Extending Datalog to cover XQuery[⋆]

David Bednárek

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University Prague
david.bednarek@mff.cuni.cz

Abstract. *Datalog is a traditional platform in database research and, due to its ability to comprehend recursion, it seems to be a good choice for modeling XQuery. Unfortunately, XQuery functions have arguments carrying sequences; therefore, logic-based models of XQuery must be second-order languages and, consequently, Datalog is usually extended by node-set variables. In this paper, we suggest an alternative approach - extending Datalog by allowing structured variables in a form similar to Dewey numbers. This extension is then used to model the behavior of a XQuery program as a whole, using predicates that reflect the semantics of XQuery functions only in the context of the given program. This fact distinguishes our approach from traditional models that strive to comprehend the behavior of a function independently of its context. The advantage of this approach is that it uses the same means to model the structural recursion of documents and the functional recursion of programs, allowing various modes of bulk processing, loop reversal and other optimization techniques.*

1 Introduction

Contemporary XQuery processing and optimization techniques are usually focused on querying and, in most cases, ignore the existence of user-defined functions. In the era of XSLT 1.0, the implementation techniques had to recognize user-defined functions (templates) well (see for instance [3]); however, this branch of research appears discontinued as the community shifted to XQuery. The recent development in the area of query languages for XML shows that the XQuery language will likely be used as one of the main application development languages in the XML world [1]. In particular, intensive use of user-defined functions may be expected.

There were several attempts to apply Datalog or Datalog-like models to XPath or XQuery. There are also top-down approaches using structural recursion, i.e. strongly syntactically limited form of Horn clauses with function symbols [7]. More general forms, using first-order logic, were also used in the area of XML constraints [8].

In this paper, we (informally) define the language BTLog as an extension of Datalog. In the Section 3, an abstraction of an XQuery program as a forest is defined. In the fourth section, the principles of the transformation to BTLog is defined and shown on an example. In the Section 5, a detailed representation of the most important XQuery core operators is given.

2 BTLog

Traditional Datalog program is a set of rules in the form of Horn clauses without function symbols. We will extend this definition with one binary function symbol T, corresponding to the creation of a binary tree $\mathsf{T}(x, y)$ from subtrees x and y. We will call this language *BTLog* (from *binary-tree* logic). Of course, addition of a function symbol raises the power of the language quite dramatically; therefore, some properties of Datalog are lost and new problems are raised:

- *Termination* – using the T-operator, any number of values may be generated. Therefore, termination is not generally guaranteed and any BTLog evaluation algorithm shall cope with termination problems.
- *Minimal model semantics* – without negation, minimal model semantics works well with BTLog, just as it works with Datalog without negation.
- *Stratification* – In Datalog$^{\neg}$, stratification is used to extend the notion of minimal model. Similar definition may be used in BTLog, resulting in the language BTLog$^{\neg,\mathrm{strat}}$.
- *Non-stratifiable program semantics* – *stable model* semantics is used for non-stratified BTLog$^{\neg}$ programs.

Definition of the abovementioned terms and detailed discussion of theoretical properties of such a language may be found for instance in [5].

3 Abstraction of a XQuery program

Similarly to the normative definition of the XQuery semantics, we use (abstract) grammar rules of the *core grammar* [9] as the base for the models. A XQuery program is formalized as a forest of abstract syntax trees (AST), one tree for each user-defined function and one for the main expression. Each node of each AST, i.e. each sub-expression appearing in the program, has a (program-wide) unique *address* E. These addresses will participate as subscripts in BTLog predicate names and they will also appear as constants in some rules.

[⋆] Project of the program "Information society" of the Thematic program II of the National research program of the Czech Republic, No. 1ET100300419

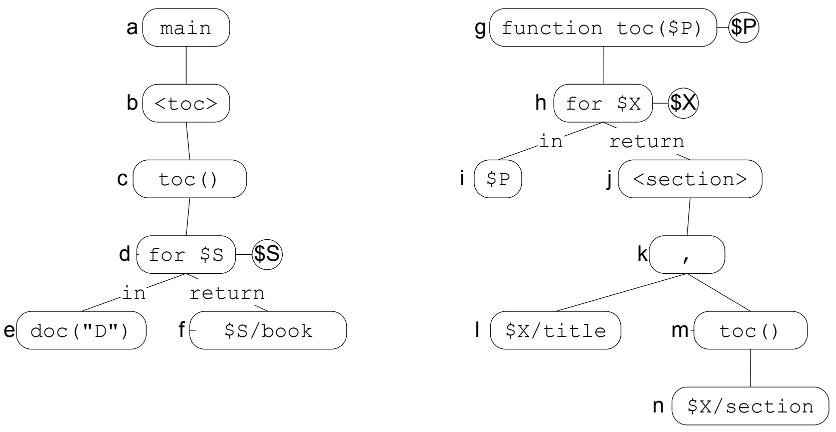

Fig. 2. Query 1 – Forest model.

```
declare function toc( $P)
{ for $X in $P return <section> {
   $X/title , toc( $X/section) } </section>
};

<toc> {
  toc( for $S in doc("D") return $S/book)
} </toc>
```

Fig. 1. Query 1.

Fig. 2 shows the abstract syntax forest corresponding to the Query 1 at Fig. 1. Node adresses are shown as letters left to the nodes.

For each AST node E, the set $\mathsf{vars}[E]$ contains the names of *accessible variables*. In particular, when E is the root of a function AST, $\mathsf{vars}[E]$ contains the names of arguments of the function, including implicit arguments like the context node.

4 Principles of the transformation

The model is based on the following principles:

– Nodes within a tree are identified by *node identifiers* using *Dewey ID* labeling scheme. (See, for instance [6].)
– A tree is encoded using a mapping of Dewey labels to *node properties*.
– A tree created during XQuery evaluation is identified by a *tree identifier* derived from the context in which the tree was constructed.
– A node is globally identified by the pair of a tree identifier and a node identifier.
– A sequence (i.e. any XQuery expression value) is modeled using a mapping of *sequence identifiers* to *sequence items*. Since a sequence may mix atomic values and document nodes, the mapping is divided into two interweaved *lists*.
– Each *sequence* containing document nodes is accompanied by a *tree environment* which contains the encoding of the document trees to which the nodes of the sequence belong.
– Evaluating a `for`-expression corresponds to iteration through all sequence identifiers in the value of the `in`-clause.
– A particular context reached during XQuery evaluation is identified by the pair of a *call stack*, containing positions in the program code, and a *control variable stack*, containing sequence identifiers selected by the `for`-expressions along the call stack.
– Node identifiers, tree identifiers, sequence identifiers, and control variable stacks share the same domain of binary trees with values on leaves, allowing to construct each kind of identifier from the others. In most cases, a binary tree is used to encode a (generalized) string – then, the rightmost path in the tree has the length of the string and the children of the rightmost path are the letters of the string.

4.1 Model predicates

Our model assigns a set of predicates to each AST node, i.e. to each address E:

– *Invocation* $\mathsf{inv}_E(i,f)$ enumerates the contexts in which the expression E is evaluated. Argument i represents the call stack that brought the execution to the examined expression E. f is the stack of sequence identifiers selected by the `for`-clauses throughout the descent along i to E. The two stacks together form the identification of the dynamic context in which an expression is evaluated. While the XQuery standard defines dynamic context as the set of variable assignments (with some negligible additions), our notion of dynamic context is based on the stack pair that determines the descent through the code to the examined expression, combining both the code path stored in i and the `for`-control variables in f. The key to the sufficiency of this model is the observation that the variable assignment is a function of the stack pair.

– *Atomic list* $\mathsf{alst}_E(i,f,s,v)$ represents the atomic value portion of the assignment of the result value of the expression E to the contexts enumerate by $\mathsf{inv}_E(i,f)$. s is a sequence identifier, v is a value of an atomic type as defined by the XQuery standard. The predicate $\mathsf{alst}_E(i,f,s,v)$ is true if the value of the expression E in the context (i,f) contains the atomic value v at position s.
– *Node list* $\mathsf{nlst}_E(i,f,s,t,n)$ represents the node portion of the result value of the expression E. The meaning of i, f, and s is the same as in alst_E. t is a tree identifier – for external documents, it is a literal value, for temporal trees, it is the expression $\mathsf{T}(i_1,f_1)$ corresponding to the environment identification at the moment of tree creation. n is a node identifier in the form of a Dewey ID.
– *Environment* $\mathsf{env}_E(i,t,n,a)$ represents the tree environment associated to the result value of the expression E. i determines the call context (note that the environment is independent of the control variable stack). t is a tree identifier, n is a node identifier, a is a tuple of properties assigned to a node by the XML Data Model, containing node kind, name, typed and string values, etc. Particular properties are accessed using predicates $\mathsf{name}(a,v)$, $\mathsf{string}(a,v)$, etc.
– $\mathsf{valst}_{E,\$\mathsf{X}}(i,f,s,v)$, $\mathsf{vnlst}_{E,\$\mathsf{X}}(i,f,s,t,n)$, and $\mathsf{venv}_{E,\$\mathsf{X}}(i,t,n,a)$ represent the assignment of the values of the variable $\$\mathtt{x} \in \mathsf{vars}[E]$ to the contexts satifying $\mathsf{inv}_E(i,f)$. The meaning of the arguments is the same as in alst_E, nlst_E, and env_E.

4.2 Example

The following example shows the Query 1 transformed to a BTLog program. The subscripts in the predicate names correspond to the adresses shown in Fig. 2; unused and identity rules were removed. The main expression of the Query 1 transforms to the following BTLog rules:

$\mathsf{inv_a}(1,1).$ -- program start

$\mathsf{env_e}(i,D,n,a) \text{ :– } \mathsf{inv_e}(i,f), \mathsf{doc}(\text{"}D\text{"},n,a).$
-- doc("D") tree environment

$\mathsf{vnlst}_{\mathsf{f},\$\mathsf{S}}(i,\mathsf{T}(s,f),1,t,n) \text{ :– } \mathsf{nlst_a}(i,f,s,t,n).$
-- variable $S

$\mathsf{nlst_f}(i,f,\mathsf{T}(t,n),t,n) \text{ :– } \mathsf{vnlst}_{\mathsf{f},\$\mathsf{S}}(i,f,s,t,m),$
$\quad \mathsf{env_e}(i,t,n,\mathsf{T}(\mathsf{element},\mathsf{book})),$
$\quad \mathsf{child}(m,n).$
-- `$S/book` node

$\mathsf{nlst_d}(i,f,\mathsf{T}(s,r),t,n) \text{ :– } \mathsf{nlst_f}(i,\mathsf{T}(s,f),r,t,n).$
-- the result of the for-expression

$\mathsf{vnlst}_{\mathsf{g},\$\mathsf{P}}(\mathsf{T}(\mathsf{c},i),f,s,t,n) \text{ :– } \mathsf{nlst_d}(i,f,s,t,n).$
-- argument `$P` in `toc`

$\mathsf{venv}_{\mathsf{g},\$\mathsf{P}}(\mathsf{T}(\mathsf{c},i),t,n,a) \text{ :– } \mathsf{env_e}(i,t,n,a).$
-- environment of `$P` in `toc`

$\mathsf{nlst_c}(i,f,s,t,n) \text{ :– } \mathsf{nlst_g}(\mathsf{T}(\mathsf{c},i),f,s,t,n).$
-- the return value of `toc`

$\mathsf{env_c}(i,t,n,a) \text{ :– } \mathsf{env_g}(\mathsf{T}(\mathsf{c},i),t,n,a).$
-- the return value environment

$\mathsf{nlst_b}(i,f,1,\mathsf{T}(i,f),1) \text{ :– } \mathsf{inv_a}(i,f).$
-- the `<toc>` node

$\mathsf{env_b}(i,\mathsf{T}(i,f),1,\mathsf{T}(\mathsf{element},\mathsf{toc})) \text{ :– } \mathsf{inv_a}(i,f).$

$\mathsf{env_b}(i,\mathsf{T}(i,f),\mathsf{T}(s,p),a) \text{ :– } \mathsf{nlst_c}(i,f,s,t,m),$
$\quad \mathsf{env_c}(i,t,n,a), \mathsf{cat}(n,m,p).$
-- the `<toc>` node environment

$\mathsf{out}(i,t,n,a) \text{ :– } \mathsf{env_b}(i,t,n,a).$
-- the output tree

The following rules correspond to the function `toc`:

$\mathsf{inv_j}(i,\mathsf{T}(s,f)) \text{ :– } \mathsf{vnlst}_{\mathsf{g},\$\mathsf{P}}(i,f,s,t,n).$
-- the invocation of the return clause

$\mathsf{vnlst}_{\mathsf{j},\$\mathsf{X}}(i,\mathsf{T}(s,f),1,t,n) \text{ :– } \mathsf{vnlst}_{\mathsf{g},\$\mathsf{P}}(i,f,s,t,n).$
-- variable `$X`

$\mathsf{nlst_l}(i,f,\mathsf{T}(t,n),t,n) \text{ :– } \mathsf{vnlst}_{\mathsf{j},\$\mathsf{X}}(i,f,s,t,m),$
$\quad \mathsf{venv}_{\mathsf{g},\$\mathsf{P}}(i,t,n,\mathsf{T}(\mathsf{element},\mathsf{title})),$
$\quad \mathsf{child}(m,n).$
-- `$X/title` expression

$\mathsf{nlst_n}(i,f,\mathsf{T}(t,n),t,n) \text{ :– } \mathsf{vnlst}_{\mathsf{j},\$\mathsf{X}}(i,f,s,t,m),$
$\quad \mathsf{venv}_{\mathsf{g},\$\mathsf{P}}(i,t,n,\mathsf{T}(\mathsf{element},\mathsf{section})),$
$\quad \mathsf{child}(m,n).$
-- `$X/section` expression

$\mathsf{vnlst}_{\mathsf{g},\$\mathsf{P}}(\mathsf{T}(\mathsf{m},i),f,s,t,n) \text{ :– } \mathsf{nlst_n}(i,f,s,t,n).$
-- argument `$P` in `toc`

$\mathsf{venv}_{\mathsf{g},\$\mathsf{P}}(\mathsf{T}(\mathsf{m},i),t,n,a) \text{ :– } \mathsf{venv}_{\mathsf{g},\$\mathsf{P}}(i,t,n,a).$
-- environment of `$P` in `toc`

$\mathsf{nlst_m}(i,f,s,t,n) \text{ :– } \mathsf{nlst_g}(\mathsf{T}(\mathsf{m},i),f,s,t,n).$
-- the return value of `toc`

$\mathsf{env_m}(i,t,n,a) \text{ :– } \mathsf{env_g}(\mathsf{T}(\mathsf{m},i),t,n,a).$
-- the return value environment

$\mathsf{nlst_k}(i,f,\mathsf{T}(1,s),t,n) \text{ :– } \mathsf{nlst_l}(i,f,s,t,n).$

$\mathsf{nlst_k}(i,f,\mathsf{T}(2,s),t,n) \text{ :– } \mathsf{nlst_m}(i,f,s,t,n).$
-- the concatenated value

$\mathsf{env_k}(i,t,n,a) \text{ :– } \mathsf{venv}_{\mathsf{g},\$\mathsf{P}}(i,t,n,a).$

$\mathsf{env_k}(i,t,n,a) \text{ :– } \mathsf{env_m}(i,t,n,a).$

-- the environment of the concatenation

$\mathsf{nlst_j}(i, f, 1, \mathsf{T}(i, f), 1) \coloneq \mathsf{inv_j}(i, f).$

-- the `<section>` node

$\mathsf{env_j}(i, \mathsf{T}(i, f), 1, \mathsf{T}(\mathsf{element}, \mathsf{toc})) \coloneq \mathsf{inv_j}(i, f).$

$\mathsf{env_j}(i, \mathsf{T}(i, f), \mathsf{T}(s, p), a) \coloneq \mathsf{nlst_k}(i, f, s, t, m), \mathsf{env_k}(i, t, n, a), \mathsf{cat}(n, m, p).$

-- the `<section>` node environment

$\mathsf{nlst_h}(i, f, \mathsf{T}(s, r), t, n) \coloneq \mathsf{nlst_j}(i, \mathsf{T}(s, f), r, t, n).$

-- the result of the for-expression

$\mathsf{nlst_g}(i, f, s, t, n) \coloneq \mathsf{nlst_h}(i, f, s, t, n).$

-- the result of the function

$\mathsf{env_g}(i, t, n, a) \coloneq \mathsf{env_j}(i, t, n, a).$

-- the result environment

Figure 3 show the dependence graph for the predicates of Query 1. There are three strongly connected components (shown in bold) – the first one carries the environment of argument `$P` (i.e. the input document) down through the recursion of the function `toc`. The second component corresponds to the recursive descent of the variable `$P` through the document. The third component collects the constructed nodes back, unwinding the call stack.

5 Representation of Core XQuery Operators

There are several variants of *core* subsets of XQuery, including the *core grammar* defined in the W3C standard [9], the LixQuery framework [4], and others [2]. Since the XSLT and XQuery are related languages and the translation from XSLT to XQuery is known (see [2]), the model may be applied also to XSLT.

Note: Most XQuery operators do not change the assignment of variable values; therefore, we will omit the propagation rules in the subsequent description. We will also omit the rules for alst_E and $\mathsf{valst}_{E,\$x}$ whenever they are similar to nlst_E and $\mathsf{vnlst}_{E,\$x}$.

Function call – E_0 = `f(` E_1 `)`

Assume that E_f is the root of the function and `$x` is the name of the formal argument. The rules implement pushing the call address E_0 onto the call stack and popping it back upon return.

$$\mathsf{inv}_{E_f}(\mathsf{T}(E_0, i), f) \coloneq \mathsf{inv}_{E_0}(i, f).$$
$$\mathsf{vnlst}_{E_f,\$x}(\mathsf{T}(E_0, i), f, s, t, n) \coloneq \mathsf{nlst}_{E_1}(i, f, s, t, n).$$
$$\mathsf{venv}_{E_f,\$x}(\mathsf{T}(E_0, i), t, n, a) \coloneq \mathsf{env}_{E_1}(i, t, n, a).$$
$$\mathsf{nlst}_{E_0}(i, f, s, t, n) \coloneq \mathsf{nlst}_{E_f}(\mathsf{T}(E_0, i), f, s, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \coloneq \mathsf{env}_{E_f}(\mathsf{T}(E_0, i), t, n, a).$$

For Expression – E_0 = `for $y in` E_{in} `return` E_{ret}

The for-expression generates a new dynamic context for each member of the sequence E_{in}; in our model, it is represented by pushing the sequence identifier s onto the control stack f:

$$\mathsf{inv}_{E_{\mathsf{ret}}}(i, \mathsf{T}(s, f)) \coloneq \mathsf{nlst}_{E_{\mathsf{in}}}(i, f, s, t, m).$$

At the same time, the variable `$y` is added to the dynamic context, defined as follows:

$$\mathsf{vnlst}_{E_{\mathsf{ret}},\$y}(i, \mathsf{T}(s, f), \mathsf{one}, t, n) \coloneq \mathsf{nlst}_{E_{\mathsf{in}}}(i, f, s, t, n).$$
$$\mathsf{venv}_{E_{\mathsf{ret}},\$y}(i, t, m, a) \coloneq \mathsf{env}_{E_{\mathsf{in}}}(i, t, m, a).$$

For older variables, the following rules are defined for each $\$x \in \mathsf{vars}[E_0] \setminus \{\$y\}$:

$$\mathsf{vnlst}_{E_{\mathsf{ret}},\$x}(i, \mathsf{T}(s, f), r, u, m) \coloneq \mathsf{nlst}_{E_{\mathsf{in}}}(i, f, s, t, n), \mathsf{vnlst}_{E_0,\$x}(i, f, r, u, m).$$
$$\mathsf{venv}_{E_{\mathsf{ret}},\$x}(i, u, m, a) \coloneq \mathsf{venv}_{E_0,\$x}(i, u, m, a).$$

Finally, the value of the for-expression is created by the concatenation of the return clause values:

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(s, r), t, n) \coloneq \mathsf{nlst}_{E_{\mathsf{ret}}}(i, \mathsf{T}(s, f), r, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \coloneq \mathsf{env}_{E_{\mathsf{ret}}}(i, t, n, a).$$

Let Expression – E_0 = `let $y :=` E_{def} `return` E_{ret}

The let-expression adds the variable `$y` to the dynamic context. Nevertheless, the identification of the context is not changed and the other variables are also preserved.

$$\mathsf{inv}_{E_{\mathsf{ret}}}(i, f) \coloneq \mathsf{inv}_{E_0}(i, f).$$
$$\mathsf{vnlst}_{E_{\mathsf{ret}},\$y}(i, f, s, t, n) \coloneq \mathsf{nlst}_{E_{\mathsf{def}}}(i, f, s, t, n).$$
$$\mathsf{venv}_{E_{\mathsf{ret}},\$y}(i, t, m, a) \coloneq \mathsf{env}_{E_{\mathsf{def}}}(i, t, m, a).$$

Where Clause – E_0 = `for $y in` E_{in} `where` E_{wh} `return` E_{ret}

Adding where clause to a for-expression affects the set of contexts generated for the return clause; similarly, variable models are affected:

$$\mathsf{inv}_{E_{\mathsf{ret}}}(i, \mathsf{T}(s, f)) \coloneq \mathsf{alst}_{E_{\mathsf{wh}}}(i, f, s, \mathsf{true}).$$
$$\mathsf{vnlst}_{E_{\mathsf{ret}},\$y}(i, \mathsf{T}(s, f), \mathsf{one}, t, n) \coloneq \mathsf{nlst}_{E_{\mathsf{in}}}(i, f, s, t, n), \mathsf{alst}_{E_{\mathsf{wh}}}(i, f, s, \mathsf{true}).$$
$$\mathsf{vnlst}_{E_{\mathsf{ret}},\$x}(i, \mathsf{T}(s, f), r, u, m) \coloneq \mathsf{nlst}_{E_{\mathsf{in}}}(i, f, s, t, n), \mathsf{alst}_{E_{\mathsf{wh}}}(i, f, s, \mathsf{true}), \mathsf{vnlst}_{E_0,\$x}(i, f, r, u, m).$$

Equality Test – E_0 = E_1 `eq` E_2

$$\mathsf{eq}_{E_0}(i, f) \coloneq \mathsf{alst}_{E_1}(i, f, s, v), \mathsf{alst}_{E_2}(i, f, r, v).$$
$$\mathsf{alst}_{E_0}(i, f, 1, \mathsf{true}) \coloneq \mathsf{eq}_{E_0}(i, f).$$
$$\mathsf{alst}_{E_0}(i, f, 1, \mathsf{false}) \coloneq \neg\mathsf{eq}_{E_0}(i, f).$$

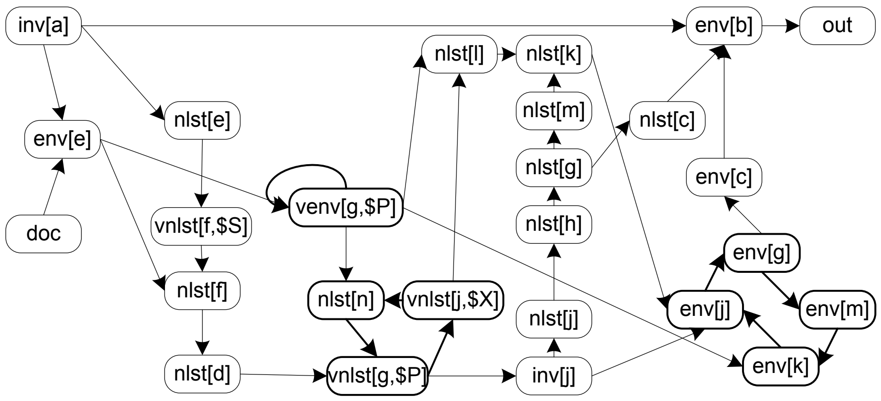

Fig. 3. Query 1 – Predicate dependence graph.

Node Construction – E_0 = `<A>{` E_1 `}</A>`

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{one}, \mathsf{T}(i, f), \mathsf{one}) \coloneq \mathsf{inv}_{E_0}(i, f).$$
$$\mathsf{env}_{E_0}(i, \mathsf{T}(i, f), \mathsf{one}, a) \coloneq \mathsf{inv}_{E_0}(i, f), \mathsf{element}_{\mathtt{A}}(a).$$
$$\mathsf{env}_{E_0}(i, \mathsf{T}(i, f), \mathsf{T}(s, p), a) \coloneq \mathsf{nlst}_{E_1}(i, f, s, t, m), \mathsf{env}_{E_1}(i, t, n, a), \mathsf{cat}(n, m, p).$$

The auxiliary predicate cat corresponds to the concatenation of Dewey identifiers $n = m.p$ and it is defined as follows:

$$\mathsf{cat}(p, \mathsf{one}, p).$$
$$\mathsf{cat}(\mathsf{T}(s, n), \mathsf{T}(s, m), p) \coloneq \mathsf{cat}(n, m, p).$$

Navigation – E_0 = E_1 `/` *axis*`::*`

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \coloneq \mathsf{nlst}_{E_1}(i, f, s, t, m), \mathsf{env}_{E_1}(i, t, n, a), axis(m, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \coloneq \mathsf{env}_{E_1}(i, t, n, a).$$

The selection operator is driven by a predicate $axis$ corresponding to the *axis* used in the navigation operator. These predicates are defined as follows:

$$\mathsf{child}(\mathsf{one}, \mathsf{T}(s, \mathsf{one})).$$
$$\mathsf{child}(\mathsf{T}(s, m), \mathsf{T}(s, n)) \coloneq \mathsf{child}(m, n).$$
$$\mathsf{parent}(m, n) \coloneq \mathsf{child}(n, m).$$
$$\mathsf{descendant}(\mathsf{one}, \mathsf{T}(s, n)).$$
$$\mathsf{descendant}(\mathsf{T}(s, m), \mathsf{T}(s, n)) \coloneq \mathsf{descendant}(m, n).$$
$$\mathsf{ancestor}(m, n) \coloneq \mathsf{descendant}(n, m).$$
$$\mathsf{descendantorself}(m, n) \coloneq \mathsf{cat}(n, m, p).$$
$$\mathsf{ancestororself}(m, n) \coloneq \mathsf{descendantorself}(n, m).$$

Node-Set Union – E_0 = E_1 `union` E_2

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \coloneq \mathsf{nlst}_{E_1}(i, f, s, t, n).$$
$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \coloneq \mathsf{nlst}_{E_2}(i, f, s, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \coloneq \mathsf{env}_{E_1}(i, t, n, a).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \coloneq \mathsf{env}_{E_2}(i, t, n, a).$$

Note that the sequence identifiers s are not referenced at the head of the rule; instead, the identifier $\mathsf{T}(t, n)$ representing document order is used.

Node-Set Intersection – E_0 = E_1 `intersection` E_2

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \coloneq \mathsf{nlst}_{E_1}(i, f, r, t, n), \mathsf{nlst}_{E_2}(i, f, s, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \coloneq \mathsf{env}_{E_1}(i, t, n, a), \mathsf{env}_{E_2}(i, t, n, a).$$

Node-Set Difference – E_0 = E_1 `except` E_2

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(t, n), t, n) \coloneq \mathsf{nlst}_{E_1}(i, f, r, t, n), \neg\mathsf{nlst}_{E_2}(i, f, s, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \coloneq \mathsf{env}_{E_1}(i, t, n, a).$$

Concatenation – E_0 = E_1 `,` E_2

$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(\mathsf{one}, s), t, n) \coloneq \mathsf{nlst}_{E_1}(i, f, s, t, n).$$
$$\mathsf{nlst}_{E_0}(i, f, \mathsf{T}(\mathsf{two}, s), t, n) \coloneq \mathsf{nlst}_{E_2}(i, f, s, t, n).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \coloneq \mathsf{env}_{E_1}(i, t, n, a).$$
$$\mathsf{env}_{E_0}(i, t, n, a) \coloneq \mathsf{env}_{E_2}(i, t, n, a).$$

Note that whenever the two environments env_{E_1} and env_{E_2} contain the same tree identifier t, the corresponding tree information is merged by the $\mathtt{env}_{E_0}$ rules. Since the tree identifier exactly determines the context in which the tree was created, trees having the same identifier must be identical; therefore, merging the tree environments do not alter them anyway.

6 Conclusion

We have presented a model of XQuery evaluation based on Horn clauses under the BTLog$^\neg$ syntax. From the syntactic point of view, this model comprehends the following XQuery structures: Declare function, function call, for-clause, let-clause, where-clause, stable-order-by-clause, quantified expressions, equality operator on atomic values, Boolean operators including negation, `union`, `intersection`, `except` operators, concatenation (`,`) operator, statically named document references (`fn:doc`), forward/reverse axis navigation, name tests, `fn:root`, and element constructors.

There are two important omissions from the core XQuery that are not covered by this model: Positional variables and aggregate functions. There are also some flaws in error handling, namely the fact that the model may silently process some situations that shall be signalled as an error. These issues will be addressed by our future research.

The universal quantified expression (`every`), the equality operator, and the node-set subtraction operator (`except`) involve negation in their BTLog rules. Some XQuery programs are not stratifiable after conversion to BTLog$^\neg$. This is not necessarily a weakness of the approach – since the XQuery language is Turing-complete, we shall not expect general stratifiability. This way, the stratifiability of its BTLog$^\neg$ equivalent may be used as a borderline between “easy” and “difficult” cases. Fortunately, it shows that most of the real-life XQuery programs fall in the “easy” stratifiable category – for instance, all the XML Query Use Cases [10] programs are stratifiable.

Since termination in XQuery is not guaranteed, it may be expected that it is not generally guaranteed also in BTLog. Our future research will focus on static analysis methods trying to discover a termination guarantee in a BTLog program (of course, due to the Turing-completeness, no method can decide on the existence of a termination guarantee).

References

1. Chamberlin, D.: XQuery: Where Do We Go from Here? In: XIMEP 2006, 3rd International Workshop on XQuery Implementation, Experiences and Perspectives. ACM Digital Library, New York (2006)
2. Fokoue, A., Rose, K., Siméon, J., Villard, L.: Compiling XSLT 2.0 into XQuery 1.0. In: WWW ’05: Proceedings of the 14th International Conference on World Wide Web, pp. 682–691, ACM, New York (2005)
3. Groppe, S., Böttcher, S., Birkenheuer, G., Höing, A.: Reformulating XPath Queries and XSLT Queries on XSLT Views. Technical report, University of Paderborn (2006)
4. Hidders, J., Michiels, P., Paredaens, J., Vercammen, R.: LixQuery: A Formal Foundation for XQuery Research. SIGMOD Rec., 34(4):21–26, ACM, New York (2005)
5. Hinrichs, T., Genesereth, M.: Herbrand Logic. Technical report LG-2006-02, Stanford University (2006)
6. Lu, J., Ling, T.W., Chan, C.-Y., Chen, T.: From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In: VLDB ’05: Proceedings of the 31st International Conference on Very Large Data Bases, pp. 193–204. ACM, New York (2005)
7. Buneman, P., Fernandez, M., Suciu, D.: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion In: The VLDB Journal, pp. 76-110, Springer-Verlag (2000)
8. Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-Variable Logic on Data Trees and XML Reasoning In: PODS’06, ACM, New York (2006)
9. XQuery 1.0 and XPath 2.0 Formal Semantics, W3C (2007)
10. XML Query Use Cases, W3C (2007), `http://www.w3.org/TR/xquery-use-cases/`