How to Recognise Different Kinds of Tree Patterns From Quite a Long Way Away

Jan Hidders
University of Antwerp

Philippe Michiels
University of Antwerp

Jérôme Siméon
IBM Research

Roel Vercammen
University of Antwerp

ABSTRACT

Tree patterns are one of the main abstractions used to access XML data. Tree patterns are used, for instance, to define XML indexes, and support a number of efficient evaluation algorithms. Unfortunately deciding whether a particular query, or query fragment, is a tree pattern is undecidable for most XML Query languages. In this paper, we identify a subset of XQuery for which the problem is decidable. We then develop a sound and complete algorithm to recognize the corresponding tree patterns for that XQuery subset. The algorithm relies on a normal form along with a set of rewriting rules that we show to be strongly normalizing. The rules have been implemented and result in a normal form which is suitable for compiling tree patterns into an appropriate XML algebra.

1. INTRODUCTION

Recognizing trees can be a difficult task. Some trees look like shrubberies, while some shrubberies, seen from afar, look like trees. Recognizing tree patterns in full-fledge XML Query languages is even harder, and notably, it is undecidable for the whole XQuery language. Tree patterns are used extensively as a representation for accessing XML data. For instance, numerous efforts have focused on the development of efficient algorithms for tree patterns [2, 11, 21, 17, 6, 12, 14] and corresponding indexes [13, 5, 19, 4, 18, 20, 3]. However, current compilers typically only recognize such tree patterns when they are written as very simple XPath expressions. In this paper, we study the problem of deciding whether a query is a tree pattern or not, and if yes how to recognize which tree pattern it is. We identify a subset of XQuery for which the problem is decidable. We then develop a sound and complete algorithm to recognize the corresponding tree pattern for a given query in that subset. The algorithm relies on a normal form along with a set of rewriting rules that we show to be strongly normalizing. The rules have been implemented and result in a normal form which is suitable for compiling Tree Patterns into an appropriate XML algebra.

Tree patterns [2] have been used extensively for XML processing because they provide the right abstraction to describe access to tree-structured data. They are typically defined as simple trees whose nodes are labelled with XML names, and edges describe either child or descendant relationships. Figure 1 on the left, shows some tree patterns expressed in XQuery. Most XML Query languages do not directly support tree patterns, but usually rely on path navigation primitives based on XPath. Very simple XPath expressions, such as **Q1a** on Figure 1 look very similar to tree patterns and are easy to recognize as such. Note however that even in that case, XPath differs from a tree pattern in that it can only return nodes from one branch while other branches act as predicates (the `emailaddress` branch in our example). This work focusses on tree patterns that correspond with XPath expressions and from now on, we also refer to tree patterns as XPath expressions, i.e., having exactly one node of the pattern as output node for which the result is returned in document order and without duplicates.

Determining whether an arbitrary expression is a tree pattern is far from trivial. For instance, **Q1b** and **Q1c** which are written using a combination of Path expressions and FLWOR expressions, are equivalent to **Q1a** and therefore are tree patterns. In some cases, subtle changes in the query can affect its semantics in a way that makes it different from a tree pattern. For instance, **Q1n** is almost identical to **Q1b** but does not return the corresponding nodes in document order. As shown in [10], deciding whether a simple XPath expression returns nodes in document order or not depends on the particular combination of axes that are used in it. For instance, **Q2** returns nodes in document order because the first step uses the child axis, making sure the nodes that are the input for the second step do not have an ancestor-descendant relationship. However, that property does not hold for **Q2n** and as a result the query may not return nodes in document order and therefore is not a tree pattern. We use similar techniques to those presented in [10] for deciding the so called *ord* and *nodup* properties for a different fragment of XQuery. We also show that whether an expression yields ordered and duplicate-free results, is the deciding factor for determining whether the expression is an XPath expression, and thus can be expressed with a tree pattern.

More generally, an important requirement for a query compiler is the ability to detect fundamental access operations independently of the way the query is written. In the case that interest us, we believe all the queries **Q1** should be recognized as tree patterns, and compiled as such to the ap-

Q1a
```
$d//person[emailaddress]/name
```

Q1b
```
(for $x in
    $d//person[emailaddress]
return $x)/name
```

Q1c
```
let $x :=
    for $y in $d//person
    where $y/emailaddress
    return $y
return $x/name
```

Q2
```
for $x in $d/item[description]
return $x//listitem
```

Q1n
```
for $x in
    $d//person[emailaddress]
return $x/name
```

Q2n
```
for $x in $d//item
where $x/description
return $x//listitem
```

Figure 1: Some examples of tree patterns (left) and non-tree patterns (right).

propriate algorithms which can take advantage of available indexes. There has been very little work on trying to address that problem. Compilation techniques that take tree patterns into account, as well as corresponding rewritings and algebraic optimization rules have been proposed in [23]. While the proposed approach works on the complete languages, it is not complete. To the best of our knowledge, this paper is the first to identify a precise fragment of XQuery for which a complete algorithm exists.

The rest of this paper is organized as follows. In Section 2, we formally introduce tree patterns, and the query fragment that we consider. In Section 3, we present the algorithm used to decide whether a query in that fragment is a tree pattern. In Section 4, we show that all expressions in the considered fragment that return a result in document order and without duplicates are tree patterns and we give a set of rules to obtain the corresponding tree pattern. Finally, we discuss some related work in Section 5 and conclude the paper in Section 6.

2. PRELIMINARIES

We first define a few essential notions that are used in the rest of the paper. We then define formally the notion of tree pattern, and introduce the XQuery fragment on which our algorithm work.

2.1 Essential Notions

Before proceeding to the heart of the problem, we present the usual concepts.

XML store: (or simply called store here) is simplified to ordered sets of ordered node-labeled trees, denoted by variables $S, S', \ldots, S_1, \ldots$ et cetera.

Sub/super-store: S is called a sub-store of S' if S' can be constructed from S by adding extra edges and / or nodes.

XML value over a store, (simply called value here) simplified here to finite sequences of nodes in the store, denoted by variables $v, v', \ldots, v_1, \ldots$, and enumerated such as $\langle n_1, n_2, n_3 \rangle$. Concatenation of two values v_1 and v_2 is denoted as $v_1 \cdot v_2$.

Sub/super-value: v is a sub-value of v' if $v' = \langle n_1, \ldots, n_k \rangle$, $\{i_1, \ldots, i_j\} \subseteq \{1, \ldots, k\}$ such that $i_1 < \ldots < i_j$ and $v = \langle n_{i_1}, \ldots, n_{i_j} \rangle$.

Variable names: ($x, $y etc., including a special variable $dot) denoted by variables $\$x, \$y, \$x', \ldots, \$x_1, \ldots$

Variable assignment: over a store S, a function that maps variable names to values over S, denoted by variable $\Gamma, \Gamma', \ldots, \Gamma_1, \ldots$. The variable $dot is always mapped to a single node.

Sub/super-assignment: Variable assignment Γ is a sub-assignment of Γ' if $\Gamma(\$x)$ is a sub-value of $\Gamma'(\$x)$ for each variable $\$x$.

Axes: The axes descendant and descendant-or-self are in this paper abbreviated to desc and d-o-s, respectively.

2.2 Forest Patterns

Our work relies on a slightly extended notion of tree patterns that we call forest patterns. A forest pattern is a set of tree patterns, all of which have an input which is denoted by a variable. This last aspect makes sure that the proposed formalization can apply to any sub-expressions in the context of an arbitrary queries.

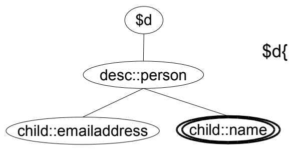
DEFINITION 2.1 (FOREST PATTERN). *A* forest pattern *is a node-labeled forest that labels root nodes with variables $\$x$ and other nodes with an axis-node test pair a::n, and in addition one node may be marked as output node such that nodes labeled with* $dot *have one child if they are not output node and no children if they are output node.*

Forest patterns with an output node are called *output patterns* and those without an output node are called *condition patterns.* Forest patterns that consist of a single tree are simply called *tree patterns.*

Although defined as graphs we will usually use a textual representation of forest patterns and their subtrees. A tree is denoted as $l\{t'_1, \ldots, t'_m\}$ where l is the label of the root and either of the form a::n or $\$x$, and $\{t'_1, \ldots, t'_m\}$ is the set of subtrees directly under the root. If the root of the tree is an output node then we add to l a superscript *out* as in $\$\mathsf{x}^{out}$. If the set of subtrees is empty then we omit it altogether. A forest pattern that consists of the trees $t_1, \ldots, t_n$ is simply denoted as $\{t_1, \ldots, t_n\}$. Since a single tree is also a forest we will identify the tree t with the forest $\{t\}$. For two forests f_1 and f_2 we denote their disjoint union as $f_1 + f_2$ which is only defined if f_1 and f_2 are not both output patterns.

An example of a tree pattern and its textual representation that correspond to the query **Q1a** in Figure 1 are given in Figure 2.

The semantics of a forest pattern are defined given a store S and variable assignment Γ over S. It is defined by *embeddings* which are functions h from the nodes of the pattern to



```
$d{  desc::person {  child::person {
                          child::emailaddress,
                          child::name^out } } }
```

Figure 2: Query Q1a as a tree pattern and its textual representation.

nodes in the store such that (1) if a node n in the pattern is labeled with variable $\$x$ then it is in $\Gamma(\$x)$ and (2) if a node is labeled with $ax{::}nt$ then (a) if n' is the parent of n in the pattern then $h(n')$ must have relationship ax with $h(n)$ in the store S, and (b) if nt is a node name then $h(n)$ must be labeled with nt in the store S. The result of a forest pattern is then defined as the sequence that (1) contains exactly all the nodes n from the store S for which there is an embedding that maps the output node to n and (2) is sorted in the document order defined by the store.

2.3 CXQ Tree Pattern Fragment

The fragment of XQuery that we consider here, is the following:

Definition 2.2 (CXQ). *A fragment of the XQuery Core language, defined by:*

$$
\begin{array}{lcl}
expr & ::= & \$x \mid axis{::}ntest \mid \mathsf{ddo}(expr) \mid \mathsf{if}\ expr\ \mathsf{then}\ expr \\
 & & \mid \mathsf{for}\ \$x\ \mathsf{in}\ expr\ \mathsf{return}\ expr \\
 & & \mid \mathsf{let}\ \$x := expr\ \mathsf{return}\ expr \\
ntest & ::= & label \mid * \\
axis & ::= & \mathsf{child} \mid \mathsf{desc} \mid \mathsf{d\text{-}o\text{-}s}
\end{array}
$$

with the restriction that in let $\$x := e_1$ return e_2 *the variable* $\$x$ *cannot be* $\$dot$.

Note that we abbreviate the expression if e_1 then e_2 else () to if e_1 then e_2 and fs:distinct-docorder to ddo.

Because this fragment is expressed in terms of the XQuery Core [8], it covers a larger fragment of the XQuery language than may seem. Notably, it is sufficient to express XPath 1.0 expressions with structural predicates (without positional predicates or comparisons), composed with FLWOR expressions. Notably, it supports all the queries used as examples in Section 1.

Finally, we will use the following basic notions for CXQ expressions, and a notion of variable substitution: $FV(e)$ denotes the set of free variables in e. It is defined as usual except that $FV(a{::}n) = \{\$\mathsf{dot}\}$. The judgment $S, \Gamma \vdash e \Rightarrow x$ where S a store, Γ a variable assignment over S, e a CXQ expression and x a value over S. Two expressions e and e' are said to be equivalent, denoted as $e \equiv e'$, if for every store S and variable assignment Γ over S it holds that $S, \Gamma \vdash e \Rightarrow x$ iff $S, \Gamma \vdash e' \Rightarrow x$.

Variable substitution is defined as usual except for \$dot and includes α conversion that may be necessary because of free variables in the expression that is substituted for the variable:

Definition 2.3 (Variable substitution). *Given a CXQ expression e, a variable $\$x$ and a CXQ expression e' we define $e[\$x/e']$ as follows:*

- $\$y[\$x/e'] = \begin{cases} e' & \textit{if}\ \$y = \$x \\ \$y & \textit{if}\ \$y \neq \$x \end{cases}$

- $a{::}n[\$x/e'] = \begin{cases} \mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ e'\ \mathsf{return}\ a{::}n & \textit{if}\ \$x = \$\mathsf{dot} \\ a{::}n & \textit{if}\ \$x \neq \$\mathsf{dot} \end{cases}$

- $\mathsf{ddo}(e)[\$x/e'] = \mathsf{ddo}(e[\$x/e'])$

- $(\mathsf{if}\ e_1\ \mathsf{then}\ e_2)[\$x/e'] = \mathsf{if}\ e_1[\$x/e']\ \mathsf{then}\ e_2[\$x/e']$

- $(\mathsf{for}\ \$y\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2)[\$x/e'] =$
  $\begin{cases} \mathsf{for}\ \$y\ \mathsf{in}\ e_1[\$x/e']\ \mathsf{return}\ e_2 & \textit{if}\ \$y = \$x \\ \mathsf{for}\ \$z\ \mathsf{in}\ e_1[\$x/e']\ \mathsf{return}\ (e_2[\$y/\$z])[\$x/e'] & \textit{if}\ \$y \neq \$x \end{cases}$
  with $\$z$ some variable not in $FV(e')$.

- $(\mathsf{let}\ \$y := e_1\ \mathsf{return}\ e_2)[\$x/e'] =$
  $\begin{cases} \mathsf{let}\ \$y := e_1[\$x/e']\ \mathsf{return}\ e_2 & \textit{if}\ \$y = \$x \\ \mathsf{let}\ \$z := e_1[\$x/e']\ \mathsf{return}\ (e_2[\$y/\$z])[\$x/e'] & \textit{if}\ \$y \neq \$x \end{cases}$
  with $\$z \neq \$\mathsf{dot}$ some variable not in $FV(e')$.

3. TREE PATTERN DECISION FOR CXQ

We present an algorithm for deciding wether the result of an XQuery expression always is in document order and duplicate-free. The approach is complete for CXQ, which was introduced in Section 2.3. We show in Section 4 that queries in CXQ that do yield ordered and duplicate-free results, can be expressed with a tree and we also provide an algorithm for determining which tree pattern the query corresponds to.

3.1 CXQ Properties

Just as for pure XPath expressions [10], it is possible to determine some static properties for CXQ expressions and their values that assist in deciding wether that expression returns ordered and duplicate-free results. A property π holds for an expression e if for any store S and any variable assignment Γ, s.t. $S, \Gamma \vdash e \Rightarrow v$, the list of nodes in v satisfies π.

Definition 3.1 (Value property). *We distinguish the following properties: no2d, gen, ord and nodup. If a value v over an store S has a property π then we denote this as $v : \pi$. The semantics of these properties is defined as follows:*

- *v : no2d iff there are not two distinct nodes in v;*
- *v : gen iff all nodes in v belong the the same generation of a tree in S;*
- *v : ord iff v is ordered in the document order of S;*
- *v : nodup iff every node appears at most once in v.*

The result of an expression v can be bound to a variable, in which case the variable is said to have the same properties as v. In order to derive properties for subexpressions that use variable references, we need to map every in-scope variable to a set of properties as follows.

Definition 3.2 (Property table). *A* property table *T is a function that maps variable names to sets of value properties. A value assignment Γ is said to satisfy T if for every variable $\$x$ and every $\pi \in T(\$x)$ it holds that $\Gamma(\$x) : \pi$.*

DEFINITION 3.3 (EXPRESSION PROPERTY). *We say that a CXQ expression e has property π under the property table T, denoted as $T \vdash e : \pi$, if it holds for every store S and every value assignment Γ over S that satisfies T that if $S, \Gamma \vdash e \Rightarrow x$ then $x : \pi$.*

In the following we define the notion of *root variable of an expression* which can be informally described as the variable from which the expression starts to navigate in order to obtain the resulting nodes.

DEFINITION 3.4 (ROOT VARIABLE). *The* root variable *of a CXQ expression e, denoted as $rv(e)$, is inductively defined as follows:*

- $rv(\$x) = \$x$
- $rv(a{::}n) = \$\mathsf{dot}$
- $rv(\mathsf{ddo}(e)) = rv(e)$
- $rv(\mathsf{if}\ e_1\ \mathsf{then}\ e_2) = rv(e_2)$
- $rv(\mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2) = \begin{cases} rv(e_1) & \textit{if}\ rv(e_2) = \$x \\ rv(e_2) & \textit{if}\ rv(e_2) \neq \$x \end{cases}$
- $rv(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_2) = \begin{cases} rv(e_1) & \textit{if}\ rv(e_2) = \$x \\ rv(e_2) & \textit{if}\ rv(e_2) \neq \$x \end{cases}$

The informal meaning of $rv(e)$ can be made more precise by the following claim: for every CXQ expression e there is a CXQ expression e' such that (1) $\mathsf{ddo}(e) \equiv \mathsf{ddo}(e')$ and (2) $FV(e') = \{rv(e)\}$ or e' is of the form if e'_1 then e'_2 such that $FV(e'_2) = \{rv(e)\}$. The will be proven formally in Section 4.

3.2 The algorithm

The algorithm for deriving the properties for a CXQ expression is defined by the set of inference rules given in Figure 3. In these rules the variables $e, e_1, \ldots$ range over expressions in CXQ, not all XQuery expressions. Indeed, not all of the rules are sound for arbitrary XQuery expressions.

Note that in the rules FORORDNORV2 and FORNODUPNORV the premisse $rv(e_2) \neq \$x$, which is present in FORSETNORV and FORORDNORV1, is omitted because it is unnecessary.

EXAMPLE 3.1. *We now illustrate the algorithm with a simple example. Consider the following XQuery expression:*

```
for $x in $d/item[description] return $x//listitem
```

This expression is normalized into CXQ as follows:

```
for $x in ddo(
    for $dot in ddo(
        for $dot in $d return child::item )
    return
        if child::description then $dot )
return ddo(
    for $dot in ddo(
        for $dot in $x return d-o-s::* )
    return child::listitem )
```

Assume $d: *no2d*, *nodup*, *then by* VARCLOS *we also know that* $d: *ord*, *gen*. *From* DDOSTEP *and* CHILDSTEP *we know that* child::item: *ord*, *nodup*, *gen*. *From* FORSETRC, FORORDRV2 *and* FORNODUPRV2 *we know that*

for $dot in $d return child::item: *ord*, *nodup*, *gen*

All these properties are preserved by the surrounding ddo *operation (*DDOSET, DDOSEQ*) and they also hold for the surrounding for expression because of* DOT, IF, FORORDRV1 *and* FORNODUPRV2. *Similarly, we can derive that*

```
ddo(
    for $dot in
        ddo( for $dot in $x return d-o-s::* )
    return child::listitem
) : ord, nodup.
```

Finally, we use FORORDRV2 *and* FORNODUPRV1 *to derive ord and nodup for the entire expression.*

The following theorem states that our algorithm is both sound and complete for CXQ, i.e., we derive *ord* (*nodup*) for a CXQ expression e iff for every XML store S and variable assignment over S it holds that the result of e is in document order (without duplicates).

THEOREM 3.1 (SOUNDNESS AND COMPLETENESS). *The inference rules in Figure 3 are sound and complete w.r.t. the ord and nodup properties for expressions in CXQ.*

PROOF SKETCH. We show for each type of expression that the presented rules are sound and complete. The proof proceeds by induction upon the structure of the expression. The reader is referred to the Technical Report [15] for the full proof. □

From the definition of the semantics of forest patterns it is clear that the *ord* and *nodup* properties are necessary properties in order for an expression to be equivalent with a forest pattern. In fact, as will be shown in the following section, this is also a sufficient condition since we demonstrate that every CXQ expressions of the form ddo(e) can be rewritten to a normal form that directly corresponds to and is equivalent with certain forest patterns. Therefore the presented inference rules are in effect an effective algorithm for deciding whether an expression is equivalent with a forest pattern.

4. RECOGNIZING FOREST PATTERNS

In this section we show that all CXQ expressions which are both *ord* and *nodup* correspond to a tree pattern and we give an algorithm to obtain this tree pattern. This algorithm is based on rewrite rules that reduce the expression to some normal form. These rewrite rules also derive information about to what extent the exact result of certain subexpression is relevant for the final result of the expression. For example, in an expression of the form ddo(e) the result of e can be changed by adding and/or remove duplicates or change the order of the nodes without affecting the result of the whole expression. Another example is an expression of the form if e_1 then e_2 where the result of e_1 can be changed as long as it is the empty sequence iff the original result of e_1 was empty without affecting the final result. To indicate these properties allow expressions to be annotated. An expression e annotated by α is denoted as ${}^{\alpha}e$ with α either $\cdot$, $\cup$ or $\vee$ which represent the *list concatenation*, *set union* and *boolean disjunction*, respectively. Informally they can be interpreted as saying that the value of the result of e may not be changed (for $\cdot$), the order may be changed

Name	Premises	Conclusion
VAR	$\pi \in T(\$x)$	$T \vdash \$x : \pi$
VARCLOS	$T \vdash \$x : no2d$	$T \vdash \$x : ord, gen$
DOT		$T \vdash \mathsf{\$dot} : no2d, nodup$
DDOSTEP	$a \in \{\mathsf{child}, \mathsf{desc}, \mathsf{d\text{-}o\text{-}s}\}$	$T \vdash a{::}n : ord, nodup$
CHILDSTEP		$T \vdash \mathsf{child}{::}n : gen$
DDOSET	$\pi \in \{no2d, gen\} \wedge T \vdash e : \pi$	$T \vdash \mathsf{ddo}(e) : \pi$
DDOSEQ		$T \vdash \mathsf{ddo}(e) : ord, nodup$
IF	$\pi \in \{no2d, gen, ord, nodup\} \wedge T \vdash e_2 : \pi$	$T \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ : \pi$
LET	$P = \{\pi \mid T \vdash e_1 : \pi\} \wedge T[\$x \mapsto P] \vdash e_2 : \pi$	$T \vdash \mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_2\ : \pi$
FORSETRV	$\pi \in \{no2d, gen\} \wedge T \vdash e_1 : \pi\ \wedge$ $T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : \pi$	$T \vdash \mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\ : \pi$
FORSETNORV	$\pi \in \{no2d, gen\} \wedge rv(e_2) \neq \$x\ \wedge$ $T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : \pi$	$T \vdash \mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\ : \pi$
FORORDRV1	$rv(e_2) = \$x \wedge T \vdash e_1 : ord\ \wedge$ $T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : no2d$	$T \vdash \mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\ : ord$
FORORDRV2	$rv(e_2) = \$x \wedge T \vdash e_1 : ord, gen, nodup\ \wedge$ $T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : ord$	$T \vdash \mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\ : ord$
FORORDNORV1	$rv(e_2) \neq \$x \wedge T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : no2d$	$T \vdash \mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\ : ord$
FORORDNORV2	$T \vdash e_1 : no2d, nodup \wedge T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : ord$	$T \vdash \mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\ : ord$
FORNODUPRV1	$rv(e_2) = \$x \wedge T \vdash e_1 : nodup, gen\ \wedge$ $T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : nodup$	$T \vdash \mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\ : nodup$
FORNODUPRV2	$rv(e_2) = \$x \wedge T \vdash e_1 : nodup\ \wedge$ $T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : nodup, gen$	$T \vdash \mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\ : nodup$
FORNODUPNORV	$T \vdash e_1 : no2d, nodup \wedge T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : nodup$	$T \vdash \mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\ : nodup$

Figure 3: Inference rules for deriving the *ord* and *nodup* properties for expressions in CXQ.

and duplicates may be added and removed (for $\cup$), and the result may be changed as long as it stays non-empty iff it was non-empty (for $\vee$). We use variables α, β, etc. to range over annotations. We use variables e, e_1 etc. to range over the expression part of an annotated expression.

For each annotation α we define an *interpretation function* I^α that is defined such that (1) $I^{\cdot}(x) = x$, (2) $I^{\cup}(x)$ is the set that contains exactly all nodes in x and (3) $I^{\vee}(x)$ is **false** if x is the empty sequence and **true** otherwise. These functions define for each annotation equivalence classes over sequences. Observe that for all annotations α it holds that $I^\alpha(e_1)\alpha I^\alpha(e_2) = I^\alpha(e_1 \cdot e_2)$. The semantics of ${}^\alpha e$ is then formally defined as the result of e which is mapped to an α-equivalent sequence. Note that this leads to a non-deterministic semantics.

We define a strict total ordering $\prec$ on the annotations such that $\cdot \prec \cup \prec \vee$.

DEFINITION 4.1 (CXQ$^+$). *The fragment* CXQ$^+$ *is defined as CXQ extended with annotations.*

The notion of variable substitution is generalized for CXQ$^+$ such that ${}^\alpha\$x[\$x/{}^\beta e] = {}^\gamma e$ with γ the maximum of α and β.

4.1 The Tree Pattern Normal Form

In this section we define the normal form to which we would like to normalize. The fundamental starting point is that we uniquely would like to find the *forest pattern* that is expressed by a CXQ expression with a ddo function applied to it. However, to understand the syntax to which we should normalize we first consider what would be the expected mapping from a forest pattern to a CXQ$^+$ expression.

4.1.1 A Mapping from Forest Patterns to CXQ$^+$

We start with describing how we expect that forest patterns are mapped to expressions in CXQ$^+$ with the mappings $\mathcal{X}^\cup$ for output patterns and $\mathcal{X}^\vee$ for condition patterns:

1. $\mathcal{X}^\alpha(\{t_1, \ldots, t_n\}) = {}^\alpha\mathsf{if}\ \mathcal{T}^\vee(t_1)\ \mathsf{then}\ \mathcal{X}^\alpha(\{t_2, \ldots, t_n\})$ if $n > 1$ and there is no output node in t_1
2. $\mathcal{X}^\alpha(\{t\}) = \mathcal{T}^\alpha(t)$

and $\mathcal{T}^\cup$ for output trees and $\mathcal{T}^\cup$ for condition trees:

1. $\mathcal{T}^\cup(l^{out}) = {}^\cup l$
2. $\mathcal{T}^\vee(l) = {}^\vee l$
3. $\mathcal{T}^\cup(l^{out}\{t_1, \ldots, t_n\}) =$ ${}^\cup\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^\cup l\ \mathsf{return}\ \mathcal{X}^\cup(\{t_1, \ldots, t_n, \mathsf{\$dot}^{out}\})$ if $n > 0$ and $l \neq \mathsf{\$dot}$
4. $\mathcal{T}^\alpha(l\{t_1, \ldots, t_n\}) =$ ${}^\alpha\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^\cup l\ \mathsf{return}\ \mathcal{X}^\alpha(\{t_1, \ldots, t_n\})$ if $n > 0$ and $l \neq \mathsf{\$dot}$
5. $\mathcal{T}^\alpha(\mathsf{\$dot}\{t\}) = \mathcal{T}^\alpha(t)$

Recall that nodes labeled $dot have no children if they are output node, and one child if they are not, so $\mathcal{T}^\cup$ and $\mathcal{T}^\vee$ are indeed defined for all output trees and condition trees, respectively. Observe that $\mathcal{X}^\cup$ preserves the semantics of the expression. Also observe that it is non-deterministic since it picks an order for the subtrees. Finally observe that it is injective, i.e., it maps different tree patterns to different CXQ$^+$ expressions.

We proceed with defining the syntax of the CXQ$^+$ fragment onto which forest patterns are mapped by $\mathcal{X}^\cup$. We will attempt to define this syntax such that (1) for every forrest pattern the result of $\mathcal{X}^\cup$ is in the syntax and (2) for every expression in the syntax there is forrest pattern that

Annotation Introduction and Propagation, and ddo Removal

Source	Result	Condition
${}^{\cdot}\mathsf{ddo}({}^{\cdot}e)$	${}^{\cdot}\mathsf{ddo}({}^{\cup}e)$	
${}^{\alpha}\mathsf{if}\ {}^{\beta}e_1\ \mathsf{then}\ {}^{\gamma}e_2$	${}^{\alpha}\mathsf{if}\ {}^{\vee}e_1\ \mathsf{then}\ {}^{\gamma}e_2$	$\gamma \prec \vee$
${}^{\alpha}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\cdot}e_1\ \mathsf{return}\ {}^{\gamma}e_2$	${}^{\alpha}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\gamma}e_2$	$\cdot \prec \alpha$
${}^{\alpha}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\beta}e_1\ \mathsf{return}\ {}^{\gamma}e_2$	${}^{\alpha}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\beta}e_1\ \mathsf{return}\ {}^{\alpha}e_2$	$\gamma \prec \alpha$
${}^{\alpha}\mathsf{let}\ \$x := {}^{\beta}e_1\ \mathsf{return}\ {}^{\gamma}e_2$	${}^{\alpha}\mathsf{let}\ \$x := {}^{\beta}e_1\ \mathsf{return}\ {}^{\alpha}e_2$	$\gamma \prec \alpha$
${}^{\alpha}\mathsf{if}\ {}^{\beta}e_1\ \mathsf{then}\ {}^{\gamma}e_2$	${}^{\alpha}\mathsf{if}\ {}^{\beta}e_1\ \mathsf{then}\ {}^{\alpha}e_2$	$\gamma \prec \alpha$
${}^{\alpha}\mathsf{ddo}({}^{\beta}e)$	${}^{\alpha}e$	$\cdot \prec \alpha \wedge \beta \preceq \alpha$

Figure 4: An overview of the propagation rules for annotations and the ddo removal rule.

is mapped to it. That (1) holds can by readily observed by noting that the following holds for the non-terminals: fp describes the range of $\mathcal{X}^{\cup}$, tp describes the range of $\mathcal{T}^{\vee}$, otp describes the range of $\mathcal{T}^{\cup}$, atp describes the range of $\mathcal{T}^{\vee}$ restricted to trees with a $\mathsf{\$dot}$ root, and $aotp$ describes the range of $\mathcal{T}^{\cup}$ restricted to trees with other roots.

Definition 4.2 (TPNF). *Defined by the syntax:*

$$\begin{array}{rcl}
fp & ::= & otp \mid {}^{\cup}\mathsf{if}\ tp\ \mathsf{then}\ fp \\
tp & ::= & atp \mid {}^{\vee}\$x \mid {}^{\vee}\mathsf{for\ \$dot\ in}\ {}^{\cup}\$x\ \mathsf{return}\ rc \\
otp & ::= & aotp \mid {}^{\cup}\$x \mid {}^{\cup}\mathsf{\$dot} \mid {}^{\cup}\mathsf{for\ \$dot\ in}\ {}^{\cup}\$x\ \mathsf{return}\ orc \\
atp & ::= & {}^{\vee}ax{::}nt \mid {}^{\vee}\mathsf{for\ \$dot\ in}\ {}^{\cup}ax{::}nt\ \mathsf{return}\ rc \\
aotp & ::= & {}^{\cup}ax{::}nt \mid {}^{\cup}\mathsf{for\ \$dot\ in}\ {}^{\cup}ax{::}nt\ \mathsf{return}\ orc \\
rc & ::= & atp \mid {}^{\vee}\mathsf{if}\ atp\ \mathsf{then}\ rc \\
orc & ::= & aotp \mid {}^{\cup}\mathsf{if}\ atp\ \mathsf{then}\ (orc \mid {}^{\cup}\mathsf{\$dot})
\end{array}$$

where $\$x$ *refers to the set of variables minus* $\mathsf{\$dot}$.

4.1.2 A Mapping from TPNF to Forest Patterns

Since the claim is that the normal form allows us to easily recognize forest patterns, we now investigate the inverse of $\mathcal{X}^{\cup}$. This is defined by the mapping $\mathcal{F}$ that maps subexpressions of TPNF expressions to forest patterns such that expressions associated with tp, atp and rc are mapped to a condition pattern, and expressions associated with fp, otp and $aotp$ are mapped to an output pattern:

1. $\mathcal{F}({}^{\cup}\$x) = \$x^{out}$
2. $\mathcal{F}({}^{\vee}\$x) = \$x$
3. $\mathcal{F}({}^{\cup}a{::}n) = \mathsf{\$dot}\{a{::}n^{out}\}$
4. $\mathcal{F}({}^{\vee}a{::}n) = \mathsf{\$dot}\{a{::}n\}$
5. $\mathcal{F}({}^{\alpha}\mathsf{if}\ {}^{\vee}e_1\ \mathsf{then}\ {}^{\alpha}e_2) = \mathcal{F}({}^{\vee}e_1) + \mathcal{F}({}^{\alpha}e_2)$
6. $\mathcal{F}({}^{\alpha}\mathsf{for\ \$dot\ in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\alpha}e_1) = \$x\{t_1, \ldots, t_n\}$
   if $\mathcal{F}({}^{\alpha}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}\}$
7. $\mathcal{F}({}^{\cup}\mathsf{for\ \$dot\ in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\cup}e_1) = \$x^{out}\{t_1, \ldots, t_n\}$
   if $\mathcal{F}({}^{\cup}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}, \mathsf{\$dot}^{out}\}$
8. $\mathcal{F}({}^{\alpha}\mathsf{for\ \$dot\ in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\alpha}e_1) =$
   $\mathsf{\$dot}\{a{::}n\{t_1, \ldots, t_n\}\}$
   if $\mathcal{F}({}^{\alpha}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}\}$
9. $\mathcal{F}({}^{\cup}\mathsf{for\ \$dot\ in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\cup}e_1) =$
   $\mathsf{\$dot}\{a{::}n^{out}\{t_1, \ldots, t_n\}\}$
   if $\mathcal{F}({}^{\cup}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}, \mathsf{\$dot}^{out}\}$

Observe that $\mathcal{F}$ is deterministic and is defined on all TPNF expressions and their subexpressions. The latter can be shown with induction on the abstract syntax tree of an expression and the observation that for expressions associated with the nonterminal rc the result of $\mathcal{F}$ is always of the form $\{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}\}$ and the result of an expression associated with the nonterminal orc is of this form or of the form $\{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}, \mathsf{\$dot}^{out}\}$.

The relationship with $\mathcal{X}^{\cup}$ is established by the following theorem.

Theorem 4.1. *The function $\mathcal{F}$ restricted to TPNF is the inverse of $\mathcal{X}^{\cup}$, i.e., for every expression ${}^{\cup}e$ in TPNF it holds that $\mathcal{F}({}^{\cup}e) = f$ iff $\mathcal{X}^{\cup}(f) = {}^{\cup}e$.*

Proof. It can be proven with induction upon the abstract syntax tree of the TPNF expression that it holds for each subexpression ${}^{\alpha}e$ of a TPNF expression that $\mathcal{F}({}^{\alpha}e) = f$ iff $\mathcal{X}^{\alpha}(f) = {}^{\alpha}e$. $\square$

Since it was already established that the range of $\mathcal{X}^{\cup}$ was a subset of TPNF it now follows that TPNF is exactly this range. Moreover, with $\mathcal{F}$ we are given a simple procedure to recognize which forest pattern is represented by a certain TPNF expression, which motivates why TPNF is an interesting normal form for recognizing forest patterns.

4.2 Normalization Rules

The normalization rules for rewriting an expression to TPNF are given in Figure 4 and Figure 5. A rewrite rule can be applied if the source matches a certain expression and the specified condition is satisfied.

The rules in Figure 4 mainly introduce and propagate annotations but do not change the structure of the expression. The only exception is the final rule a ddo operation if the annotation tells us that it is not necessary. Observe that in an expression with only $\cdot$ annotations the first two rules will usually start with introducing annotations and the other rules will propagate these annotations to subexpression. There are two important exceptions: a $\vee$ annotation is

Structural Manipulation

Name	Source	Result	Condition
Substitution	${}^{\alpha}$let $\$x := {}^{\beta}e_1$ return ${}^{\alpha}e_2$	${}^{\alpha}e_2[\$x/{}^{\beta}e_1]$	${}^{\beta}e_1, {}^{\alpha}e_2 \in CXQ^{+}, \cup \preceq \alpha$
Loop Fusion	${}^{\alpha}$for \$dot in ${}^{\cup}$for \$dot in ${}^{\cup}e_1$ return ${}^{\cup}e_2$ return ${}^{\alpha}e_3$	${}^{\alpha}$for \$dot in ${}^{\cup}e_1$ return ${}^{\alpha}$for \$dot in ${}^{\cup}e_2$ return ${}^{\alpha}e_3$	$\cup \preceq \alpha$
Condition Detection	${}^{\alpha}$for $\$x$ in ${}^{\cup}e_1$ return ${}^{\alpha}e_2$	${}^{\alpha}$if ${}^{\vee}e_1$ then ${}^{\alpha}e_2$	$\cup \preceq \alpha$ and $\$x \notin FV(e_2)$
Condition Shift	${}^{\alpha}$if (${}^{\vee}$if ${}^{\vee}e_1$ then ${}^{\vee}e_2$) then ${}^{\alpha}e_3$	${}^{\alpha}$if ${}^{\vee}e_1$ then ${}^{\alpha}$if ${}^{\vee}e_2$ then ${}^{\alpha}e_3$	none
Return Condition Lift	${}^{\alpha}$for $\$x$ in ${}^{\cup}e_1$ return (${}^{\alpha}$if ${}^{\vee}e_2$ then ${}^{\alpha}e_3$)	${}^{\alpha}$if ${}^{\vee}e_2$ then (${}^{\alpha}$for $\$x$ in ${}^{\cup}e_1$ return ${}^{\alpha}e_3$)	$\cup \preceq \alpha$ and $\$x \notin FV(e_2)$
Nested Return Cond. Lift	${}^{\alpha}$for $\$x$ in ${}^{\cup}e_1$ return (${}^{\alpha}$if ${}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n$ then ${}^{\alpha}e_{n+1}$)	${}^{\alpha}$if ${}^{\vee}e_n$ then (${}^{\alpha}$for $\$x$ in ${}^{\cup}e_1$ return (${}^{\alpha}$if ${}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_{n-1}$ then ${}^{\alpha}e_{n+1}$))	$\cup \preceq \alpha$ and $n > 2$ and $\$x \notin FV(e_n)$
Return Result Lift	${}^{\alpha}$for $\$x$ in ${}^{\cup}e_1$ return ${}^{\alpha}$if ${}^{\vee}e_2$ then ${}^{\alpha}e_3$	${}^{\alpha}$if (${}^{\vee}$for $\$x$ in ${}^{\cup}e_1$ return ${}^{\vee}e_2$) then ${}^{\alpha}e_3$	$\cup \preceq \alpha$ and $\$x \notin FV(e_3)$
Nested Return Result Lift	${}^{\alpha}$for $\$x$ in ${}^{\cup}e_1$ return (${}^{\alpha}$if ${}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n$ then ${}^{\alpha}e_{n+1}$)	${}^{\alpha}$if (${}^{\vee}$for $\$x$ in ${}^{\cup}e_1$ return (${}^{\vee}$if ${}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_{n-1}$ then ${}^{\vee}e_n$)) then ${}^{\alpha}e_{n+1}$	$\cup \preceq \alpha$ and $n > 2$ and $\$x \notin FV(e_{n+1})$
For Condition Lift	${}^{\alpha}$for $\$x$ in (${}^{\cup}$if ${}^{\vee}e_1$ then ${}^{\cup}e_2$) return ${}^{\alpha}e_3$	${}^{\alpha}$if ${}^{\vee}e_1$ then ${}^{\alpha}$for $\$x$ in ${}^{\cup}e_2$ return ${}^{\alpha}e_3$	$\cup \preceq \alpha$
Trivial Dot Condition	${}^{\alpha}$if ${}^{\vee}$\$dot then ${}^{\alpha}e_2$	${}^{\alpha}e_2$	None
Trivial Loop	${}^{\alpha}$for $\$x$ in ${}^{\cup}e$ return ${}^{\alpha}\$x$	${}^{\alpha}e$	$\cup \preceq \alpha$
Introduction of Dot	${}^{\alpha}$for $\$x$ in ${}^{\cup}e_1$ return ${}^{\alpha}e_2$	${}^{\alpha}$for \$dot in ${}^{\cup}e_1$ return ${}^{\alpha}e_2[\$x/{}^{\cup}$\$dot$]$	\$dot $\notin FV(e_2)$ and $\$x \neq$ \$dot
Dot Loop	${}^{\alpha}$for \$dot in ${}^{\cup}$\$dot return ${}^{\alpha}e_1$	${}^{\alpha}e_1$	$\cup \preceq \alpha$
Shortening Condition	${}^{\vee}$if ${}^{\vee}e$ then ${}^{\vee}$\$dot	${}^{\vee}e$	None

Figure 5: An overview of the structural rewrites.

propagated in the form of a $\cup$ annotation to the expression in the for clause of a for expression, and no annotation is propagated to the let clause of a let expression.

The rules in Figure 5 actually change the structure of the expression. The first rule is the **substitution** rule that remove a let expression. Note that this rule is not sound for general XQuery expressions due to possible side effects of node construction, so the condition restricts this rule to only CXQ+ expression for which it is in fact correct. All the other rules are correct for arbitrary XQuery expressions, provided they are correctly annotated. Although the substitution rule may lead to duplication of expressions, and therefore a less efficient query plan, it is only applied to $\cup$ annotated CXQ expressions and therefore the result is guaranteed to be a forest pattern for which there is probably an efficient physical query plan. On the other hand it can be shown that a more conservative substitution rule that only substitutes when the variable appears once in e_2 is not sufficient. Consider, for example, an expression of the form

```
for $x in $y/p1 return
    let $z := $x/p2 return
        if $z/p3 then $z/p4
```

where $\$\mathsf{y}/p_1$, $\$\mathsf{x}/p_2$, $\$\mathsf{z}/p_3$ and $\$\mathsf{z}/p_4$ denote TPNF expressions with the indicated variable as the only free variable. This let expression would then not be removed, and TPNF would not be reached, although it is equivalent with the path expression $\$\mathsf{y}/p_1[p_2/p_3]/p_2/p_4$.

The rules after **substitution** all presume that the expression is already in some intermediate normal form, which is defined by the following lemma.

Lemma 4.1. *When the rules in Figure 4 and the* **Substitution** *rule from Figure 5 are applied exhaustively to a CXQ$^+$ expression of the form $^{\cup}e$ where all subexpressions in e are annotated with · then the result is in the following syntax:*

$$
\begin{array}{lcl}
se & ::= & {}^{\cup}\$x \mid {}^{\cup}ax{::}nt \mid {}^{\cup}\mathsf{if}\ be\ \mathsf{then}\ se \mid {}^{\cup}\mathsf{for}\ \$x\ \mathsf{in}\ se\ \mathsf{return}\ se \\
be & ::= & {}^{\vee}\$x \mid {}^{\vee}ax{::}nt \mid {}^{\vee}\mathsf{if}\ be\ \mathsf{then}\ be \mid {}^{\vee}\mathsf{for}\ \$x\ \mathsf{in}\ se\ \mathsf{return}\ be \\
nt & ::= & label \mid * \\
ax & ::= & \mathsf{child} \mid \mathsf{desc} \mid \mathsf{d\text{-}o\text{-}s}
\end{array}
$$

Informally the non-terminal *se* defines the *set expressions* and *be* the *boolean expressions.*

Example 4.1. *Since we can derive ord and nodup for the expression e in Example 3.1, we know that e is equivalent to* ddo(e)*. If we now add annotations, we obtain the following CXQ$^+$ expression:*

```
·ddo(
    ·for $x in ·ddo(
        ·for $dot in ·ddo(
            ·for $dot in ·$d return ·child::item )
        return
            ·if ·child::description then ·$dot )
    return ·ddo(
        ·for $dot in ·ddo(
            ·for $dot in ·$x return ·d-o-s::* )
        return ·child::listitem )
)
```

If we now apply the rules of Figure 4 exhaustively, we obtain the following expression:

```
·ddo(
    ∪for $x in
        ∪for $dot in
            ∪for $dot in ∪$d return ∪child::item
        return
            ∪if ∨child::description then ∪$dot
    return
        ∪for $dot in
            ∪for $dot in ∪$x return ∪d-o-s::*
        return ∪child::listitem
)
```

We then have a few possible structural manipulations to perform. For example, after using loop fusion on both subexpressions of the upper for *expression we obtain the following (intermediate) result:*

```
·ddo(
    ∪for $x in
        ∪for $dot in ∪$d return
            ∪for $dot in ∪child::item return
                ∪if ∨child::description then ∪$dot
    return
        ∪for $dot in ∪$x return
            ∪for $dot in ∪d-o-s::* return ∪child::listitem
)
```

We can then introduce the $dot *variable in the outer* for *expression and again apply a few times loop fusion, a for condition lift, and dot loop. We then obtain the following CXQ$^+$ expression:*

```
·ddo(
  ∪for $dot in ∪$d return
    ∪for $dot in ∪child::item return
      ∪if ∨child::description then
        ∪for $dot in ∪d-o-s::* return ∪child::listitem
)
```

The expression within the ddo *call is clearly in TPNF.*

4.3 Correctness of Normalization Rules

The soundness of the rewrite rules, i.e., they preserve the semantics of the expressions, is easily verified. However, we also need to show that when applied exhaustively they rewrite any CXQ$^+$ expression of the form $^{\cup}e$ to an expression in TPNF.

Lemma 4.2. *If a CXQ$^+$ expression is in TPNF, then none of the rewrite rules apply.*

Proof. (Sketch) When considering every rewrite rule separately, we can see that none of the conditions that enable this rule can be satisfied in a normalized expression. □

Lemma 4.3. *If no rewrite rule applies to a CXQ$^+$ expression $^{\cup}e$ then it is in TPNF.*

Proof. (Sketch) If no rewrite rule applies then the expression must be in the normal form of Lemma 4.1. Then we prove the lemma by induction on the structure of an expression in this normal form. We assume that all subexpressions are in TPNF and then show for all cases that either the complete expression is already in TPNF or one of the rules applies. □

Summarizing the two preceding lemmas state that if the rewrite process terminates then the result will be in TPNF. So it remains to be proven that the rewrite process always terminates.

Lemma 4.4. *The rewrite process always terminates.*

Proof. (Sketch) We associate with each CXQ^+ expression a cost which strictly diminishes when applying any of the structural rewrite rules. A cost function c maps CXQ^+ expressions to a natural number. Based on this notion, a combined cost function C is defined by an n-tuple of cost functions $\langle c_1, \dots, c_n\rangle$, where $C(e) = \langle c_1(e), \dots, c_n(e)\rangle$.

We can define a combined cost function C by a 5-tuple $\langle c_1, c_2, c_3, c_4, c_5\rangle$ for which it holds that, when looking at the lexicographical order, the cost for all expressions e diminishes when applying a structural rewrite rule, i.e., if e_1 is rewritten to e_2 then it holds for some c_i that $c_i(e_2) < c_i(e_1)$ and for all c_j with $j < i$ it holds that $c_i(e_2) = c_i(e_1)$. Intuitively, the function c_1 indicates the number of let expressions, c_2 indicates the size of the expression, c_3 states that we want to make the in-clauses of for-loops as simple as possible, c_4 states that we want to get as much subexpressions as possible outside of for-loops and finally, c_5 states that we want to push as much as possible out of the if-clause.

This sketch only considers structural manipulations. It can easily be seen that the annotation propagation does not change the cost of an expression, because the structure remains the same. Moreover, we can only do a finite number of propagations between the application of two structural manipulations. □

Combining these lemmas we then obtain the desired theorem.

Theorem 4.2. *When applied in an arbitrary order the rewrite rules rewrite every CXQ^+ expression of form ${}^{\cup}e$ to an expression in TPNF within a finite number of steps.*

Proof. This follows from Lemma 4.3, Lemma 4.2 and Lemma 4.4. □

As a corollary it now follows that every CXQ expression e that has the properties *ord* and *nodup* is equivalent with a forest pattern. This follows since then e is equivalent with ${}^{\cdot}\mathsf{ddo}({}^{\cdot}e)$ where ${}^{\cdot}e$ is e extended with the $\cdot$ annotation for each subexpression. Since the rewrite rules can rewrite this to ${}^{\cdot}\mathsf{ddo}({}^{\cup}e)$ and ${}^{\cup}e$ can always be rewritten to an expression in TPNF, it follows that e is equivalent with a forest pattern.

4.4 Confluence of Normalization

The presented set of rewriting rules is strictly speaking not confluent, i.e., the obtained final result may depend on which rules are applied in what order, because, for example, if the order in which conditions are lifted by the **Nested Return Condition Lift** is changed the final result might be different. However it can be shown that the result will be a unique forest pattern if after rewriting the result is transformed into a tree pattern with the function $\mathcal{F}$.

Next to the forest pattern union $f_1 + f_2$ we introduce the following operations for forest patterns:

- $\emptyset$ denotes the empty forest pattern.
- $c(f)$ removes from f the output marking of the output node. If this leaves a $\mathsf{\$dot}$ node with no children it is removed.
- $f_1 \lhd f_2$ adds the children of roots in f_2 children under the output node in f_1 and removes the output marking from the output node of f_1 except if one of the roots of f_2 is an output node.
- $f_1 \lhd^* f_2$ is defined such that $f_1 \lhd^* \{t_1, \dots, t_n\} = f_1 \lhd \{t_1\} + \dots + f_1 \lhd \{t_n\}$ and $f_1 \lhd^* \emptyset = \emptyset$.
- $f^{\$x}$ and $f^{-\$x}$ select from f the trees with roots labeled $\$x$ and with roots not labeled $\$x$, respectively.

Based on these operations we then define the mapping $\mathcal{M}$ from arbitrary CXQ$^+$ expressions to forest patterns as follows. For expressions annotated with $\alpha \preceq \cup$ we define $\mathcal{M}$ such that:

- $\mathcal{M}({}^{\alpha}\$x) = \$x^{out}$
- $\mathcal{M}({}^{\alpha}a{::}n) = \mathsf{\$dot}\{a{::}n^{out}\}$
- $\mathcal{M}({}^{\alpha}\mathsf{ddo}({}^{\beta}e)) = \mathcal{M}({}^{\beta}e)$
- $\mathcal{M}({}^{\alpha}\mathsf{if}\ {}^{\beta}e_1\ \mathsf{then}\ {}^{\gamma}e_2) = c(\mathcal{M}({}^{\beta}e_1)) + \mathcal{M}({}^{\gamma}e_2)$
- $\mathcal{M}({}^{\alpha}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\beta}e_1\ \mathsf{return}\ {}^{\gamma}e_2) =$
  $\mathcal{M}({}^{\gamma}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd \mathcal{M}({}^{\gamma}e_2)^{\$x})$
- $\mathcal{M}({}^{\alpha}\mathsf{let}\ \$x := {}^{\beta}e_1\ \mathsf{return}\ {}^{\gamma}e_2) =$
  $\mathcal{M}({}^{\gamma}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd^* \mathcal{M}({}^{\gamma}e_2)^{\$x})$

And for expressions annotated with $\vee$ we define $\mathcal{M}$ such that $\mathcal{M}({}^{\vee}e) = c(\mathcal{M}({}^{\cup}e))$.

Observe that in all cases the result is a well-defined output pattern. For example, for $\mathcal{M}(\mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2)$ it holds that either $\mathcal{M}(e_2)^{-\$x}$ or $\mathcal{M}(e_2)^{\$x}$ is an output pattern and since $\mathcal{M}(e_1) \lhd M(e_2)^{\$x}$ is an output pattern iff $M(e_2)^{\$x}$ is an output pattern, the expression $\mathcal{M}(e_2)^{-\$x} + (\mathcal{M}(e_1) \lhd \mathcal{M}(e_2)^{\$x})$ has a well-defined result.

This mapping has two important properties. The first is that it is invariant under the rewrite rules.

Lemma 4.5. *For all rewrites rules it holds that if they rewrite e_1 to e_2 then $\mathcal{M}(e_1) = \mathcal{M}(e_2)$.*

Proof. (Sketch) This can be shown for each of the rules using the algebraic properties that hold for the operations used to define $\mathcal{M}$. □

The second important property is that it coincides with $\mathcal{F}$ on TPNF.

Lemma 4.6. *For every CXQ$^+$ expression ${}^{\cup}e$ it holds that if $\mathcal{F}({}^{\cup}e)$ is defined then $\mathcal{F}({}^{\cup}e) = \mathcal{M}({}^{\cup}e)$.*

Proof. (Sketch) It can be proven with induction upon the structure of ${}^{\cup}e$ that for all CXQ$^+$ expressions ${}^{\alpha}e$ it holds that if $\mathcal{F}({}^{\alpha}e)$ is defined then $\mathcal{F}({}^{\alpha}e) = \mathcal{M}({}^{\alpha}e)$. □

This then leads to the theorem that states the confluence of the rewrite process.

Theorem 4.3. *If a CXQ$^+$ expressions ${}^{\cup}e$ can be rewritten to the TPNF expressions ${}^{\cup}e_1$ and ${}^{\cup}e_2$ then $\mathcal{F}({}^{\cup}e_1) = \mathcal{F}({}^{\cup}e_2)$.*

Proof. By Lemma 4.5 the result of mapping $\mathcal{M}$ remains the same after every rewrite and therefore $\mathcal{M}({}^{\cup}e) = \mathcal{M}({}^{\cup}e_1) = \mathcal{M}({}^{\cup}e_2)$. It then follows by Lemma 4.6 that $\mathcal{F}({}^{\cup}e_1) = \mathcal{F}({}^{\cup}e_2)$. □

5. RELATED LITERATURE

Detecting and identifying tree patterns within XQuery expressions has gained importance as a result of two – not entirely unrelated – technical evolutions. First, many XQuery algebra systems are capable of expressing tree patterns with an algebraic operator, like *TAX* [16] or *Galax* [23] and second, a growing number of advanced evaluation strategies and accompanying indexing systems for tree patterns is being published, for instance the *staircase join* [14] and *holistic twig joins* [2].

More closely related to our work, the framework presented in [7] and extended in [24], focusses on minimizing navigation within nested subqueries. In contrast to our work, they do not focus on discovering tree patterns inside queries and they ignore existential XPath queries. Hence their approach is fully complementary to our normalization strategy. Quite similarly, a proposed technique for identifying tree patterns [1], uses tree patterns as a way of identifying the set of views that can be used during query evaluation. This is in contrast with our approach, where we try to identify the parts of the query that can be evaluated using optimal XPath evaluation strategies. Similar strategies have been proposed to project out those parts of XML document trees that are not accessed by a query [22].

Another seemingly useful and promising means for XQuery normalization in general, is to the monoid calculus as described for object base query languages [9]. The use of this approach is the subject of futher research. In more general terms, the relevance of our work is illustrated by [25], where normal forms open up the road to a better understanding of some formal properties of functional query languages,as well as further optimization oportunities.

6. CONCLUSION

We have presented a method for detecting tree pattern expressions in arbitrary XQuery expressions. It remains to be noted that many of the rules for deriving the *ord* and *nodup* properties for the supported CXQ fragment of XQuery can be generalized. Similarly, some of the normalization rules can be generalized to operate over the entire language and in the absence of annotations. The extent of this robustness is the subjkect of further research. The proposed strategy is complete for an important fragment of the XQuery language and it is complementatry to other query optimization approaches like those in NEXT [7], Galax [22] or the view-based rewrites in [1]. Our normalization algorithms are capable of identifying and extracting tree pattern expressions from queries, enabling the use of specialized algorithms to evaluate them. To our knowledge, this paper is the first to present a complete aproach towards the identification and normalization of tree pattern expressions. The presented techniques are designed to easily fit inside any XQuery compiler.

7. REFERENCES

[1] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-based identification of tree patterns in xquery. In *FQAS*, pages 13–25, 2006.

[2] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, pages 310–321, Madison, Wisconsin, 2002.

[3] T. Chen, T. W. Ling, and C. Y. Chan. Prefix path streaming: A new clustering method for optimal holistic XML twig pattern matching. In *DEXA*, pages 801–810, 2004.

[4] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, Hong Kong, China, Aug. 2002.

[5] J.-K. M. Chin-Wan Chung and K. Shim. APEX : An adaptive path index for XML data. *SIGMOD*, 15(5):121–132, June 2002.

[6] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In *DEXA*, pages 28–37, 2003.

[7] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT logical framework for XQuery. In *VLDB*, pages 168–179, Toronto, Canada, Aug. 2004.

[8] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C working draft. Candidate Recommendation, Nov. 2005.

[9] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

[10] M. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. Optimizing sorting and duplicate elimination in XQuery path expressions. volume 3588 of *Lecture Notes in Computer Science*, pages 554–563, Copenhagen, Denmark, 2005.

[11] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing cursor movement in holistic twig joins. In *CIKM*, pages 784–791, New York, NY, USA, 2005. ACM Press.

[12] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[13] T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29(1):91–131, 2004.

[14] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *VLDB*, pages 524–525, 2003.

[15] J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. How to recognise different kinds of tree patterns from quite a long way away. Technical Report TR-UA-2006-13, University of Antwerp and IBM Watson, 2006. `http://adrem.ua.ac.be/bibrem/`.

[16] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. Tax: A tree algebra for xml. In *DBPL*, pages 149–164, 2001.

[17] H. Jiang, H. Lu, and W. Wang. Efficient processing of twig queries with or-predicates. In *SIGMOD*, pages 59–70, 2004.

[18] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB*, pages 273–284, 2003.

[19] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, September 2001.

[20] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *VLDB*,

pages 193–204, 2005.

[21] J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni. Efficient processing of ordered XML twig pattern. In *DEXA*, pages 300–309, 2005.

[22] A. Marian and J. Siméon. Projecting xml documents. In *VLDB*, pages 213–224, 2003.

[23] P. Michiels, G. A. Mihăilă, and J. Siméon. Put a tree pattern in your tuple algebra. In *ICDE*, 2007. to appear.

[24] S. Wang, E. A. Rundensteiner, and M. Mani. Optimization of nested xquery expressions with orderby clauses. In *ICDE Workshops*, page 1277, 2005.

[25] L. Wong. Normal forms and conservative properties for query languages over collection types. In *PODS*, pages 26–36, Washington, D.C., United States, 1993.