistic twig join algorithm, *TwigStack*, which is claimed that is optimal for AD relationship. Lu et al. in [7] proposed *TJFast* algorithm based on extended Dewey to access only leaf elements. To avoid unnecessary path merger existed in two-phase algorithms, *Twig²Stack* in [8] uses complex hierarchical-stacks instead of enumeration of path matches to avoid the merging phase. Qin et al in [9] proposed another one-phase algorithm, *TwigList*, which uses a much simpler data structure, a set of lists, to store the final solutions. Since basic tree pattern query which only contains structural constrains and node test is a small part of XQuery query. In order to contain more XQuery semantics in high-performance tree pattern query, many studies have extended XML tree pattern with different features and developed corresponding matching algorithm. For instance, GTP in [3] extends the pattern tree by extending the tree pattern to include semantics related to output nodes, optional nodes, and boolean expressions which are part of the XQuery language. Reference [10] makes a study on extended XML tree pattern which includes PC, AD relationships, negation functions, wildcards and order restriction. Based on their theoretical framework, they propose *TreeMatch* algorithm to process the extended XML tree pattern efficiently. Reference [11] utilizes the semantic structure of the XML data being queried to process twig queries and develop OTQ. In addition, many TPQs have been proposed for particular optimization purposes. For example, logic operators AND, OR, XOR and NOT are introduced in[11]. The TPQ proposed in [13] is designed for XML graphs data. Whereas, almost all of studies on tree pattern queries focus on queries algorithm, the tree patterns are graphical representation. Fig. 1 shows a simple XPath query Q1 and its corresponding graphical tree pattern TPQ1, where single/double edge denotes PC/AD relationship respectively and capital letter denotes the node type of node test. In Fig. 1(c), grey node denotes the matched XML node, and solutions are enumerated under XML document tree.

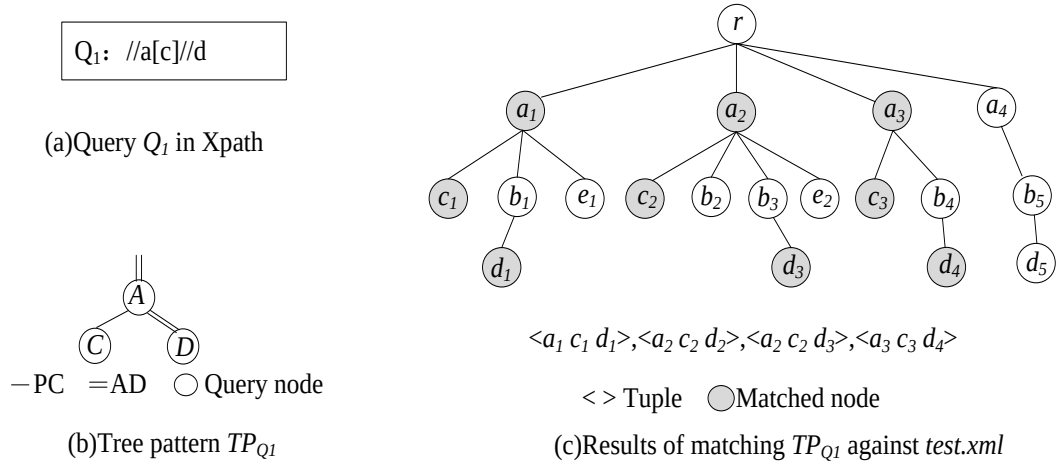


Figure 1. A Case of Tree Pattern and Its Matching

On the other hand, note that a TPQ that cannot be expressed in a physical form is usually considered useless [1]. A number of algebras techniques have been developed for XQuery implementations [4][14]. The XQuery compiler built in [15] is claimed that is complete, correct, and efficient. In order to support tree pattern query, Michiels et al. in [16] put a tree pattern operator in the query plan that allow an XQuery compiler to detect when efficient tree pattern algorithms can be used. Based on the lambda calculus, reference [4] develops a concise functional intermediate language for XQuery implementation, called Functional XML XQuery Language (FXQL), which is based on query algebra technology for XQuery and compiler technology, and proposes a framework for XQuery system with XML algebra and tree pattern query.

3. Preliminaries

3.1. GTP++ With Its Representation

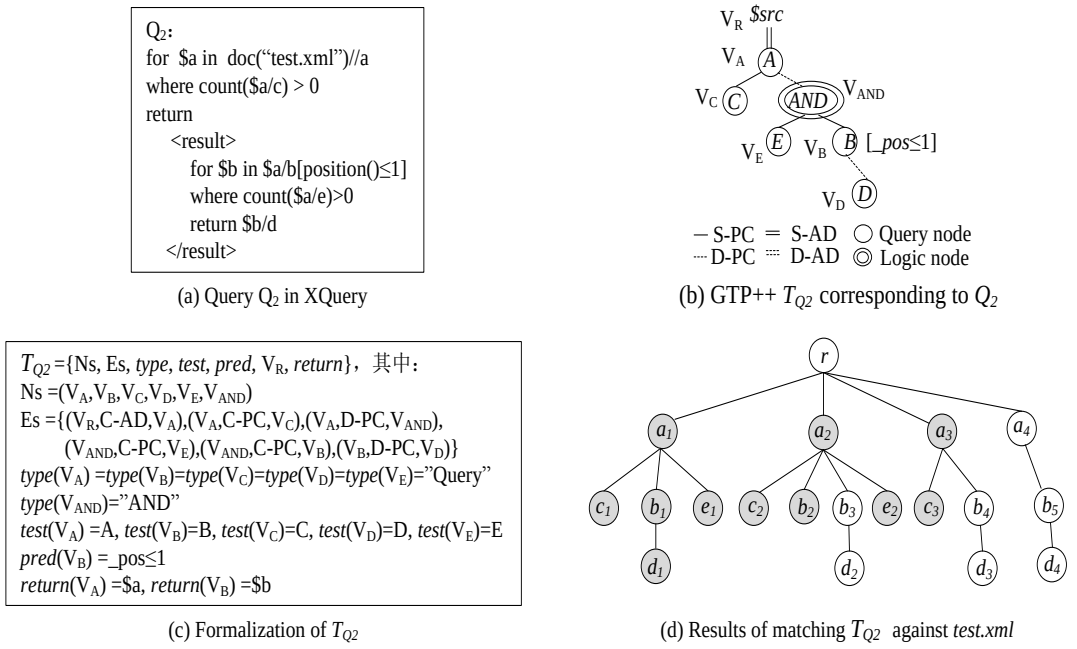
GTP++ is an extended generalized tree query pattern. It extends GTP in [2] with logic AND node, OR node, NOT node, wildcard and various predicates, which is able to express the tree pattern query request in nested FLWOR expressions. There are four kinds of nodes in GTP++: query node, AND node, OR node and NOT node. Query node can bind to variables, indicating that it is a return node. Besides, any query node can be annotated with predicates. There are four kinds of edges in GTP++: compulsory PC/AD relationship and optional PC/AD relationship. Optional relationship indicates that the matching of corresponding to a sub-tree pattern query is not essential, but the matched XML nodes will also be returned as query results.

Definition 1 A GTP++ *T* is a septuple (QNode, Edge, *type*, *test*, *pred*, root, *return*), where:

- QNode is the set of all nodes in GTP++;
- Edge = QNode × BindType × QNode, is the set of all edges in GTP++, which describe the structural constraints between nodes, where BindType = {C-PC, C-AD, O-PC, O-AD} represent the compulsory PC, AD, and optional PC, AD relationship between nodes respectively;
- *type*: QNode → {QUERY, AND, OR, NOT}, is the partial function of evaluating node type. Given any one node $q_n \in$ QNode, *type*(q_n) describes node type of q_n, which is one of four type, query nodes type QUERY, logic nodes type AND, OR, and NOT.
- *test*: QNode → NodeTest, is the partial function of evaluating node test conditions;
- *pred*: QNode → Exp*, is the function of describing predicate expressions for every $q_n \in$ QNode.
- root∈ QNode, is the root of the GTP++;
- *return*: QNode → Ide, gets the branch variable reference for given query node.

Figure 2 shows an example of GTP++, T_{Q2}, which is corresponding to a sample XQuery program, Q_2, and its matching results against an XML document tree, *test.xml*. The graphical GTP++ derived from Q_2 is shown in Fig. 2(b), which is composed of five query nodes, V_A, V_B, V_C, V_D, V_E, notated by single circle, and a logic node, V_{AND}, notated by double circles. Solid/dotted edge denotes compulsory/optional relationship. Single/double edge denotes PC/AD relationship. Query nodes in GTP++ can be annotated with predicate constraints, for example, query node V_B is required to satisfy the predicate *position() <= 1*, that is, only the first *B-type* XML nodes which has *C-type* siblings was committed to the final results. According to definition 1, its formalization is shown as Fig. 2(c). Grey XML nodes in Fig. 2(d) are matched XML nodes for T_{Q2}.

In Q_2, the first *for-where* clauses describe the compulsory conditions that must be satisfied, that is, there must be *A-type* elements which have *C-type* children. Thereby the structure relationship between them is compulsory PC relationship, which is identified as edge (V_A, C-PC, V_C). Edge (V_B, O-PC, V_D) means that a *B-type* XML element which has at least one *E-type* sibling commit to T_{Q2} even though it does not contain *D-type* children. Meanwhile, there is nested FLWOR in *return* clause, which applies on the results of the previous query, but the results of this kind of nested queries can be empty. The nested *for-where* clauses use the previous query results, and *B-type* child and *E-type* child must exist both at once although they are optional relationship for previous query. Therefore, for this situation, it adds logic AND, and provides optional relationship between query node and logic node in tree pattern, such as edge (V_A, D-PC, V_{AND}). Similarly, there may be some situations need logic OR and NOT in tree pattern to describe XQuery queries. This shows that GTP++ T_{Q2} is able to represent the query request of Q_2 with the help of AND operation and optional relationship, while the GTP does not have this capability.



(a) Query Q_2 in XQuery

(b) GTP++ T_{Q2} corresponding to Q_2

(c) Formalization of T_{Q2}

(d) Results of matching T_{Q2} against *test.xml*

Figure 1 A Case of GTP++ and Its Matching

3.2. WTree Structure

Data structure is a crucial factor that often impairs the efficiency of traditional algorithms for evaluating tree pattern queries. XQuery 1.0 and XPath 2.0 Data Model (XDM) recommended by W3C is composed of atomic values, XML nodes and sequences. Because it is hard to determine that which sub-sequence corresponds to which expression calculation from this kind of data model, it is difficult for logic optimization. In order to better support the logic and optimization of the query, some XML query algebra learn tuples concept from relational databases. However, the basic structure of XML data is tree shape structure. If adopt tuples completely, there must be an excess of reduplicative intermediate data. Meanwhile, there may be more than one return node in a tree pattern query for XQuery, and those return nodes will be referenced in various

ways by diverse parts of loop body. Tree pattern branch variable provides reference method for different return nodes. For certain query node, it can assess its arbitrary sub-query results by branch variable expression. Further, starting with these sub-query results, their sub-query results can be got easily.

Based on above analysis, it is needed to add additional data structure in XTPL data model which represent the results of tree pattern queries.

Definition 2 Given a tree pattern, T_Q = (QNode, Edge, type, test, pred, root, return), the WTree instance of T_Q, notated W_Q, is a triple (WNode, WEnv, WMatch), where:

- WNode: Nodes × WMatch, is the set of WTree instance nodes, where Nodes is defined in XDM;
- WEnv: Ide→QNode, is the function to describe bind relationship between tree pattern branch variable and query node in T_Q;
- WMatch: QNode → WNode*, is the function to evaluate all WNode instances for a given query node in T_Q.

According to definition 2, tree pattern query results are saved in a mapping table which represents the function relationships in WMarch. Each XML node matched by the corresponding tree pattern query node is encapsulated in a WNode instance. Besides that XML node, there are a number of sub-queries results saved by mapping table, which are the query results of the sub-tree rooted by that XML node. When it is needed to get some part of the query results, it firstly obtains the query node from WEnv by tree pattern branch variable, and then it gets WNode instances matched by the query node from the corresponding WMarch instance. From this kind of WNode instance, not only XML node, but also the sub-queries results can be obtained straightforwardly.

Continue the above example, Figure 3 shows the WTree instance W_{Q2}, corresponding to T_{Q2} against *test.xml* in Figure 2.

W_{Q2}=({ba_1,ba_2,ba_3,bb_1,bb_2,bc_1,bc_2,bc_3,bd_1,be_1,be_2},Env, march)
Env($a)=$V_A$, Env($b)=V_B
march(V_A)=[ba_1,ba_2,ba_3] , ba_1 =<a_1,$march_{a1}$>, ba_2 =<a_2,$march_{a2}$>,
ba_3 =<a_3,$march_{a3}$>
$march_{a1}$(V_B)=[bb_1],bb_1=<b_1,$march_{b1}$>,$march_{b1}$(V_D)=[bd_1],bd_1=<d_1,nil>
$march_{a1}$(V_C)=[bc_1],bc_1=<c_1,nil>
$march_{a1}$(V_E)=[be_1],be_1=<e_1,nil>
$march_{a2}$(V_B)=[bb_2],bb_2=<b_2,$march_{b2}$>,$march_{b2}$(V_D)=[ ]
$march_{a2}$(V_C)=[bc_2],bc_2=<c_2,nil>
$march_{a2}$(V_E)=[be_2],be_2=<e_2,nil>
$march_{a3}$(V_B)=[ ]
$march_{a3}$(V_C)=[bc_3],bc_3=<c_3,nil>
$march_{a3}$(V_E)=[ ]

Figure 1. WTree instance W_{Q2}

4. XML Tree Pattern Language

As discussed in Section 2, GTP++ can represent XQuery query written by nested FLWOR clauses. The results still need be processed according to the processing logic described by FLWOR, further be organized as XML nodes sequence. Thereby, GTP++ should be organized as a query plan, that is, it needs a kind of intermediate language to describe GTP++. Since functional language has the characteristics such as simple structure and reference transparency, etc. We develop the functional XML tree pattern language to

describe GTP++, named XTPL. The following tree pattern mentioned in this paper all refer to GTP++ without special instruction.

4.1. Abstract Syntax

Table 1 and Table 2 show syntactic domain and abstract productions of XTPL respectively, where symbols in bold are keywords. An expression *e* can be a variable reference, axis operation, or a function. Function parameters can be expressions or anonymous functions, which support nested function calls and functional parameters. The expression $ide_1.ide_2$ is *branch variable reference* expression, which is composed of two variables divided by symbol '.'. The former, ide_1, is the bind variable in *with* clause for a tree pattern and represents the tree pattern query results, while the latter, ide_2, is tree pattern branch variable name. Branch variable reference is used to achieve tree patter results according to bind variable. The structure of *with* clause is composed of return results expression *e* and a tree pattern *tb* which was bound to a specified variable. The non-terminal *tb* stands for a tree pattern query request, which indicates that the tree pattern query will be applied on computation results of given expression, and the results will be referred with the specified variable. Nodes in tree pattern are represented in *tn*, including query node, AND node, OR node and NOT node. Query node with binding denotes return node, where the bind variable can be used to access the matched XML nodes. Such variables called *tree pattern branch variable*. Among various *tn* representations, symbol '?' in bold indicates that the structural constraint is optional, while the match option of the other nodes are compulsory. Non-terminal *ts* shows the representation of node tests and predicates. Recursive definition of *tn* structure is used to describe the hierarchical relationships of the nodes in tree pattern. In the description of predicate, any expression or anonymous function can be used.

Table 1. Syntactic Domain of XTPL

Syntactic domain	Interpretation
e:Exp	Expression
ide:Ide	Identifier(name of variable and function)
axis:Axis	Name of Axis operation
test:Test	Node test
tb: TBind	Root of tree pattern
tn: TNode	Query node in tree pattern
ts: TStep	Query step
arg: Arg	Argument
fun (*ide**){*e*}	Anonymous function

Table 1. Abstract Production Rules of XTPL

Production	Interpretation
e::= *ide*	Variable name
e::= *axis* **(***e*$_1$, *test* ([*e*])***)**	Axis operation
e::= *ide* **(***arg**)	Function call
e::= ide_1**.** ide_2	Get tree patter result
e::= *e* **with** *ide=tb*	With expression

tb::= *e* **{** *tn** **}**	Root of tree pattern
tn::= *ide*= **?**?*ts* {*tn* * } |	Query node with binding
tn::=**??***ts* { *tn** } |	Query node without binding
tn::=**??and** {*tn* * } |	AND node
tn::=**??or**{*tn* * } |	OR node
tn::=**??not**{*tn* }	NOT node
ts::= (/|//)? *test* ([**fun** (*ide**) {*e*}])* |	Element query step
ts::= (/**@**|//**@**")?*test*([**fun** (*ide**){*e*}])*	Attribute query step
arg::= *e*| **fun** (*ide**) { *e* }	Argument

Symbols in bold are keywords

According to the above syntax, the XTPL program corresponding to T_{Q2} is shown in Figure 4. The tree pattern branch variable reference $\$t.\$r_2$ represent the final return results. It is a remarkable fact that XTPL only covers the tree pattern query request and can be used in the realizations of XPath and XQuery, but it does not cover complete XQuery semantics.

```
$t.$r2
with  $t= doc("test.xml")
            { $a=//a
               { $r1=/c,
                 ?and ($b=/b[fun(_dot,_pos,_sz)le(_pos,1)]{ $r2=?/d }
               }
             }
```

Figure 1. WTree Instance W_{O2}

5. Denotational Semantics of XTPL

5.1. Data Model

In XTPL program, results of a given query node in WTree instance can be accessed by tree pattern branch variables, therefore it is unnecessary to save the information of tree pattern query nodes. Then, we merge WEnv and WMatch as WMap to represent tree pattern queries results in XTPL data model. Finally, the production rules of XTPL data model are shown in Table 3. Actually, it is a generalized list, and each item in the list can be an XML node, atomic values, or a generalized list which represents sub-query results rooted by certain XML node.

Table 1. Data Model of XTPL

Production	Interpretation
List::=Elem*	Generalized list
Elem::=Item | List	Data item or generalized list
Item::=Node | Atom | WNode	XML node, atomic values, or WNode
WNode::=Node WMap	XML node with its sub-queries results
WMap::=WBind*	Containing several branch variable
WBind::=Idn WNode+	Binding branch variable to WNode instances

5.2. Semantics Domain

The semantics of XTPL expression is captured by the expression evaluation, that is, XTPL data model instance.

Table 4 shows the semantics domain of XTPL. The data type *Value* is the set of data model instances. As discuss in above, it can be an XML node, atomic values, sub-query results rooted by certain XML node, or generalized list composed of above data items. The evaluation environment, *Env* is a mapping table from identifier to data model instance or function closure. When the identifier is a tree pattern branch variable, it is mapped to corresponding data model instance. If it is a function name, then the identifier is mapped to function closure. The function closure is a triple where *Ide**, *Exp*, and *Env* represent virtual arguments, function body, and external environment respectively.

Table 1. Semantic Domain of XTPL

Production	Interpretation
v,z,x,y,dum:Value	Set of data model instance
d:Value	Current item
p:Integer	Index of current item
s: Integer	Size of current sequence
tt,ff: Boolean=(True,False)	Boolean
u,w:Env = (Ide →Value + Closure)	Evaluation environment
Closure = Ide* × Exp × Env	Function closure

5.3. Semantic Functions

Table 5 shows the semantics functions of evaluating XTPL expressions.

Table 1. Semantic Functions of XTPL

Semantic Functions	Interpretation
$[\![]\!]_{Exp}$: Exp → Env → Value	Expression evaluation
$[\![]\!]_{Arg}$: Arg →Env → Closure + Value	Argument evaluation
$[\![]\!]_{Step}$: TStep→Value→Env→Value	Query step evaluation
$[\![]\!]_{TBind}$: TBind→Env→WMap	Tree pattern evaluation
$[\![]\!]_{TNode}$:TNode→Env→Node→Bool×WBind*	Node evaluation

The semantics function $[\![]\!]_{Exp}$ evaluates XTPL expressions according to the input expression and evaluation environment. The semantic function $[\![]\!]_{Arg}$ calculates the value of actual argument or function closure of anonymous functions according to the input XTPL argument and evaluation environment. The semantic function $[\![]\!]_{Step}$ is a single XPath step evaluation function. It calculates the value after processing this XPath step according to query step, current context, and evaluation environment. The evaluation result of semantic function $[\![]\!]_{TBind}$ is a WTree instance. The semantic function $[\![]\!]_{TNode}$ is an evaluation function for tree pattern node. For a given tree pattern node, it obtains the boolean value which infer whether matching successful or not, or the WBind instances set of query results.

5.4. Semantics Equations

In order to facilitate description, some auxiliary functions used in semantics equations are listed in Table 6. Given function name *fn* and arguments list, the auxiliary function *call* active the *fn* function call. The auxiliary function *map* executes given node test for each item of the expression calculation results. Each node test is processed through a physical operator which is named by axis operation and takes current item and node test marker as arguments. The auxiliary function *getNodeList* is used to get WNode instances according to given tree pattern branch variable.

Table 1. Auxiliary Functions

Auxiliary Functions	Interpretation
name(*ide*)	Get built-in function name
call(*fn*, arg_1, …, arg_n)	Function call
map(*v*, f) = if *v*=[ ] then[ ] else cons(f(*v*[1]), map(*v*[2..], f))	Node test
filter(*v*, f) = fil(v,1, size(v)) where fil(*v*, *p*, *s*) = if *v*=[ ] then [ ] if fil(*hd*(*v*),*p*,*s*) then *cons*(*hd*(*v*), fil(*tl*(*v*), *p*+1, *s*)) else fil(*tl*(*v*), *p*+1, *s*)	Process Predicate
cons: Elem × List→List	Construct list
hd : List → Elem	Get list head
tl : List → List	Get list tail
node:Elem→Boolean	Whether an XML node or not
newWNode: Node→WMap→WNode	Construct WNode instance
newWMap: WBind*→WMap	Construct WMap instance
newWBind: ide→WNode*→WBind	Construct WBind instance
getNodeList: WNode→Ide→WNode*	Get WNode instances

Semantics equations listed in Table 7 illustrate how semantic functions perform for each grammar structure. Semantics equation (1) shows that the value of a variable is obtained from evaluation environment. Semantics equation (2) processes function calls which applies built-in function on the calculation results of arguments. The semantics of tree pattern branch variable is illustrated through semantics equation (3). It declares that all WTree instances bound to branch variable ide_2 are obtained from the query results bound to branch variable ide_1. Semantics equation (4) explain if expression is a tree pattern binding then the branch variable of tree pattern query results and the results should be add to current evaluation environment and generate the new evaluation environment simultaneously.

Table 2. Semantics Equations of XTPL

Semantics Equations	Number
$[\![ide]\!]_{Exp}\ u = u(ide)$	(1)
$[\![ide\ (arg_{1,\ldots,}\ arg_n)]\!]_{Exp}\ u = call(fn, [\![arg_1]\!]_{Arg}\ u, \ldots, [\![arg_n]\!]_{Arg}\ u)$ where $fn=name(ide)$	(2)
$[\![ide_1.\ ide_2]\!]_{Exp}\ u =$ **if** $u(ide_1) = \emptyset$ **then** error **else** $getNodeList(u(ide_1), ide_2)$	(3)
$[\![e$ with $ide=tb]\!]_{Exp}\ u = [\![e]\!]_{Exp}\ w$ where $w = u$ ++{< *ide*, $[\![tb]\!]_{TBind}\ u$ >}	(4)

$[\![e\{ tn_1, \ldots, tn_n \}]\!]_{\text{TNode}}\, u =$ (5)
if node(v) **then** *newWNode*(v,*newWMap*(fn(v)))
else *newWNode*(*dum*, *newWMap*(bd_1++ bd_m)) where (bd_1++ bd_m)=*map*(v,*fn*)
where $v = [\![e]\!]_{\text{Exp}}\, u$
fn(z)= **if** m_1=…=m_n='*tt*' **then** tws_1 ++ … ++ tws_n
else [ ] where (m_i, tws_i) =$[\![tn_i]\!]_{\text{TNode}}\, z\, u$ for i=1,…,n

$[\![ide = ts\{tn_1 \ldots tn_n\}]\!]_{\text{TNode}}\, x\, u$= **if** $ns \neq [\,]$ **then** ('*tt*', [*newWBind(ide, map*(y, *fn*))]) **else** ('*ff*', [ ]) (6)
where y = $[\![ts]\!]_{\text{Step}}$ x u;
fn(z) = **if** m_1=' *ff*' or… or m_n=' *ff* '**then** [ ]
else *newWNode*(z,*newWMap*(tws_i++…++ tws_n))
where (m_i, tws_i) = $[\![tn_i]\!]_{\text{TNode}}\, z\, u$ for i=1,…,n

$[\![ide = ?ts\ tn_1 \ldots tn_n]\!]_{\text{TNode}}\, x\, u$ = ('*tt*', [*newWBind*(*ide*, map(y, fn))]) (7)
where y =$[\![ts]\!]_{\text{Step}}$
fn(z) = if m_1='*ff*' or … or m_n=' *ff*' then [ ] else *newWNode*(*z*,*newWMap*(tws_i++…++tws_n))
where (m_i, tws_i) = $[\![tn_i]\!]_{\text{TNode}}\, z\, u$ for i=1,…,n

$[\![ts\{tn_1 \ldots tn_n\}]\!]_{\text{TNode}}\, x\, u$ = **if** $tws \neq [\,]$ **then** ('*tt*', tws) **else** ('*ff*', [ ]) (8)
where tws = *map*($[\![ts]\!]_{\text{Step}}\, x\, u$, fn);
fn(z) = **if** m_1='*ff*' or … or m_n='*ff*' **then** [ ] **else** tws_i ++ … ++ tws_n
where (m_i, tws_i) = $[\![tn_i]\!]_{\text{TBind}}\, z\, u$ for i=1,…,n

$[\![$ **?***ts* $\{tn_1 \ldots tn_n\}$ $]\!]_{\text{TNode}}\, x\, u$ = ('*tt*', *map*($[\![ts]\!]_{\text{Step}}\, x\, u$, fn)) (9)
where fn(z) = **if** m_1=' *ff*' or … or m_n=' *ff*' **then** [ ] **else** tws_i ++ … ++ tws_n
where (m_i, tws_i) = $[\![tn_i]\!]_{\text{TBind}}\, z\, u$ for i=1,…,n

$[\![$ **and** ($tn_1, \ldots, tn_n$) $]\!]_{\text{TBind}}\, x\, u$ = **if** m_1='*tt*' and…and m_n='*tt*' **then** ('*tt*', tws_1 ++…++ tws_n) **else** ('*ff*', [ ]) (10)
where (m_i, tws_i) = T $[\![tn_i]\!]_{\text{TBind}}\, x\, u$ for i=1,…,n

$[\![$ **or**($tn_1, \ldots, tn_n$) $]\!]_{\text{TBind}}\, x\, u$ = **if** m_1='*tt*' or … or m_n='*tt*' **then** ('*tt*', tws_1 ++ … ++ tws_n) **else** ('*ff*', [ ]) (11)
where (m_i, tws_i) = $[\![tn_i]\!]_{\text{TBind}}\, x\, u$ for i=1,…,n

$[\![$ **not**(*tn*) $]\!]_{\text{TBind}}\, x\, u$ = **if** m='*tt*' **then** ('*ff*', [ ]) **else** ('*tt*', tws) where (m, tws) = $[\![tn]\!]_{\text{TBind}}\, x\, u$ (12)

$[\![$ **?**op (*args*) $]\!]_{\text{TBind}}\, x\, u$ = for op∈{ **and**, **or**, **not** } **if** m=' *ff*' **then** ('*tt*', tws) **else** ('*ff*', tws) (13)
where (m, tws) = $[\![$ op(*args*) $]\!]_{\text{TBind}}\, x\, u$

$[\![$ */test* [**fun**(d_1,p_1,s_1){e_1}]…[**fun**(d_n,p_n,s_n){e_n}]$]\!]_{\text{Step}}\, x\, u$ = ns_{n+1} (14)
where ns_1 = *call*('child', *x, test*);ns_{i+1} = *filter*(ns_i, λd_i. λp_i. λs_i. $[\![e_i]\!]_{\text{Exp}}\, u$) for i=1,…,n

$[\![$ *//test* [**fun**(d_1,p_1,s_1)e_1]…[**fun**(d_n,p_n,s_n)e_n] $]\!]_{\text{Step}}\, x\, u$ = ns_{n+1} (15)
where ns_1 = *call*('descendant-or-self', *x, test*));
ns_{i+1} = *filter*(ns_i, λd_i. λp_i. λs_i. $[\![e_i]\!]_{\text{Exp}}\, u$); for i=1,…,n

$[\![$ */@test* [**fun**(d_1,p_1,s_1)e_1]…[**fun**(d_n,p_n,s_n)e_n] $]\!]_{\text{Step}}\, x\, u$= ns_{n+1} (16)
where ns_1 = *call*('attribute', *x, test*); ns_{i+1} =*filter*(ns_i, λd_i. λp_i. λs_i..$[\![e_i]\!]_{\text{Exp}}\, u$); for i=1,…,n

$[\![$ //@test [**fun**(d_1,p_1,s_1)e_1]…[**fun**(d_n,p_n,s_n)e_n] $]\!]_{\text{Step}}\, x\, u$= ns_{n+1} (17)
where ns_0 = *call*('descendant-or-self', *x*, '*')); ns_1 = *map*(ns_0, λd.call('attribute', *d, test*))
ns_{i+1} =*filter*(ns_i, λd_i. λp_i. λs_i. .$[\![e_i]\!]_{\text{Exp}}\, u$) for i=1,…,n

$[\![e]\!]_{\text{Arg}}\, u$ =.$[\![e]\!]_{\text{Exp}}\, u$ (18)

$[\![$ **fun** (ide_1,…, ide_m) { e } $]\!]_{\text{Arg}}\, u$ = <λide_1…λide_n . e, u > (19)

In the tree pattern query semantics equation (5), it first evaluates expression *e*. If the calculation results only contain one XML node, then it matches branches and generates the WMap instance by constituting all branch variables after all branch have been matched successfully. Further, the WNode instance which represents the root of the WTree instance will be constructed. If the calculation results are an XML nodes sequence, then it processes each XML node respectively and constructs the virtual root for the WTree instance. Semantics equations (6) and (7) illustrate the query nodes evaluation which has a bind variable. It first processes query step and gets several XML nodes, further, it implement each sub-query for these XML nodes. For compulsory relationship, there must be a WNode instance, otherwise the matching is failing and return false. For optional relationship, it can bind empty list to the tree pattern branch variable. The evaluations of query nodes without bind variable are shown in semantics

equations (8) and (9). Semantics of logic nodes are listed during equations (10) to (13). Semantics equations (10), (11), (12) illustrate the three logic operation with compulsory relationship respectively, while the three logic operation with optional relationship are shown in semantics equations (13). For optional relationship, it always return true to infer the matching is successful. Semantics equations (14) to (17) are semantics of query step which contain axis operation, node test, and various kind predicates. Since the query step is a kind of operation for current XML node, it needs context to evaluate predicates. Semantics equations (18) and (19) are semantics of argument; it is used to evaluate expression and construct function closure respectively.

6. Extraction of Tree Pattren

To take advantage of tree pattern query for effectively realization of XQuery, it is inevitable to analysis the query plan and extract tree pattern from it. For save space, this paper only gives the denotational semantics of processing path expression, which is the core part of tree pattern extraction.

Extraction of tree pattern is the process of rewriting the structure information in the query expression according to certain rules as XTPL expression with the *with* clause. The essence of path expression in XQuery can be considered composed of several axis operations in Table1. Therefore the core of tree pattern extraction is rewriting axis operation expression. The semantics equation of tree pattern extraction is declared as follows:

$[\![]\!]_{Extr}$: Exp→Ide×TBind→ExEnv→Exp×Tbind

Exp and *TBind* in this semantics equation stand for XTPL expressions and the tree pattern in *with* clause respectively. *ExEnv* is the context environment which is used to store bind relationships between variables and tree pattern branch. If the result of expression rewriting applied semantics equation $[\![]\!]_{Extr}$ is a tree pattern branch variable, it will return a pair composed of branch variable name and *TBind* instance; otherwise, the target expression is the output and *TBind* is nil.

The semantic domain and equations are listed in Table 8 and 9 respectively. In Table 9, according to semantics equation (1), the variables without binding do not need to be rewritten; otherwise, this variable should be replaced with tree pattern branch variable. Semantics equation (2) processes the common comparison operations in predicates, TPQ which bind may be extended because its arguments are often axis operations over current node. Semantics equation (3) is the core part of tree pattern extraction. The tree pattern extraction from axis operation can be divided into two cases: the first is forward axis, which is used for the axis operations supported by basic tree pattern, such as PC, AD, property, etc; the second is reverse axis and other operation. For the first case, the source expression *e* in axis operation is processed first. If the result is a tree pattern branch variable (b'≠nil), then this tree pattern will be extended with new query nodes using current axis operation; otherwise, a new *TBind* instance will be constructed and extended with a new query node *n* using current axis operation. Subsequently, all predicates are processed to extend this query node. Other predicates p_i except exist predicate are added to this query node. The new constructed tree pattern will be rewritten as *with* clause in XTPL, otherwise, a new constructed tree pattern branch variable <v, $b_{\#}$> will be returned. If it belongs to the second situation, a new tree pattern will be constructed after processing source expression *e*. Further, this tree pattern will be extended by processing predicates and the predicates which not be replaced by tree

pattern will be added as constraint conditions of query node. Finally, the axis operation will be rewritten as *with* clause in XTPL.

Table 2. Semantic Domain of Tree Pattern Extraction

Semantic Domain	Interpretation
a,*p*,*x*,*y*:Exp	Expression in XTPL
*tn:*TNode	Query node
tb,*nil*:TBind	Tree pattren
υ:ExEnv=(Ide→Ide×TBind)	Extraction environment

Table 3. Semantic Equations of Tree Pattern Extraction

Semantic Equations	Number
$[\![ide]\!]_{Extr} <x, tb> \upsilon =$ **if** $\upsilon(ide) = \emptyset$ **then** $< ide,nil >$ **else if** $ide=d$ **then** $< x, tb >$ **else** $\upsilon(ide)$	(1)
$[\![\text{cmp}(arg_1, arg_2)]\!]_{Extr} <x,tb> \upsilon =$ <genExp[cmp(<e_1>, <e_2>)], *nil*> where e_1=**if** $tb_1 \neq$nil **then** genExp[getNode(<var(tb_1)>.<a_1>)] **else** a_1 e_2=**if** $tb_2 \neq$nil **then** genExp[getNode(<var(tb_2)>.<a_2>) **else** a_2 $<a_1, tb_1>= [\![arg_1]\!]_{Extr} < x,tb > \upsilon$ $<a_2, tb_2>= [\![arg_2]\!]_{Extr} <x,tb_1> \upsilon$	(2)
$[\![axis (e, test [e_1]\ldots[e_n])]\!]_{Extr} <x,tb> \upsilon=$ **if** axis∈{child, descendant-or-self, attribute} then **if** $tb' \neq nil$ **then** $<y, tb^{\#}>$ **else** <genExp[<*y*>with<$tb^{\#}$>], *nil*> where $<e_0, tb'> = [\![e]\!]_{Extr} <x, tb> \upsilon$ $tb'' =$ **if** $tb' \neq nil$ **then** tb' **else** genTBind[<newVar( )>=<e_0>] y = newVar( ) $<e_i, tb_i> = [\![e_i]\!]_{Extr} <y,tb''> \upsilon$ for i=1,…,n tn = newTNode(y, *axis*, *test*) p_i = **if** $tb_i \neq nil$ **then** ∅ **else** genPred[e_i] tn' = addPred(tn, $p_1 \ldots p_n$) $tb^{\#}$ = addTNode(tb'', y, tn') **else** < genExp[<*a*> with <tb_n>], nil > where $<e_0, tb'> = [\![exp]\!]_{Extr} <x,tb> \upsilon$ e' = **if** tb'≠nil **then** genExp[<var(tb).<e_0>] **else** e_0 y = newVar( ) tb_0 = newTBind(y, e') $<e_i, tb_i> = [\![exp_i]\!]_{Extr} <y,tb_0> \upsilon$ for i=1,…,n p_i =**if** $tb_i \neq nil$ **then** ∅ **else** genPred[fun(d,p,s)e_i] a = genExp[axis(<y>, <*test*><p_1>…<p_n>)]	(3)

Auxiliary Functions *genExp* represent generating expression based on template in which its arguments are specified with symbol '< >', and *genTBind* represent generating tree pattern with representation of *with* clause. Function *newTNode* generates query node. Function *addTNode* add sub-query node with its bind variable to the given query node. Function *addPred* add predicates to the given query node. Function *genPred* construct a representation of predicates in *with* clause.

7. Conclusion

For extended XML tree pattern GTP++, which covers optional relationship, logical operators AND, OR, NOT, wildcard and various predicates, we develop a tree pattern description language XTPL, and give its denotational semantics to present the behavioral characteristics of tree pattern query. Meanwhile, in order to explain the tree

pattern extraction process, we present the semantics equations of rewriting rules for extracting tree pattern from path expressions.

With the increasingly widespread application of XML data and the development of XQuery, efficient processing of XML data need to be supported in many fields. This paper focuses on XML tree pattern description language and its formalization. However, tree pattern query is only a subset of XQuery. To support complete XQuery query, the tree pattern query results are still need to be calculated with the help of other expressions. The future work aims to develop a complete intermediate language which contains XTPL and XML query algebra, and tree pattern extraction rules, so that we can effectively organize XQuery queries into a query plan consisted of tree pattern query and query algebra. Meanwhile, we will study various optimization techniques and provide a complete solution for XQuery queries.

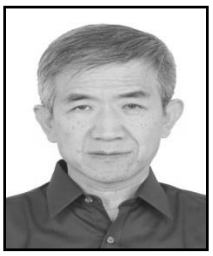
Acknowledgements

This work was both supported in part by the Beijing Nature Science Foundation under Grant 4122011and the National Science Foundation for Young Scientists of China under Grant 61202074.

References

[1] M. Hachicha and J. Darmont, "IEEE Trans", Knowl. Data Eng., vol. 25, **(2013)**
[2] Y. W. Qu, Editor, "Formal semantics: Foundation and formal specification", Science Press, Beijing, **(2010)**.
[3] X. B. Zhang and H. S. Liao, "Front. Comput. Sci. Technol.", vol. 4, **(2010)**
[4] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava and Keith *et. al*., "TAX: A Tree Algebra for XML", Database Programming Languages, Springer Berlin Heidelberg, (**2001**), pp. 149-164.
[5] N. Bruno, N. Koudas and D. Srivastava, "Holistic twig joins: Optimal XML pattern matching", Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin**, (2002)** June 3-6.
[6] J. Lu, T. Chen and T. W. Ling, "TJFast: Effective processing of XML twig pattern matching", Proceedings of the 14th International Conference of World Wide Web, Chiba, Japan, **(2005)** May 10-14, pp. 455-466.
[7] S. Chen, H. Li, J. Tatemura, W. Hsiung, D. Agrawal, *et.al.*, "Twig^{2}Stack:bottom-up processing of generalized-tree-pattern queries over XML documents", Proceedings of the 32nd International Conference of Very Large Databases, **(2006)** September 12-15, Seoul, Korea.
[8] L. Qin, J. X. Yu and B. Ding, "TwigList: Make twig pattern matching fast", Advances in Databases: Concepts, Systems and Applications, Springer Berlin Heidelberg, vol. 4443, **(2007),** pp.850-862.
[9] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan and S. Paparizos, "From tree patterns to generalized tree patterns: on efficient evaluation of XQuery", Proceedings of the 29th International Conference of Very Large Databases, Berlin, Germany**, (2003)** September 9-12.
[10] J. H. Lu, T. W. Ling, Z. F. Bao and C. Wang, "IEEE Trans.,Knowl. Data Eng.", vol. 23, **(2011)**.
[11] N. S. Alghamdi, W. Rahayu and E. Pardede, "Object-Based Semantic Partitioning for XML Twig Query Optimization", Proceedings of the 27th International Conference on Advanced Information Networking and Applications, Barcelona, Spain, **(2013)** March 25-28.
[12] S. K. Izadi, T. Harder and M. S. Haghjoo, "Data Knowl. Eng.", vol. 68, **(2009).**
[13] Q. Zeng, X. Jiang and Z. Hai, "Adding Logical Operators to Tree Pattern Queries on Graph Structured Data", Proceedings of the 29th International Conference of Very Large Databases, Istanbul, Turkey**, (2012)** August 27 – 31.
[14] D. Che, T. Ling and W. Hou, "EEE Trans. Knowl. Data Eng.", vol. 24, **(2012)**.
[15] C. Re, J. simeon and M. Fernandez, "A Complete and Efficient Algebraic Compiler for XQuery", Proceedings of the 22nd International Conference on Data Engineering. **(2006)** April3-8, Atlanta, GA, USA.
[16] P. Michiels, G. A. Mihaila and J. Simeon, "Put a Tree Pattern in Your Algebra. Proceedings of the 23rd Int. International Conference on Data Engineering", **(2007)** April 15-20, Istanbul, Turkey.

Authors



Husheng Liao was born in Changchun in 1954. He is a professor and doctoral supervisor at Beijing University of Technology in P.R.China. His research interests include software automation methods and data integration technology, etc.

Xiaoqing Li was born in Tangshan in 1983. She is a Ph.D. candidate at Beijing University of Technology in P.R.China. Her research interest is XML database technology.

Hang Su was born in Shenyang in 1978. He is a lecturer of computer science at the Beijing University of Technology in P.R.China. His current interests include XML technology, query languages and program transformation.