How to Recognise Different Tree Patterns from quite a Long Way Away

Jan Hidders    Philippe Michiels    Jérôme Siméon    Roel Vercammen

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 [Bruno et al., 2002, Fontoura et al., 2005, Lu et al., 2005b, Jiang et al., 2004, Choi et al., 2003, Gottlob et al., 2002, Grust et al., 2003] and corresponding indexes [Grust et al., 2004, Chin-Wan Chung and Shim, 2002, Li and Moon, 2001, Chien et al., 2002, Jiang et al., 2003, Lu et al., 2005a, Chen et al., 2004]. 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 [Bruno et al., 2002] 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 focuses 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 [Fernández et al., 2005], 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 [Fernández et al., 2005] 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.

$Q1a$  
$\texttt{d}[/\text{x} paradigm]/\text{x}[/\text{emailaddress}]/\text{x}[/\text{name}]

$Q1b$  
\begin{align*}
\text{let } x := \\
\text{for } y \text{ in } d[/\text{x} paradigm]
\text{return } y
\end{align*}

$Q1c$  
\begin{align*}
\text{for } y \text{ in } \text{x}[/\text{emailaddress}]
\text{where } y \text{/emailaddress}
\text{return } y
\end{align*}

$Q2$  
\begin{align*}
\text{for } x \text{ in } \text{d}[/\text{x} paradigm]
\text{return } x[/\text{name}]
\end{align*}

$Q2n$  
\begin{align*}
\text{for } x \text{ in } \text{d}[/\text{x} paradigm]
\text{where } x[/\text{description}]
\text{return } x[/\text{listitem}]
\end{align*}

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

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 appropriate 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 [Michiels et al., 2007]. 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 6 and conclude the paper in Section 7.

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, (simply called store here) simplified here to ordered sets of ordered node-labeled trees, denoted by variables \( S, S', \ldots, S_1, \ldots \).
- Sub/super-store: \( S \) is 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 \( \langle n_1, n_2, n_3 \rangle \).

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 \( \$\text{dot} \) 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 directed under the root. If the root of the tree is an output node then we add to \( l \) a superscript \( \text{out} \) as in \( \$x^{\text{out}} \). If the set of subtrees is empty then we omit it altogether. A forest pattern that consists of the trees \( t_1, \ldots, t_a \) is simply denoted as \( \{t_1, \ldots, t_a\} \). 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 set of trees is defined given a store \( S \) and variable assignment \( \Gamma \) over \( S \). It is defined by embeddings which are functions \( h \) from the nodes of the pattern to 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 \( S \) or with \( \Gamma(\$\text{dot}) \) if \( n \) has not parent, 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. For a forest pattern this will be expressed by the judgment $S, \Gamma \vdash f \Rightarrow x$ where $S$ a store, $\Gamma$ a variable assignment over $S$ and $x$ the value over $S$ that is the result of evaluating $f$ under $S$ and $\Gamma$.

Note that the forest patterns $\{x\{child:a\}, x\{child:b^{out}\}\}$ and $\{x\{child:a, child:b^{out}\}\}$ do not have the same semantics since the first returns all $b$ children of nodes in $x$ if there is a node in $x$ with an $a$ child, whereas the second output pattern returns the $b$ children of nodes in $x$ if these nodes in $x$ have an $a$ child.

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:

\[
\begin{align*}
expr & ::= \$x \mid \text{axis}:\text{ntest} \mid \text{ddo}(expr) \mid \text{if}\ expr \ \text{then}\ expr \ \text{else}\ () \mid \text{for}\ $x\ \text{in}\ expr \ \text{return}\ expr \\
\text{ntest} & ::= \text{label} \mid \ast \\
\text{axis} & ::= \text{child} \mid \text{desc} \mid \text{d-o-s}
\end{align*}
\]

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 [Draper et al., 2005], 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) = \{$$dot$$\}$. The judgment $S, \Gamma \vdash e \Rightarrow x$ where $S$ a store, $\Gamma$ a variable assignment over $S$, $e$ a CXQ expression and $x$ a value over $S$ denotes that $x$ can be the result of the evaluation of $e$ under $S$ and $\Gamma$. Two expressions $e$ and $e'$ are said to be equivalent, denoted as $e \equiv e'$, if for every store $S$ and variable assignment $\Gamma$ 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 $\alpha$ 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:
It can be shown that the result of a substitution is well-defined. For this we define the expanded size of an expression as the size of the syntax tree of $e$ after all occurrences of the form $ax::nt$ that do not appear as $e$ in an expression of the form for $\$dot$ in $e'$ return $ax::nt$ if $\$y = \$dot$ and if $\$x \neq \$dot$. 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. Before we proceed, we need to discuss some preliminary lemmas and theorems that are required to show soundness and completeness for the decision algorithm.

**Lemma 3.1** (Unique result of CXQ). Given a store $S$, an assignment $\Gamma$ over $S$ and a CXQ expression $e$ there is exactly one value $v$ over $S$ such that $S, \Gamma \vdash e \Rightarrow v$.

**Proof.** Easily shown with induction upon the structure of $e$. \hfill \square

**Lemma 3.2** (Monotonicity of CXQ). For all stores $S, S'$, assignments $\Gamma, \Gamma'$ over $S$ and $S'$, respectively, and CXQ expressions $e$ it holds that if $S'$ is a super-store of $S$ and $\Gamma'$ is a super-assignment of $\Gamma$ then the result of $e$ under $S'$ and $\Gamma'$ is a super-value of the result of $e$ under $S$ and $\Gamma$.

**Proof.** We show this with induction upon the structure of $e$:

- Assume $e = \$x$ : By definition $\Gamma'($\$x$) is a super-value of $\Gamma($\$x$).
- Assume $e = ax::nt$ : Since $\Gamma'$ is a super-assignment of $\Gamma$ it holds that $\Gamma'$($\$dot$) = $\Gamma($\$dot$), and since $S$ is a projection of $S'$ it holds that the axis returns under $S'$ and a super-value of the value that is returned under $S$.
- Assume $e = ddo(e_1)$ : By induction the lemma holds for $e_1$, and clearly if $v$ is a sub-value of $e'$ then $ddo(v)$ is a sub-value of $ddo(e')$. 

3 Tree Pattern Decision for CXQ

We present an algorithm for deciding whether 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. Before we proceed, we need to discuss some preliminary lemmas and theorems that are required to show soundness and completeness for the decision algorithm.

**Lemma 3.1** (Unique result of CXQ). Given a store $S$, an assignment $\Gamma$ over $S$ and a CXQ expression $e$ there is exactly one value $v$ over $S$ such that $S, \Gamma \vdash e \Rightarrow v$.

**Proof.** Easily shown with induction upon the structure of $e$. \hfill \square

**Lemma 3.2** (Monotonicity of CXQ). For all stores $S, S'$, assignments $\Gamma, \Gamma'$ over $S$ and $S'$, respectively, and CXQ expressions $e$ it holds that if $S'$ is a super-store of $S$ and $\Gamma'$ is a super-assignment of $\Gamma$ then the result of $e$ under $S'$ and $\Gamma'$ is a super-value of the result of $e$ under $S$ and $\Gamma$.

**Proof.** We show this with induction upon the structure of $e$:

- Assume $e = \$x$ : By definition $\Gamma'($\$x$) is a super-value of $\Gamma($\$x$).
- Assume $e = ax::nt$ : Since $\Gamma'$ is a super-assignment of $\Gamma$ it holds that $\Gamma'$($\$dot$) = $\Gamma($\$dot$), and since $S$ is a projection of $S'$ it holds that the axis returns under $S'$ and a super-value of the value that is returned under $S$.
- Assume $e = ddo(e_1)$ : By induction the lemma holds for $e_1$, and clearly if $v$ is a sub-value of $e'$ then $ddo(v)$ is a sub-value of $ddo(e')$. 

5
Lemma 3.3 (Condition Satisfaction). Given a store $S$, an assignment $\Gamma$ over $S$ and a CXQ expression $e$ there is a super-store $S'$ of $S$, a super-assignment $\Gamma'$ of $\Gamma$ such that the result of $e$ under $S'$ and $\Gamma'$ is non-empty and if $\Gamma'(\$x) \neq \Gamma(\$x)$ then $\Gamma'(\$x) = \langle \rangle$ and $\Gamma'(\$x) = \{\}$.

Proof. We show this with induction upon the structure of $e$:

- Assume $e = if\ e_1 \ then\ e_2$ : By the semantics of $e$ it holds that if $S, \Gamma \vdash e \Rightarrow v$ then at least one of the following holds: (1) the result $e_1$ of $e$ under $S$ and $\Gamma$ is non-empty and $v$ is the result of $e_2$ under $S$ and $\Gamma$ and (2) the result of $e_1$ under $S$ and $\Gamma$ is empty and $v = \{\}$. If we assume (1) then, by induction it follows that the result of $e_1$ under $S'$ and $\Gamma'$ is non-empty and the result $v'$ of $e_2$ under $S'$ and $\Gamma'$ is a super-value of $v$. By the semantics of $e$ it then holds that $v'$ is the value of $e$ under $S'$ and $\Gamma'$. If we assume (2) then the result of $e$ under $S'$ and $\Gamma'$ is a super-value of $v$ because every value is a super-value of $\{\}$.

- Assume $e = for\ \$x\ in\ e_1\ return\ e_2$ : By the semantics of $e$ it holds that if $S, \Gamma \vdash e \Rightarrow v$ then there is a value $v_1 = \{n_1, \ldots, n_k\}$ such that $S, \Gamma[\$x \mapsto \langle n_i \rangle] \vdash e_2 \Rightarrow v_2, i$ for each $1 \leq i \leq k$ and $v = v_2, \ldots v_{2,k}$. By induction there is a super-value $v_1'$ of $v_1$ such that $S', \Gamma' \vdash e_1 \Rightarrow v_1'$. It also holds by induction that there are values $v_2', \ldots v_{2,k}'$ such that for each $1 \leq i \leq k$: (1) $S', \Gamma'[\$x \mapsto \langle n_i \rangle] \vdash e_2 \Rightarrow v_2'$, and (2) $v_2'$ is a super-value of $v_2'$. By Lemma 3.1 and the semantics of $e$ it follows that there is a value $v'$ such $S', \Gamma' \vdash e \Rightarrow v'$ and $v'$ is a super-value of $v_2', \ldots v_{2,k}'$ and, hence, also of $v = v_2, \ldots v_{2,k}$.

- Assume $e = let\ \$x := e_1\ end$ : By induction the result $v'$ of $e_1$ under $S'$ and $\Gamma'$ is a super-value of the result $v$ under $S$ and $\Gamma$. By induction it also follows that the result of $e_2$ under $S'$ and $\Gamma'[$\$x \mapsto v'\]$ is a super-value of the result $\Gamma[$\$x \mapsto v\]$.

- Assume $e = ddo(e_1)$ : By induction the lemma holds for $e_1$, and since $ddo(e_1)$ returns a non-empty result if $e_1$ returns a non-empty result, it also holds for $ddo(e_1)$.

- Assume $e = if\ e_1\ then\ e_2$ : By induction there is a super-store $S_1$ of $S$ and super-assignment $\Gamma_1$ of $\Gamma$ such that the result of $e_1$ is non-empty. By induction there also exist the super-store $S_2$ of $S_1$ and super-assignment $\Gamma_2$ of $\Gamma_1$ such that $e_2$ returns a non-empty result. Moreover, if $\Gamma_2$ differs from $\Gamma_1$ for $\$x$ then either $\Gamma_1(\$x) = \{\}$, in which case $\Gamma(\$x) = \{\}$, or $\Gamma_1(\$x) \neq \{\}$, in which case $\Gamma_2(\$x) = \Gamma_1(\$x)$ and, since $\Gamma_2$ differs from $\Gamma_1$ for $\$x$, so does $\Gamma_1$ and therefore $\Gamma'($\$x$) = $\{\}$. By Lemma 3.2 it holds that the result of $e_1$ under $S_2$ and $\Gamma_2$ is also non-empty, so the result of $e$ under $S_2$ and $\Gamma_2$ is indeed non-empty.

- Assume $e = for\ \$x\ in\ e_1\ return\ e_2$ : By induction there is a super-store $S_1$ of $S$ and super-assignment $\Gamma_1$ of $\Gamma$ such that the result of $e_1$ is non-empty. Let $n$ be a node in this result, then by induction there is a superstore $S_2$ of $S_1$ and super-assignment $\Gamma_2$ of $\Gamma_1[$\$x \mapsto \langle n \rangle]$ such that the result of $e_2$ under $S_2$ and $\Gamma_2$ is non-empty and $\Gamma_2(\$x) = \langle n \rangle$. By Lemma 3.2 it holds that the result of $e_1$ under $S_2$ and $\Gamma_2[$\$x \mapsto \Gamma_1(\$x)$] also contains $n$, so the result of $e$ under $S_2$ and $\Gamma_2[$\$x \mapsto \Gamma_1(\$x)$] is indeed non-empty. Moreover, if $\Gamma_2[\$x \mapsto \Gamma_1(\$x)$] differs from $\Gamma$ for $\$x$, then $\Gamma_1$ differs from $\Gamma$ for $\$x$ and therefore $\Gamma(\$x) = \{\}$. If, on the other hand, $\Gamma_2[\$x \mapsto \Gamma_1(\$x)$] differs from $\Gamma$ for $\$y \neq \$x$, then either $\Gamma_1(\$y) = \{\}$ and therefore also $\Gamma(\$y) = \{\}$, or $\Gamma_1(\$y) \neq \{\}$, in which case $\Gamma_2(\$y) = \Gamma_1(\$y)$ and, since $\Gamma_2$ differs from $\Gamma$ for $\$y$, so does $\Gamma_1$ and therefore $\Gamma(\$y) = \{\}$. 


• Assume $e = \text{let } $x := e_1 \text{ return } e_2$ : By induction there is a super-store $S_1$ of $S$ and super-assignment $\Gamma_1$ of $\Gamma$ such that the result of $e_1$ is non-empty. Let $v_1$ be this result, then by induction there is a superstore $S_2$ of $S_1$ and super-assignment $\Gamma_2$ of $\Gamma_1[\$x \mapsto v_1]$ such that the result of $e_2$ under $S_2$ and $\Gamma_2$ is a non-empty value $v_2$ and $\Gamma_2[\$x] = v_1$. By Lemma 3.2 it holds that the result of $e_1$ under $S_2$ and $\Gamma_2[\$x \mapsto \Gamma_1[\$x]]$ is a super-value $v'_1$ of $v_1$, and the result of $e_2$ under $S_2$ and $\Gamma_2[\$x \mapsto v'_1]$ is a super-value $v'_2$ of $v_2$, and so the result of $e$ under $S_2$ and $\Gamma_2[\$x \mapsto \Gamma_1[\$x]]$ is indeed non-empty. Moreover, if $\Gamma_2[\$x \mapsto \Gamma_1[\$x]]$ differs from $\Gamma$ for $\$x$ then $\Gamma_1$ differs from $\Gamma$ for $\$x$ and therefore $\Gamma[\$x] = \emptyset$. If, on the other hand, $\Gamma_2[\$x \mapsto \Gamma_1[\$x]]$ differs from $\Gamma$ for $\$y \neq \$x$ then either $\Gamma_1[\$y] = \emptyset$ and therefore also $\Gamma[\$y] = \emptyset$, or $\Gamma_1[\$y] \neq \emptyset$, in which case $\Gamma_2[\$y] = \Gamma_1[\$y]$ and, since $\Gamma_2$ differs from $\Gamma$ for $\$y$, so does $\Gamma_1$ and therefore $\Gamma[\$y] = \emptyset$.

\[\square\]

Just as for pure XPath expressions [Fernández et al., 2005], it is possible to determine some static properties for CXQ expressions and their values that assist in deciding whether that expression returns ordered and duplicate-free results. A property $\pi$ holds for an expression $e$ if for any store $S$ and any variable assignment $\Gamma$, s.t. $S, \Gamma \vdash e \Rightarrow v$, the list of nodes in $v$ satisfies $\pi$.

**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 $\pi$ then we denote this as $v : \pi$. The semantics of these properties is defined as follows:

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

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

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 3.3** (Expression property). We say that a CXQ expression $e$ has property $\pi$ under the property table $T$, denoted as $T \vdash e : \pi$, if it holds for every store $S$ and every value assignment $\Gamma$ 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:

- $rv(\$x) = \$x$
- $rv(a::n) = \$dot$
- $rv(ddo(e)) = rv(e)$
- $rv(if \ e_1 \ then \ e_2) = rv(e_2)$
- $rv(for \ \$x \ in \ e_1 \ return \ e_2) = \left\{ \begin{array}{ll} \text{rv}(e_1) & \text{if rv}(e_2) = \$x \\ \text{rv}(e_2) & \text{if rv}(e_2) \neq \$x \end{array} \right.$

\[7\]
Theorem 3.1. For every CXQ expression $e$, there is a CXQ expression $e'$ such that (1) $\text{ddo}(e) \equiv \text{ddo}(e')$ and (2) $FV(e') = \{rv(e)\}$ or $e'$ is of the form $e_1'$ then $e_2'$ such that $FV(e_2') = \{rv(e)\}$.

This will be proven formally in Section 4.

### 3.1 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 premise $rv(e_2) \neq \$x$, 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 $ddo($

The meaning of $rv(e)$ can be made more precise by the following claim:

Let $\pi$ be a mapping from variables to $\{\text{ord}, \text{gen}, \text{nodup}\}$, and let $\text{for } x \text{ in } d \text{ do } e \Rightarrow e'$ be a XQuery expression. Then $\text{rv}(e')$ is defined as follows:

$$rv(e') = \begin{cases} 
\text{ord} & \text{if } e' \text{ contains an ordering} \\
\text{gen} & \text{if } e' \text{ contains a generation} \\
\text{nodup} & \text{if } e' \text{ contains no duplicates}
\end{cases}$$
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 3.2** (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.

The proof of this theorem is given in the following subsection.

### 3.2 Proof of Soundness and Completeness

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.

#### 3.2.1 Variable

Soundness for Var follows from the fact that $\Gamma(x)$ satisfies the properties in $T(x)$. Soundness for VarClos holds since if all the nodes in $\Gamma(x)$ are the same then $x$ is sorted in document order and all the nodes belong to the same generation. Soundness for Dot holds since every assignment maps $\dot{x}$ to a single node.

For completeness we consider each property:

**no2d** If no2d is not derived then $x \neq \dot{x}$ and $T(x)$ is a subset of $\{gen, ord, nodup\}$ and it is easy to construct a store $S$ and a value $v$ over $S$ such that $v : \pi$ for each $\pi \in T(x)$ but not $v : no2d$ as follows. In $S$ a tree with root $n_1$ and two children $n_2$ and $n_3$, and $v = \langle n_2, n_3 \rangle$.

**gen** If gen is not derived then $x \neq \dot{x}$ and $T(x)$ is a subset of $\{ord, nodup\}$ and it is easy to construct a store $S$ and a value $v$ over $S$ such that $v : \pi$ for each $\pi \in T(x)$ but not $v : gen$ as follows. In $S$ a tree with root $n_1$ and child $n_2$, and $v = \langle n_1, n_2 \rangle$. 


ord If ord is not derived then $x \neq \text{dot}$ and $T(x)$ is a subset of $\{\text{gen}, \text{nodup}\}$ and it is easy to construct a store $S$ and a value $v$ over $S$ such that $v : \pi$ for each $\pi \in T(x)$ but not $v : \text{ord}$ as follows. In $S$ a tree with root $n_1$ and two children $n_2$ and $n_3$, and $v = (n_3, n_2)$.

nodup If nodup is not derived then $x \neq \text{dot}$ and $T(x)$ is a subset of $\{\text{no2d}, \text{gen}, \text{ord}\}$ and it is easy to construct a store $S$ and a value $v$ over $S$ such that $v : \pi$ for each $\pi \in T(x)$ but not $v : \text{nodup}$ as follows. In $S$ a tree with root $n_1$, and $v = (n_1, n_1)$.

### 3.2.2 Step Expression

Soundness of DDOStep follows from the fact that after each step of a path expression the result is sorted in document order and duplicates are removed. The soundness of ChildStep follows from the child axis preserves the gen property.

For completeness we consider each property:

no2d The property no2d is never derived, which is correct since we can construct a store $S$ and an assignment $\Gamma$ that maps $\text{dot}$ to a value $v$ over $S$ and satisfies all tables $T$ but the result of $a:n$ under $\Gamma$ does not have property no2d as follows. In $S$ a tree with root $n_1$ and two children $n_2$ and $n_3$, and $v = (n_1)$.

gen For the axes desc and d-o-s the property gen is never derived, which is correct since we can construct a store $S$ and an assignment $\Gamma$ over $S$ that maps $\text{dot}$ to a value $v$ over $S$ and satisfies all tables $T$ but the result of $a:n$ under $\Gamma$ does not have property gen as follows. In $S$ a tree with root $n_1$ and a child $n_2$ which again has a child $n_3$, and $v = (n_1)$. For the axis child the property gen is always derived.

ord The property ord is always derived.

nodup The property nodup is always derived.

### 3.2.3 The ddo operation

Since no2d and gen are properties of the set of nodes in a sequence and this set is not changed by the ddo operation it follows that DdoSet is sound and complete for these properties. Since the result of the ddo operation is always sorted in document order and without duplicates, the rule DdoSeq is also sound and complete.

### 3.2.4 The if expression

Soundness of If follows from that fact that the expression either returns the result of $e_2$, which has property $\pi$ by induction, or the empty sequence, which has all properties in the mentioned set.

Completeness of the rule If follows from two observations: The first is that, as shown by Lemma 3.3, given a store $S$ and assignment $\Gamma$ over $S$ that satisfies $T$ we can always construct a super-store $S'$ and a super-assignment $\Gamma'$ over $S'$ such that $e_1$ returns a non-empty result and $\Gamma'$ also satisfies $T$. The second observation, as shown by Lemma 3.2, is that the result of $e_2$ under $S'$ and $\Gamma'$ will be a super-value of the result under $S$ and $\Gamma$. Therefore, since the properties no2d, gen, ord and nodup are monotonic in the sense that if they do not hold for a value $v$ then they also do not for all super-values of $v$, it holds that if they did not hold for the result of $e_2$ under $S$ and $\Gamma$ then they also not hold for the result of $e$ under $S'$ and $\Gamma'$.

### 3.2.5 The let expression

Soundness of the rule Let can be shown as follows. By induction we know that for a store $S$ and assignment $\Gamma$ over $S$ that satisfies a property table $T$ all properties in $P$ hold for the result $v$ of $e_1$. It follows that the assignment $\Gamma[\$x \mapsto v]$ satisfies the property table $T[\$x \mapsto P]$, and by induction
it follows that if we derive $T[^{\$x \mapsto P}] \vdash e_2 : \pi$ then $\pi$ indeed holds for the result of $e_2$ under $S$ and $T[^{\$x \mapsto e}]$, and therefore for the result of $e$.

Completeness of Let is based on two observations. The first is that for CXQ expressions it holds that $(\text{let } \$x := e_1 \text{ return } e_2) \equiv (e_2[^{\$x/e_1}])$ where $e_2[^{\$x/e}]$ is expression $e_2$ with all free occurrences of $\$x$ replaced with $e_1$. The second observation is that for every property $\pi$ it holds that the rules derive $(\text{let } \$x := e_1 \text{ return } e_2 : \pi)$ if they derive that $e_2[^{\$x/e_1}] : \pi$. It follows that if the rules without the rule Let are complete for CXQ without let expressions then the rules with Let are complete for CXQ.

### 3.2.6 The for expression

We consider each property separately:

**Property no2d.** Soundness of ForSetRv for no2d follows by induction and Lemma 3.1. Indeed for the node in the result of $e_1$, if it exists, the result of $e_2$ contains always the same node if it is non-empty. Soundness of ForSetNoRv for no2d follows also by induction and Lemma 3.1 and Lemma 3.1. The two lemmas together show that for every possible value of $\$x$ the result of $e_2$ contains the same node if it is non-empty.

Completeness: if no2d is not derived for a for expression, then from the inference rules the following should be true

$$(T \vdash e_1 : \neg \text{no2d} \lor T[^{\$x \mapsto \{\text{no2d}, \text{nodup}\}}] \vdash e_2 : \neg \text{no2d})$$

which is equivalent to

$$(\text{rv}(e_2) \neq \$x \lor T[^{\$x \mapsto \{\text{no2d}, \text{nodup}\}}] \vdash e_2 : \neg \text{no2d})$$

where the second and third conjunctions are subsumed by the last condition. Thus we need to show that for some instance $S$ and an assignment $\Gamma$ over $S$ that satisfies $T$:

- for $\$x$ in $e_1$ return $e_2 : \neg \text{no2d}$
- $T[^{\$x \mapsto \{\text{no2d}, \text{nodup}\}}] \vdash e_2 : \text{no2d}$

where $\text{rv}(e_2) = \$x \lor T \vdash e_1 : \neg \text{no2d}$

If $e_1 : \text{no2d}$, then by Lemma 3.1, it follows that the for expression also has the no2d property, which is a contradiction. On the other hand, if $\text{rv}(e_2) \neq \$x$, then it follows that the result of $e_2$ is repeated one or more times, which does not affect the no2d property. So once again, the for expression has the no2d property, which is a contradiction.

**Property gen.** Soundness of ForSetRv for gen follows by induction, Lemma 3.1, Lemma 3.1 and the observation that if gen is derived for a CXQ expression $e$ with a single free variable $\$x$ then the set of nodes in the result of $e$ is a subset of the set of nodes in the result of $\$x[/child:*]n for some $n \geq 0$. Soundness of ForSetNoRv for gen follows also by induction and Lemma 3.1 and Lemma 3.1. The two lemmas together show that for every possible value of $\$x$ the result of $e_2$ contains the same set of nodes if it is non-empty.

Completeness: Analogous to no2d, we need to show that

- for $\$x$ in $e_1$ return $e_2 : \neg \text{gen}$
- $T[^{\$x \mapsto \{\text{no2d}, \text{nodup}\}}] \vdash e_2 : \text{gen}$

where $\text{rv}(e_2) = \$x \lor T \vdash e_1 : \neg \text{gen}$

11
Lemma 3.5. For every instance \( \text{gen} \) has the \( x \) expression \( e \) with a single free variable \( \$x \) then the set of nodes in the result of \( e \) is a subset of the set of nodes in the result of \( \$x/(\text{child}:*)^n \) for some \( n \geq 0 \), it follows that the \( \forall \) expression also has the \( \text{gen} \) property, which is a contradiction. On the other hand, if \( rv(e_2) \neq \$x \), then it follows that the result of \( e_2 \) is repeated one or more times, which does not affect the \( \text{gen} \) property. So once again, the \( \forall \) expression has the \( \text{gen} \) property, which is a contradiction.

**Property ord.**

Lemma 3.4. For every instance \( S \) and an assignment \( \Gamma \) over \( S \) that satisfies \( T \), if for a CXQ expression \( e \) it holds that \( rv(e) = \$x \), where \( T \vdash \$x : \text{no2d,nodup} \) and the rules derive that \( T \vdash e : \text{no2d} \) then the result of \( e \) will either be empty or equal to \( \$x \).

**Proof.** By induction on the subexpressions of \( e \).

- Suppose \( e \) is a step: This is impossible, because there are no rules that derive \( \text{no2d} \) for a step expression;
- Suppose \( e \) is a variable reference: In that case, since \( rv(e) = \$x \), \( e \) must be \( \$x \) itself. Obviously, only \( \$x \) is returned here.
- Suppose \( e \) is if \( e_1 \) then \( e_2 \): We know that the lemma holds for \( e_1 \) and \( e_2 \) separately. It is easy to see that, depending on the outcome of \( e_1 \) and since \( rv(e) = \$x \), \( e_2 \) will either return the node in \( \$x \) or the empty sequence.
- Suppose \( e \) is for \( \$l \) in \( e_1 \) return \( e_2 \). If \( rv(e_2) = \$l \) then \( rv(e_1) = \$x \), but since \( T \vdash e_2 : \text{no2d} \), by induction it can only return either \( \$l \), which contains the node in \( \$x \) or the empty sequence. If \( rv(e_2) \neq \$l \) then \( rv(e_2) = \$x \) and \( e_2 \) by induction only returns nodes from \( \$x \).
- Suppose \( e \) is let \( \$l := e_1 \) return \( e_2 \). If \( rv(e_2) = \$l \) then \( rv(e_1) = \$x \), but since \( T \vdash e_2 : \text{no2d} \), by induction it can only return either \( \$l \), which contains the node in \( \$x \) or the empty sequence. If \( rv(e_2) \neq \$l \) then \( rv(e_2) = \$x \) and \( e_2 \) by induction only returns nodes from \( \$x \).

Lemma 3.5. For every instance \( S \) and an assignment \( \Gamma \) over \( S \) that satisfies \( T \), if for a CXQ expression \( e \) it holds that \( rv(e) = \$x \), where \( T \vdash \$x : \text{no2d,nodup} \) then the set of nodes in the result of \( e \) will be a subset of the set of nodes in the result of \( \$x/d-o-s:* \).

**Proof.** By induction on the subexpressions of \( e \).

- Suppose \( e \) is a step or a variable reference, then the Lemma holds, since any step in CXQ only selects descendants of \( \$x \).
- Suppose \( e \) is if \( e_1 \) then \( e_2 \): It is easy to see that since \( rv(e) = \$x \), \( e_2 \) by induction returns a subset of the result of \( \$x/d-o-s:* \), possibly the empty sequence depending on the outcome of \( e_1 \).
- Suppose \( e \) is for \( \$l \) in \( e_1 \) return \( e_2 \). If \( rv(e_2) = \$l \) then \( rv(e_1) = \$x \), then by induction \( e_2 \) returns only descendants of \( \$x \), possibly including \( \$x \) or the empty sequence. If \( rv(e_2) \neq \$l \) then \( rv(e_2) = \$x \) and the same holds.
- Suppose \( e \) is let \( \$l := e_1 \) return \( e_2 \). If \( rv(e_2) = \$l \) then \( rv(e_1) = \$x \), then by induction \( e_2 \) returns only descendants of \( \$x \) possibly including \( \$x \), or the empty sequence. If \( rv(e_2) \neq \$l \) then \( rv(e_2) = \$x \) and the same holds.

\[\square\]
The \textit{ord} property is derived by four rules: \texttt{ForOrdRv1}, \texttt{ForOrdRv2}, \texttt{ForOrdNoRv1} and \texttt{ForOrdNoRv2}. Soundness of \texttt{ForOrdRv1} follows by induction and the fact that if \(rv(e_2) = \$x\) and the rules derive that \(T[\$x \mapsto \{\text{no2d, nodup}\}] \vdash e_2 : \text{no2d}\) then the set of nodes in the result of \(e_2\) is a subset of the set of nodes in the result of \(\$x\), see Lemma 3.4. Soundness of \texttt{ForOrdRv2} follows by induction and the fact that if \(rv(e_2) = \$x\) then the set of nodes in the result of \(e_2\) is a subset of the set of nodes in the result of \(\$x/d-o-s:*\), see Lemma 3.5. Soundness of \texttt{ForOrdNoRv1} follows by induction and Lemma 3.1 and Lemma 3.1. Indeed, the result of \(e_2\) will, if it is non-empty, contain the same node for each node in the result of \(e_1\). Soundness of \texttt{ForOrdNoRv2} follows by induction and the fact that the result of \(e_1\) will be empty or a singleton sequence.

Completeness: From the inference rules \texttt{ForOrdRv1}, \texttt{ForOrdRv2}, \texttt{ForOrdNoRv1} and \texttt{ForOrdNoRv2}, we need to show that

\[
\begin{align*}
(rv(e_2) & \neq \$x \lor e_1: \neg\text{ord} \lor e_2: \neg\text{ord}) \\
\land (rv(e_2) & \neq \$x \lor e_1: \neg\text{ord} \lor e_1: \neg\text{gen} \lor e_1: \neg\text{nodup} \lor e_2: \neg\text{ord}) \\
\land (rv(e_2) & = \$x \lor e_2: \neg\text{no2d}) \\
\land (rv(e_2) & = \$x \lor e_1: \neg\text{no2d} \lor e_1: \neg\text{nodup} \lor e_2: \neg\text{ord}) \\
\implies & (\text{for } \$x \text{ in } e_1 \text{ return } e_2): \neg\text{ord}
\end{align*}
\]

This can be rewritten as

\[
\begin{align*}
(rv(e_2) & \neq \$x \land e_1: \neg\text{no2d} \land e_2: \neg\text{no2d}) \\
\lor (rv(e_2) & = \$x \land e_1: \neg\text{ord}) \lor (e_1: \neg\text{ord} \land e_2: \neg\text{no2d}) \\
\lor (e_1: \neg\text{gen} \land e_2: \neg\text{no2d}) \lor (e_1: \neg\text{nodup} \land e_2: \neg\text{no2d}) \\
\lor e_2: \neg\text{ord} \\
\implies & (\text{for } \$x \text{ in } e_1 \text{ return } e_2): \neg\text{ord}
\end{align*}
\]

\begin{itemize}
  \item if \(rv(e_2) \neq \$x \land e_1: \neg\text{no2d} \land e_2: \neg\text{no2d}\) then, since the result of \(e_1\) can contain multiple nodes, the result of \(e_2\) - also having multiple nodes - will possibly be concatenated multiple times resulting in an out of order sequence;
  \item if \(rv(e_2) = \$x \land e_1: \neg\text{ord}\) then either the nodes bound to \(\$x\) or a subset of the nodes in the results of \(\$x/d-o-s:*\) are returned. If the nodes in the result of \(e_1\) are out of document order, then so is the concatenation of the corresponding sets of descendants;
  \item if \(e_1: \neg\text{ord} \land e_2: \neg\text{no2d}\), then there are two possibilities: if \(rv(e_2) = \$x\), then the problem is a special case of the previous one, otherwise, since \(e_1: \neg\text{ord}\), the result of \(e_1\) may contain several nodes, causing the result of \(e_2\) to be concatenated several times;
  \item if \(e_1: \neg\text{gen} \land e_2: \neg\text{no2d}\): analogous to the previous case;
  \item if \(e_1: \neg\text{nodup} \land e_2: \neg\text{no2d}\): analogous to the previous case;
  \item if \(e_2: \neg\text{ord}\) then the result is trivially out of document order.
\end{itemize}

Property \textit{nodup}. The \textit{nodup} property is derived by three rules: \texttt{ForNodupRv1}, \texttt{ForNodupRv2} and \texttt{ForNodupNoRv}. The soundness of \texttt{ForNodupRv1} follows by induction and the fact that if \(rv(e_2) = \$x\) then the set of nodes in the result of \(e_2\) is a subset of the set of nodes in the result of \(\$x/d-o-s:*\). The soundness for \texttt{ForNodupRv2} follows from the fact that if a set of nodes contain no duplicates, then the set of nodes, obtained from taking all the children from the nodes in the first set, will aslo be free of duplicates. The soundness of \texttt{ForNodupNoRv} follows by induction and the fact that the result of \(e_1\) will be empty or a singleton sequence.

Completeness: From the inference rules \texttt{ForNodupRv1}, \texttt{ForNodupRv2} and \texttt{ForNodupNoRv}, we need to show that

\[
\begin{align*}
(rv(e_2) & \neq \$x \lor e_1: \neg\text{nodup} \lor e_1: \neg\text{gen} \lor e_2: \neg\text{nodup}) \\
\land (rv(e_2) & \neq \$x \lor e_1: \neg\text{nodup} \lor e_2: \neg\text{nodup} \lor e_2: \neg\text{gen})
\end{align*}
\]
\( \land (rv(e_2) = \$x \lor e_1 : \neg\text{no2d} \lor e_1 : \neg\text{nodup} \lor e_2 : \neg\text{nodup}) \)
\( \Rightarrow \) (for \$x in e_1 return e_2) : \neg\text{nodup} \\

which can be rewritten as
\( (rv(e_2) \neq \$x \land e_1 : \neg\text{no2d}) \lor (e_1 : \neg\text{gen} \land e_1 : \neg\text{gen}) \)
\( \lor e_1 : \neg\text{nodup} \lor e_2 : \neg\text{nodup} \)
\( \Rightarrow \) (for \$x in e_1 return e_2) : \neg\text{nodup} \\

- if \( rv(e_2) \neq \$x \land e_1 : \neg\text{no2d} \), then since the result of \( e_1 \) can contain multiple nodes, the result of \( e_2 \) will be repeated multiple times;
- if \( e_1 : \neg\text{gen} \land e_1 : \neg\text{gen} \) then the \( \lor \) expression can contain duplicates because (1) if \( \neg\text{gen} \) is derived for an expression, then this is due to the following of one or more \text{desc} or \text{desc-or-self} steps (2) thus, the result of \( e_1 \) will contain ancestor-descendant related nodes, (3) for such nodes, following either the \text{desc} or \text{desc-or-self} step again, will result in duplicates;
- if \( e_1 : \neg\text{nodup} \) then since \( e_1 \) contains multiple nodes, the result of the \( \lor \) expression can contain duplicates, even if \( rv(e_2) \neq \$x \);
- if \( e_2 : \neg\text{nodup} \) then the result of the \( \lor \) expression can trivially contain duplicates.

From the definition of the semantics of forest patterns it is clear that the \text{ord} and \text{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 and therefore the following theorem holds.

**Theorem 3.3.** A CXQ expression is equivalent with a forest pattern iff it has the \text{ord} and \text{nodup} properties.

The proof of this theorem proceeds by demonstrating that every CXQ expressions of the form \text{ddo}(e) can be rewritten to a normal form that directly corresponds to and is equivalent with certain forest patterns. Because of this theorem the presented inference rules are effectively an 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 \text{ord} and \text{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 \text{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 \( \alpha \) is denoted as \( \alpha e \) with \( \alpha \) either \( \cdot \), \( \cup \) or \( \lor \) which represent the \text{list concatenation}, \text{set union} and \text{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 and duplicates may be added and removed (for \( \cup \)), and the result may be changed as long as it stays non-empty (for \( \lor \)). We use variables \( \alpha, \beta \), 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 \( \alpha \) we define an \textit{interpretation function} \( I^\alpha \) 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) \( I^\lor(x) \) is \text{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 $\alpha$ 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 $\alpha$-equivalent sequence. Note that this leads to a non-deterministic semantics.

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

**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 $^{\alpha}s[x/\beta]e = \gamma e$ with $\gamma$ the maximum of $\alpha$ and $\beta$.

### 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 $\text{ddo}$ 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 $X^\cup$ for output patterns and $X^\lor$ for condition patterns:

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

2. $X^\alpha(\{t\}) = T^\alpha(t)$

and $T^\cup$ for output trees and $T^\lor$ for condition trees:

1. $T^\cup(l^{\text{out}}) = \cup l$

2. $T^\lor(l) = \lor l$

3. $T^\cup(l^{\text{out}}(t_1, \ldots, t_n)) = \cup l$ for $\text{dot}$ in $\cup l$ return $X^\cup(\{t_3, \ldots, t_n, \text{dot}^{\text{out}}\})$
   if $n > 0$ and $l \neq \text{dot}$

4. $T^\lor(l(t_1, \ldots, t_n)) = ^\alpha$ for $\text{dot}$ in $\cup l$ return $X^\alpha(\{t_1, \ldots, t_n\})$
   if $n > 0$ and $l \neq \text{dot}$

5. $T^\lor(\text{dot}(t)) = T^\alpha(t)$

Recall that nodes labeled $\text{dot}$ have no children if they are output node, and one child if they are not, so $T^\cup$ and $T^\lor$ are indeed defined for all output trees and condition trees, respectively. Observe that $X^\cup$ preserves the semantics of the expression. Also observe that it is non-deterministic since it picks an order for the subtrees.

**Example 4.1.** We now illustrate the mapping $X$ with a simple example. Consider the following output patterns:

```xml
$sx \{ desc::person 
  child::emailaddress,
  child::name^{out} 
  child::middlename } }$
```

This expression is mapped by $X$ to CXQ$^+$ as follows:
Theorem 4.1. \[ \text{Proof.} \]
We prove with induction upon the structure of $x$ for some non-empty sequence $x$.

1. $X^\cup\{t_1, \ldots, t_n\} = \cup l$ if $T^V(t_1)$ then $X^\cup\{t_2, \ldots, t_n\}$ if $n > 1$ and there is no output node in $t_1$. A node $n'$ is in the result of $\{t_1, \ldots, t_n\}$ if there is an embedding that maps the output node of $\{t_1, \ldots, t_n\}$ to $v'$. This is true iff there is an embedding of $t_1$ and $v'$ is in the result of $\{t_2, \ldots, t_n\}$. By induction it holds that this is true iff $T'(t_1)$ returns a non-empty sequence and $v'$ is in the result of $X^\cup\{t_1, \ldots, t_n\}$. This is true iff $v'$ is in the result of $\cup l$ if $T'(t_1)$ then $X^\cup\{t_2, \ldots, t_n\}$.

2. $X^\cup\{t_1, \ldots, t_n\} = \cup l$ if $T^V(t_1)$ then $X^\cup\{t_2, \ldots, t_n\}$ if $n > 1$ and there is no output node in $t_1$. There is an embedding of $\{t_1, \ldots, t_n\}$ if there is an embedding of $t_1$ and of $\{t_2, \ldots, t_n\}$. By induction it holds that this is true iff both $T'(t_1)$ and $X^\cup\{t_2, \ldots, t_n\}$ return a non-empty sequence. This is true iff $\cup l$ if $T'(t_1)$ then $X^\cup\{t_2, \ldots, t_n\}$ returns a non-empty sequence.

3. $X^\cup\{t\} = T^\cup(t)$. Follows by induction.

4. $X^\cup\{t\} = T^V(t)$. Follows by induction.

Consider the cases for $T^\alpha$:

1. $T^\cup(l(n)) = \cup l$. If $l$ is of the form $\ldots$ then it is clear that a node $n'$ is in the result of $\cup l$ if there is an embedding of $\ldots$ that maps $n$ to $n'$. If $l$ is of the form $\ldots$ then node $n'$ is in the result of $\ldots$ if $n'$ is labeled with $nt$ if $nt$ is a node name and has in $S$ the $ax$ relationship with $\Gamma$. The latter is true iff there is an embedding of $\ldots$ that maps $n$ to $n'$.

2. $T^V(l(n)) = \cup l$. If $l$ is of the form $\ldots$ then it is clear that the result of $\ldots$ is non-empty if there is an embedding of $\ldots$. If $l$ is of the form $\ldots$ then the result of $\ldots$ is non-empty iff there is a node $n'$ in $S$ that is labeled with $nt$ if $nt$ is a node name and has the $ax$ relationship with $\Gamma$. The latter is true iff there is an embedding of $ax::nt^\alpha(n)$ that maps $n$ to $n'$.

3. $T^\cup(l'(n)) = \cup l$ if $l'(n)$ is in the result of $\cup l$ return $\ldots$ if $n > 0$ and $l \neq \ldots$. Consider the proposition that node $v'$ in $S$ is in the result of $\ldots$ that maps $n$ to $n'$. This is true iff there is an embedding of $\ldots$ that maps $n$ to $n'$. This is true iff $n'$ is in the output of $\ldots$ and there is an embedding for $\{t_1, \ldots, t_n\}$ under the assignment $\Gamma' = \Gamma[\ldots \mapsto n']$. This is true iff $n'$ is in the output of $\ldots$ and in the output of $\{t_1, \ldots, t_n, \ldots\}$ under $\Gamma'$. By induction this is true iff $n'$ is in the result of $\cup l$ and $\ldots$ under $\Gamma'$. This, then, is true iff $n'$ is in the result of $\cup l$ return $\ldots$.

4. $T^\cup(l(n)) = \cup l$ if $l(n)$ is in the result of $\cup l$ return $\ldots$ if $n > 0$ and $l \neq \ldots$. Consider the proposition that node $v'$ in $S$ is in the result of $\ldots$. This is true iff
there is an embedding of $l(n)\{t_1,\ldots,t_n\}$ that maps the output node to $n'$. This is true iff there is a node $n''$ in the output of $l^{\text{out}}(n)$ such that $n'$ is in the result of $\{t_1,\ldots,t_n\}$ under the assignment $\Gamma' = \Gamma[\text{fp} \mapsto n'']$. By induction this is true iff there is a node $n''$ in the result of $\cup l$ such that $n'$ is in the result of $\mathcal{X}'(\{t_1,\ldots,t_n\})$ under $\Gamma'$. This, then, is true iff $n'$ is in the result of $\cup \text{for}$ $\text{fp}$ in $\cup l$ return $\mathcal{X}'(\{t_1,\ldots,t_n\})$.

5. $T'\left(l(l(n)\{t_1,\ldots,t_n\})\right) = \forall \text{fp} \in \cup l \text{ return } \mathcal{X}'(\{t_1,\ldots,t_n\})$ if $n > 0$ and $l \neq \text{fp}$. This proof proceeds similar to the previous point except that the fact that $n'$ is in the result is replaced with the fact that the result is non-empty.

6. $T'\left(\text{fp}(n)\{t\}\right) = T'(t)$. Consider the proposition that there is an embedding of $\text{fp}(n)\{t\}$ that maps the output node to $n'$. Since the root of $t$ must be labeled with a label of the form $ax::nt$ this is true iff there is an embedding of $t$ that maps its output node to $n'$. By induction it follows that this is true iff $n'$ is in the result of $T'(t)$.

7. $T'\left(\text{fp}(n)\{t\}\right) = T'(t)$. This proof proceeds similar to the previous point except that the fact that $n'$ is in the result is replaced with the fact that the result is non-empty.

We proceed with defining the syntax of the CXQ$^+$ fragment onto which forest patterns are mapped by $\mathcal{X}'$. We will attempt to define this syntax such that (1) for every forest pattern the result of $\mathcal{X}'$ is in the syntax and (2) for every expression in the syntax there is forest pattern that is mapped to it. That (1) holds can be readily observed by noting that the following holds for the non-terminals: $\text{fp}$ describes the range of $\mathcal{X}'$, $\text{tp}$ describes the range of $T'$, $\text{otp}$ describes the range of $T'$, $\text{atp}$ describes the range of $T'$ restricted to trees with a $\text{fp}$ root, and $\text{aotp}$ describes the range of $T'$ restricted to trees with other roots.

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

\[
\begin{align*}
\text{fp} &::= \text{otp} \mid \forall \text{if } \text{tp} \text{ then } \text{fp} \\
\text{tp} &::= \text{otp} \mid \forall \text{x} \\
&\quad \forall \text{fp} \text{ in } \forall \text{x} \text{ return } \text{rc} \\
\text{atp} &::= \forall \text{ax::nt} \\
&\quad \forall \text{fp} \text{ in } \forall \text{ax::nt} \text{ return } \text{rc} \\
\text{rc} &::= \text{atp} \mid \forall \text{if } \text{atp} \text{ then } \text{rc} \\
\text{aotp} &::= \forall \text{fp} \text{ in } \forall \text{ax::nt} \text{ return } \text{orc} \\
\text{orc} &::= \text{aotp} \mid \forall \text{if } \text{atp} \text{ then } (\text{orc} \mid \forall \text{fp})
\end{align*}
\]

where $\forall \text{x}$ refers to the set of variables minus $\text{fp}$.

### 4.1.2 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 $\mathcal{X}'$. This is defined by the mapping $\mathcal{F}$ that maps subexpressions of TPNF expressions to forest patterns such that expressions associated with $\text{tp}$, $\text{atp}$ and $\text{rc}$ are mapped to a condition pattern, and expressions associated with $\text{fp}$, $\text{otp}$ and $\text{aotp}$ are mapped to an output pattern:

1. $\mathcal{F}(\forall \text{x}) = x^{\text{out}}$
2. $\mathcal{F}(\forall \text{x}) = x$
3. $\mathcal{F}(\forall \text{a::n}) = \text{fp}\{a::n^{\text{out}}\}$
4. $\mathcal{F}(\forall \text{a::n}) = \text{fp}\{a::n\}$
5. $\mathcal{F}(\forall \text{a if } \forall e_1 \text{ then } \forall e_2) = \mathcal{F}(\forall e_1) + \mathcal{F}(\forall e_2)$
6. $\mathcal{F}(\forall \text{fp} \text{ in } \forall \text{x} \text{ return } \forall e_1) = \{x\{t_1,\ldots,t_n\}\}$
   \[
   \text{if } \mathcal{F}(\forall e_1) = \{\text{fp}\{t_1\},\ldots,\text{fp}\{t_n\}\}
   \]
7. $F(\cup \text{for } \text{dot in } \cup \text{x return } \cup e_1) = \text{x}^{\text{out}}\{t_1, \ldots, t_n\}$
   if $F(e_1) = \{\text{dot}(t_1), \ldots, \text{dot}(t_n), \text{dot}^{\text{out}}\}$

8. $F(\alpha \text{for } \text{dot in } \alpha: n \text{ return } \alpha e_1) = \text{dot}\{a: n\{t_1, \ldots, t_n\}\}$
   if $F(e_1) = \{\text{dot}(t_1), \ldots, \text{dot}(t_n)\}$

9. $F(\cup \text{for } \text{dot in } \cup a: n \text{ return } \cup e_1) = \text{dot}\{a: n^{\text{out}}\{t_1, \ldots, t_n\}\}$
   if $F(e_1) = \{\text{dot}(t_1), \ldots, \text{dot}(t_n), \text{dot}^{\text{out}}\}$

Observe that $F$ 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 orc the result of $F$ is always of the form $\{\text{dot}(t_1), \ldots, \text{dot}(t_n)\}$ and the result of an expression associated with the nonterminal ort is of this form or of the form $\{\text{dot}(t_1), \ldots, \text{dot}(t_n), \text{dot}^{\text{out}}\}$.

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

**Theorem 4.2.** The function $F$ is the inverse of $X^\cup$, i.e., for every expression $\cup e$ in TPNF it holds that $F(\cup e) = f$ if and only if $X^\cup(f) = \cup e$.

**Proof.** We prove with induction upon the abstract syntax tree of the TPNF expression that it holds for each subexpression $\alpha e$ of a TPNF expression it holds that $F(\alpha e) = f$ if $X^\cup(f) = \alpha e$.

Observe that for expressions associated with the nonterminals $fp$, $otp$ and $aotp$ it holds that $\alpha = \cup$ and for the nonterminals $tp$ and $atp$ it is $\exists$.

We first consider the cases for $fp$:

- Assume $fp = otp$. Then by induction $F(\cup \text{for } \text{dot in } \cup \text{x return } \cup e_1) = f$ if $X(f) = otp$.
- Assume $fp = \cup \text{if } \text{tp}_1 \text{ then } \text{fp}_2$. Then $F(\cup \text{if } \text{tp}_1 \text{ then } \text{fp}_2) = f$ if and only if there are $f_1$ and $f_2$ such that $f_1 = F(\text{tp}_1)$ and $f_2 = F(\text{fp}_2)$ and $f_1$ is a condition pattern with one root, $f_2$ is an output pattern and $f = f_1 + f_2$. By induction it holds that $F(\text{tp}_1) = f_1$ and $X^\cup(f_1) = \text{tp}_1$ and $F(\text{fp}_2) = f_2$ if $X^\cup(f_2) = \text{fp}_2$. It follows by the definition of $X$ that this is then true if $X^\cup(f) = \cup(\text{tp}_1)$ then $X^\cup(f_2)$.

Next we consider the cases for $otp$:

- Assume $otp = aotp$. By induction $F(\cup \text{for } \text{dot in } \cup \text{x return } \cup e_1) = f$ if $X(f) = aotp$.
- Assume $otp = \cup \text{x such that } \text{x} \neq \text{dot}$. Then $F(\cup \text{for } \text{x such that } \text{x} \neq \text{dot}) = \cup \text{x and } X^\cup(\cup \text{x}) = \cup \text{x}$.
- Assume $otp = \cup \text{dot}$. Then $F(\cup \text{for } \text{dot in } \cup \text{x return } \cup e_1) = f$ if $X(f) = \text{dot}$.

In the first case $F(\cup \text{for } \text{dot in } \cup \text{x return } \cup e_1) = \cup \text{x}\{t_1, \ldots, t_n\}$. By induction this is equivalent with $M(\cup(\text{dot}\{t_1, \ldots, \text{dot}(t_n)\}) = \cup e_1$ and since $X^\cup(\text{dot}\{t_1, \ldots, \text{dot}(t_n)\}) = \cup e_1$, this is equivalent with $X^\cup(\cup \text{x}\{t_1, \ldots, t_n\}) = \cup e_1$ for $\text{dot}$ in $\cup \text{x return } \cup e_1$.

In the second case $F(\cup \text{for } \text{dot in } \cup \text{x return } \cup e_1) = \cup \text{x}^{\text{out}}\{t_1, \ldots, t_n\}$. By induction this is equivalent with $X^\cup(\cup \text{x}\{\text{dot}\{t_1, \ldots, \text{dot}(t_n), \text{dot}^{\text{out}}\}) = \cup e_1$ and since $X^\cup(\text{dot}\{t_1, \ldots, \text{dot}(t_n), \text{dot}^{\text{out}}\}) = \cup e_1$, this is equivalent with $X^\cup(\cup \text{x}\{t_1, \ldots, t_n\}) = \cup e_1$ for $\text{dot}$ in $\cup \text{x return } \cup e_1$.

Next we consider the cases for $aotp$:

- Assume $aotp = \cup a: n$. Then $F(\cup a: n) = \cup a: n^{\text{out}}$ and $M(\cup a: n^{\text{out}}) = \cup a: n$.  

18
Assume $atp = \forall$ for $\$dot$ in $\forall a::n$ return $orc_1$. Then $F(orc_1)$ is either of the form

$\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\}$ or $\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}, \$dot^{out}\}$.

In the first case $F((\forall \forall a::n \ return \ orc_1)) = \forall a::n\{t_1, \ldots, t_n\}$. By induction this is equivalent with $M^c(\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\}) = orc_1$ and since $X^c(\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\})$ this is equivalent with $X^c(\{x\{t_1, \ldots, t_n\}\})$ for $\$dot$ in $\forall a::n$ return $orc_1$.

In the second case $F((\forall \forall a::n \ return \ orc_1)) = \forall a::n^{out}\{t_1, \ldots, t_n\}$. By induction this is equivalent with $X^c(\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}, \$dot^{out}\}) = orc_1$ and since $X^c(\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\}) = X^c(\{t_1, \ldots, t_n\})$ this is equivalent with $X^c(\{a::n{t_1, \ldots, t_n}\}) = \forall a::n$ return $orc_1$.

Next we consider the cases for $t_p$:

- Assume $t_p = atp$. Then by induction $F(atp) = f$ iff $X^c(f) = atp$.
- Assume $t_p = \forall x$ such that $\$dot \neq \$dot$. Then $F(\forall x) = \forall x$ and $X^c(\forall x) = \forall x$.

- Assume $t_p = \forall$ for $\$dot$ in $\forall x$ return $rc_1$. Then $F(rc_1)$ is of the form $\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\}$. Then $F((\forall \forall x \ return \ rc_1)) = \forall x\{t_1, \ldots, t_n\}$. By induction this is equivalent with $M^c(\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\}) = rc_1$ and since $X^c(\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\}) = X^c(\{t_1, \ldots, t_n\})$ this is equivalent with $X^c(\forall x\{t_1, \ldots, t_n\}) = \forall \forall x \ return \ rc_1$.

Finally we consider the cases for $a::n$:

- Assume $a::n = \forall a::n$. Then $F((\forall a::n) = \forall a::n$ and $X^c(\forall a::n) = X^c(a::n) = \forall a::n$.
- Assume $a::n = \forall$ for $\$dot$ in $\forall a::n$ return $rc_1$. Then $F(rc_1)$ is of the form $\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\}$. Then $F((\forall \forall a::n \ return \ rc_1)) = \forall a::n\{t_1, \ldots, t_n\}$. By induction this is equivalent with $M^c(\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\}) = rc_1$ and since $X^c(\{\$dot\{t_1\}, \ldots, \$dot\{t_n\}\}) = X^c(\{t_1, \ldots, t_n\})$ this is equivalent with $X^c(\forall a::n\{t_1, \ldots, t_n\}) = \forall \forall a::n \ return \ rc_1$.

Since it was already established that the range of $X^c$ was a subset of TPNF it now follows that TPNF is exactly this range. Moreover, with $F$ 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 $\$dot$ 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 $\forall$ annotation is 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

19
Annotation Introduction and Propagation, and ddo Removal

<table>
<thead>
<tr>
<th>Source</th>
<th>Result</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddo(e)</td>
<td>ddo(\ddo e)</td>
<td></td>
</tr>
<tr>
<td>\alpha \text{if } \beta e_1 \text{ then } \gamma e_2</td>
<td>\alpha \text{if } \bigvee e_1 \text{ then } \gamma e_2</td>
<td>\beta \prec \bigvee</td>
</tr>
<tr>
<td>\alpha \text{for } $x$ in $e_1$ return $\gamma e_2$</td>
<td>\alpha \text{for } $x$ in $\bigvee e_1$ return $\gamma e_2$</td>
<td>$\cdot \prec \alpha$</td>
</tr>
<tr>
<td>\alpha \text{let } $x := \beta e_1$ return $\gamma e_2$</td>
<td>\alpha \text{let } $x := \beta e_1$ return $\alpha e_2$</td>
<td>$\gamma \prec \alpha$</td>
</tr>
<tr>
<td>\alpha \text{if } \beta e_1 \text{ then } \gamma e_2</td>
<td>\alpha \text{if } \beta e_1 \text{ then } \alpha e_2</td>
<td>\gamma \prec \alpha</td>
</tr>
<tr>
<td>\alpha \text{ddo}(\beta e)</td>
<td>\alpha e</td>
<td>$\cdot \prec \alpha \land \beta \preceq \alpha$</td>
</tr>
</tbody>
</table>

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

Rule may lead to duplication of expressions, and therefore a less efficient query plan, it is only applied to $\bigvee$ 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 $y/p_1$ return
  let $z := x/p_2$ return
    if $z/p_3$ then $z/p_4$
```

where $y/p_1$, $x/p_2$, $z/p_3$ and $z/p_4$ 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 $y/p_1[p_2/p_3]/p_2/p_4$.

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 $\bigvee e$ where all subexpressions in $e$ are annotated with $\cdot$ then the result is in the following syntax:

- $se ::= \bigvee x | \bigvee ax::nt | \bigvee \text{if } be \text{ then } se | \bigvee \text{for } x \text{ in } se \text{ return } se$
- $be ::= \bigvee x | \bigvee ax::nt | \bigvee \text{if } be \text{ then } be | \bigvee \text{for } x \text{ in } se \text{ return } be$
- $nt ::= \text{label} | \ast$
- $ax ::= \text{child} | \text{desc} | \text{d-o-s}$

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

### 4.3 Soundness and Completeness of the Rewrite 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 $\bigvee e$ to an expression in TPNF.

**Lemma 4.2.** If no rewrite rule applies to a CXQ$^+$ expression $\bigvee e$ then it is in TPNF.

**Proof.** If no rewrite rule applies then we may assume that the expression is in the normal form of Lemma 4.1. We proceed to show that for every such expression where all the subexpressions are in TPNF it holds that the full expression is either already in TPNF or there is at least one rewrite rule that applies.

Let $e$ be of the form
<table>
<thead>
<tr>
<th>Name</th>
<th>Source</th>
<th>Result</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Substitution</td>
<td>( \text{let } \alpha := \beta \alpha \text{ return } \alpha \beta )</td>
<td>( \alpha \beta \text{ return } \alpha \beta )</td>
<td>( \beta \alpha, \beta \beta \in CXQ, U \leq \alpha )</td>
</tr>
<tr>
<td>Loop Fusion</td>
<td>( \text{for } \dot \text{ in } U )</td>
<td>( \text{for } \dot \text{ in } U )</td>
<td>( \dot \text{ return } \alpha \beta )</td>
</tr>
<tr>
<td>Condition Detection</td>
<td>( \text{for } \alpha \text{ in } U \alpha )</td>
<td>( \text{if } \alpha )</td>
<td>( \beta \alpha \text{ then } \alpha \beta )</td>
</tr>
<tr>
<td>Condition Shift</td>
<td>( \text{if } )</td>
<td>( \text{if } )</td>
<td>( \alpha \beta )</td>
</tr>
<tr>
<td>Return Condition Lift</td>
<td>( \text{for } \alpha \text{ in } U \alpha )</td>
<td>( \text{if } \alpha )</td>
<td>( \beta \alpha \text{ then } \alpha \beta )</td>
</tr>
<tr>
<td>Nested Return Cond. Lift</td>
<td>( \text{for } \alpha \text{ in } U \alpha )</td>
<td>( \text{if } \alpha )</td>
<td>( \beta \alpha \text{ then } \alpha \beta )</td>
</tr>
<tr>
<td>Return Result Lift</td>
<td>( \text{for } \alpha \text{ in } U \alpha )</td>
<td>( \text{if } \alpha )</td>
<td>( \beta \alpha \text{ then } \alpha \beta )</td>
</tr>
<tr>
<td>Nested Return Result Lift</td>
<td>( \text{for } \alpha \text{ in } U \alpha )</td>
<td>( \text{if } \alpha )</td>
<td>( \beta \alpha \text{ then } \alpha \beta )</td>
</tr>
<tr>
<td>For Condition Lift</td>
<td>( \text{for } \alpha \text{ in } U \alpha )</td>
<td>( \text{if } \alpha )</td>
<td>( \beta \alpha \text{ then } \alpha \beta )</td>
</tr>
<tr>
<td>Trivial Dot Condition</td>
<td>( \text{if } \dot \text{ then } \alpha \beta )</td>
<td>( \alpha \beta )</td>
<td>None</td>
</tr>
<tr>
<td>Trivial Loop</td>
<td>( \text{for } \alpha \text{ in } U \alpha )</td>
<td>( \alpha \beta )</td>
<td>None</td>
</tr>
<tr>
<td>Introduction of Dot</td>
<td>( \text{for } \alpha \text{ in } U \alpha )</td>
<td>( \text{for } \alpha \beta )</td>
<td>None</td>
</tr>
<tr>
<td>Dot Loop</td>
<td>( \text{for } \alpha \beta \text{ then } \alpha \beta )</td>
<td>( \alpha \beta )</td>
<td>None</td>
</tr>
<tr>
<td>Shortening Condition</td>
<td>( \text{if } \alpha \beta \text{ then } \alpha \beta )</td>
<td>( \alpha \beta )</td>
<td>None</td>
</tr>
</tbody>
</table>

Figure 5: An overview of the structural rewrites.
\[\forall x (\text{OK})\]

\[\forall ax::nt (\text{OK})\]

\[\forall \text{if } be \text{ then } se\]
- if \(be\) is a \(tp\) in TPNF (OK)
- else if \(be\) is a \($dot\) (Trivial Dot Condition)
- else if \(be\) is of the form \(\forall \text{if } tp_1 \land \ldots \land tp_n \text{ then } tp_{n+1}\) (Condition Shift)

\[\forall \text{for } x \text{ in } se_1 \text{ return } se_2\]
- let $x \neq $dot
  * if $dot \notin FV(se_2)$ (Dot Introduction)
  * if $x \notin FV(se_2)$ (Condition Detection)
  * else, since \(\{dot, x\} \subset FV(se_2)\) we know that \(se_2\) is of the form: if \(tp_1 \land \ldots \land tp_n\) then \(otp\), and \$x\$ is either in \(FV(tp_i), i \leq n\) or \(FV(otp)\)
    - if \$x\$ \(\notin FV(tp_i), i \leq n\) ((Nested) Return Condition Lift)
    - else if \$x\$ \(\notin FV(otp)\) ((Nested) Return Result Lift)
- let $x = $dot
  * \(se_2\) is of form \(otp\) and \$y\$ is in \(FV(otp)\) ((Nested) Return Result Lift)
  * \(se_2\) is of the form if \(tp_1 \land \ldots \land tp_n\) then \(otp\) and \$y\$ is in \(FV(otp)\) ((Nested) Return Condition Lift)
  * \(FV(se_2) = \{dot\}\), thus \(se_2\) is an \(orc\)
    - \(se_1\) is of form if \(tp_1 \land \ldots \land tp_n\) then \(otp\) (For Condition Lift)
    - \(se_1\) is of the form \(otp\)
      * if \(otp = y\) (OK)
      * else if \(otp = dot\) (Dot Loop)
      * else if \(otp = ax::nt\) (OK)
      * else if \(otp = for dot\) in \$x\$ return \(orc\) (Loop Fusion)
      * else if \(otp = for dot\) in ax::nt return \(orc\) (Loop Fusion)

\[\forall x\]
- if $x \neq $dot (OK)
- else if $x = $dot (OK)

\[\forall ax::nt (\text{OK})\]

\[\forall \text{if } be_1 \text{ then } be_2\]
- if \(be_1\) is a \(dot\) (Trivial Dot Condition)
- else if \(be_1\) is a \(tp\)
  * if \(be_2\) is a \(dot\) (Condition Shortening)
  * else if \(be_2\) is a \(tp\) (OK)
  * else if \(be_2\) is of the form if \(tp_1 \land \ldots \land tp_n\) then \(tp_{n+1}\) (OK)
- else \(be_1\) is of the form if \(tp_1 \land \ldots \land tp_n\) then \(tp_{n+1}\) (Condition Shift)

\[\forall \text{for } x \text{ in } se \text{ return } be\]
Lemma 4.3. If a CXQ\(^+\) expression is in TPNF, then none of the rewrite rules apply.

Proof. We consider every rewrite rule separately:

- **Loop Fusion**: In TPNF, in-clauses of for-expressions cannot contain a for-expression;
- **Condition Detection**: In TPNF, $\dot{x}$ is always the loop variable, and all rc/orc expressions have $\dot{x}$ as the free variable;
- **Condition Shift**: In TPNF, if-clauses are not allowed to occur recursively;
- **(Nested) Return Condition Lift**: In TPNF, the if-clause in a return-clause must be an \(atp\) and the free variable of an \(atp\)-expression always is $\dot{x}$;
- **(Nested) Return Result Lift**: In TPNF, the then-clause in a return-clause must be an rc, orc or $\dot{x}$. The free variable in an rc, orc, $\dot{x}$ always is $\dot{x}$;
- **For Condition Lift**: In TPNF, the in-clause may not contain an if-expression;
- **Trivial Dot Condition**: In TPNF, the if-clause contains \(tp\) or \(atp\) expressions and \(tp/atp\) expressions cannot be $\dot{x}$;
- **Trivial Loop**: In TPNF, no variable reference is allowed to occur directly in the return-clause;
- **Introduction of Dot**: In TPNF, every loop variable is $\dot{x}$;
- **Shortening Condition**: In TPNF, $\dot{x}$ in a then-clause is only allowed in orc-expressions, but orc never has an existentional annotation.
Lemma 4.4. The rewriting process of a $CXQ^+$ expression always stops after a finite number of rewrites.

Proof. We show this lemma by associating with each $CXQ^+$ expression a structural cost and an annotation cost. All structural manipulations strictly reduce the structural cost, while the annotation propagation does not change the structural cost (because the structure is not changed), but reduces the annotation cost.

A structural cost function $c$ is defined by a 15-tuple $(w_1, \ldots, w_{15})$ of natural numbers and maps $CXQ$ expressions to a natural number in the following way:

$$
c(\text{for } x \text{ in } e_1 \text{ return } e_2) = w_1 * c(e_1) + w_2 * c(e_2) + w_3
$$

$$
c(\text{for } x::nt \text{ in } e_1 \text{ return } e_2) = w_4 * c(e_1) + w_5 * c(e_2) + w_6
$$

$$
c(\text{if } e_1 \text{ then } e_2) = w_7 * c(e_1) + w_8 * c(e_2) + w_9
$$

$$
c(ax::nt) = w_{10}
$$

$$
c(\$x) = w_{11}
$$

$$
c(\$dot) = w_{12}
$$

$$
c(\text{let } x := e_1 \text{ return } e_2) = w_{13} * c(e_1) + w_{14} * c(e_2) + w_{15}
$$

where we assume that $x \neq \$dot$. Based on this notion of cost function, we define a combined cost function $C$ by an n-tuple $(c_1, c_2, c_3, c_4, c_5)$ for which it holds that, when looking at the lexicographical order, the cost for all expressions $e$ diminishes when applying a rewrite rule, i.e., if $e_1$ if 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 $c_j(e_2) = c_j(e_1)$.

The first cost function is defined by $(1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1)$ and counts the number of let expressions. It is clear that the value of $c_1(e)$ strictly diminishes when applying the substitution rule and does not change when applying any other structural manipulation rule.

The second cost function $c_2$ is defined by $(1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0)$ and intuitively indicates the size of the expression. The value of $c_2(e)$ strictly diminishes when applying:

- condition detection: $c_2(e_1) + c_2(e_2) + 1 \sim c_2(e_1) + c_2(e_2)
- trivial dot condition: $c_2(e_2) + 1 \sim c_2(e_2)
- trivial loop: $c_2(e) + 2 \sim c_2(e)
- introduction of dot: $c_2(e_1) + c_2(e_2) + 1 \sim c_2(e_1) + c_2(e_2)
- dot loop: $c_2(e_1) + 1 \sim c_2(e_1)
- shortening condition: $c_2(e_1) + 1 \sim c_2(e_1)

The value of $c_2(e)$ remains the same when applying:

- loop fusion: $c_2(e_1) + c_2(e_2) + c_2(e_3) + 2 \sim c_2(e_1) + c_2(e_2) + c_2(e_3) + 2
- condition shift: $c_2(e_1) + c_2(e_2) + c_2(e_3) \sim c_2(e_1) + c_2(e_2) + c_2(e_3)$
- (nested) return condition lift: $c_2(e_1) + c_2(e_2) + \ldots + c_2(e_n) + c_2(e_{n+1}) + 1 \sim c_2(e_1) + c_2(e_2) + \ldots + c_2(e_n) + c_2(e_{n+1}) + 1
- (nested) return result lift: $c_2(e_1) + c_2(e_2) + \ldots + c_2(e_n) + c_2(e_{n+1}) + 1 \sim c_2(e_1) + c_2(e_2) + \ldots + c_2(e_n) + c_2(e_{n+1}) + 1
- for condition lift: $c_2(e_1) + c_2(e_2) + c_2(e_3) + 1 \sim c_2(e_1) + c_2(e_2) + c_2(e_3) + 1

The third cost function $c_3$ is defined by $(2, 1, 1, 2, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0)$ and intuitively states that we want to make the in-clauses of for-loops as simple as possible. The value of $c_3(e)$ strictly diminishes when applying:

- loop fusion: $4 * c_3(e_1) + 2 * c_3(e_2) + c_3(e_3) + 3 \sim 2 * c_3(e_1) + 2 * c_3(e_2) + c_3(e_3) + 2$
• for condition lift: \( c_3(e_1) + c_3(e_2) + c_3(e_3) + 1 \sim c_3(e_1) + c_3(e_2) + c_3(e_3) + 1 \)

The value of \( c_3(e) \) remains the same when applying:

• condition shift: \( c_3(e_1) + c_3(e_2) + c_3(e_3) \sim c_3(e_1) + c_3(e_2) + c_3(e_3) \)

• (nested) return condition lift: \( 2 \ast c_3(e_1) + c_3(e_2) + \ldots + c_3(e_n) + c_3(e_{n+1}) + 1 \sim 2 \ast c_3(e_1) + c_3(e_2) + \ldots + c_3(e_n) + c_3(e_{n+1}) + 1 \)

• (nested) return result lift: \( 2 \ast c_3(e_1) + c_3(e_2) + \ldots + c_3(e_n) + c_3(e_{n+1}) + 1 \sim 2 \ast c_3(e_1) + c_3(e_2) + \ldots + c_3(e_n) + c_3(e_{n+1}) + 1 \)

The fourth cost function \( c_4 \) is defined by \( \langle 2, 2, 1, 2, 1, 1, 0, 1, 1, 1, 1, 0 \rangle \) and intuitively states that we want to get as much subexpressions as possible outside of for-loops. The value of \( c_4(e) \) strictly diminishes when applying:

• (nested) return condition lift: \( 2 \ast c_4(e_1) + 2 \ast c_4(e_2) + \ldots + 2 \ast c_4(e_n) + 2 \ast c_4(e_{n+1}) + 1 \sim 2 \ast c_4(e_1) + 2 \ast c_4(e_2) + \ldots + 2 \ast c_4(e_n) + 2 \ast c_4(e_{n+1}) + 1 \)

• (nested) return result lift: \( 2 \ast c_4(e_1) + 2 \ast c_4(e_2) + \ldots + 2 \ast c_4(e_n) + 2 \ast c_4(e_{n+1}) + 1 \sim 2 \ast c_4(e_1) + 2 \ast c_4(e_2) + \ldots + 2 \ast c_4(e_n) + c_4(e_{n+1}) + 1 \)

Finally, the fifth cost function \( c_5 \) is defined by \( \langle 1, 1, 1, 1, 1, 1, 2, 1, 0, 1, 1, 1, 1, 0 \rangle \), which intuitively states that we want to push as much as possible out of the if-clause. The value of \( c_5(e) \) strictly diminishes when applying:

• condition shift: \( c_4(e_1) + c_4(e_2) + c_4(e_3) \sim c_4(e_1) + c_4(e_2) + c_4(e_3) \)

This concludes the proof for the structural manipulation. We now show that the annotation propagation also terminates. Associate with each annotation symbol a cost, such that stronger annotations have a smaller cost. If the total annotation cost is the sum of all individual annotation costs, then it is clear that this cost strictly diminishes after each annotation propagation.

Combining these lemmas we then obtain the desired theorem.

**Theorem 4.3.** When applied in an arbitrary order the rewrite rules rewrite every CXQ expression of form ‘e’ to an equivalent expression in TPNF within a finite number of steps.

**Proof.** This follows from Lemma 4.2, Lemma 4.3 and Lemma 4.4, and the observation that each rewrite rule preserves the semantics of the rewritten expression.

Observe that as a corollary we obtain Theorem 3.3 because if a CXQ expression \( e \) has the properties ord and nodup then it is equivalent with ‘ddo(e)’ where ‘e’ is equal to \( e \) except that all subexpressions are annotated with ‘. The rewrite process will rewrite this to the equivalent ‘ddo(‘e’)’ where ‘e’ is in TPNF. By Theorem 4.1 we know that this is equivalent with the semantics of a forest pattern. By the same argument we also get the proof for Lemma 3.1 since the expression ‘e’ will be either of the form otp or of the form if \( tp_1 \land \ldots \land tp_n \) then otp. In the first case it holds that \( \mathsf{FV}(‘e’) \) is a single variable which must be \( rv(e) \) since the rewrite rules do not change the set of free variables, and in the second case ‘e’ is either equivalent with if (if \( tp_1 \land \ldots \land tp_{n-1} \) then \( tp_n \)) then otp if \( n > 1 \), or if \( tp_1 \) then otp if \( n = 1 \). Since the rewrite rules do not change the root variable of an expression, it follows that \( rv(e) = rv(‘e’) = rv(otp) \) which is the only element in \( \mathsf{FV}(otp) \) because it has at most one free variable.
4.4 Proof of Confluence

The presented set of rewriting rules is strictly speaking not confluent 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 tree pattern if after rewriting the result is transformed into a tree pattern with the function $F$.

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 $\emptyset$ node with no children it is removed.
- $f_1 + f_2$ defines the disjoint union of $f_1$ and $f_2$. Only defined if $f_1$ and $f_2$ are not both output patterns.
- $f_1 \triangleleft f_2$ 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.
- $f_1 \triangleleft^* f_2$ is defined such that $f_1 \triangleright^* \{t_1, \ldots, t_n\} = f_1 \triangleright \{t_1\} + \ldots + f_1 \triangleright \{t_n\}$ and $f_1 \triangleright^* \emptyset = \emptyset$.
- $f^{s_x}$ selects from $f$ the trees roots labeled $s_x$.
- $f^{-s_x}$ selects from $f$ the trees with a root not labeled $s_x$.

**Lemma 4.5.** For the operations the following algebraic identities hold ($vp$ denotes something of the form $s_x$ or $-s_x$):

\[
\begin{align*}
    f_1 + f_2 & = f_2 + f_1 & f_1 + \emptyset & = f_1 \\
    f_1 + (f_2 + f_3) & = (f_1 + f_2) + f_3 & c(f_1 + f_2) & = c(f_2) + c(f_1) \\
    (f_1 \triangleleft f_2)^{vp} & = f_1^{vp} \triangleleft f_2 & (f_1 + f_2) \triangleright^* f_3 & = (f_1 \triangleright^* f_3) + (f_2 \triangleright^* f_3) \\
    (f_1 \triangleright f_2)^{vp} & = f_1^{vp} \triangleright f_2 & f_1 \triangleright^* (f_2 + f_3) & = (f_1 \triangleright^* f_2) + (f_1 \triangleright^* f_3) \\
    (f_1 + f_2)^{vp} & = f_1^{vp} + f_2^{vp} & f_1 \triangleright (f_2 + f_3) & = (f_1 \triangleright f_2) + (f_1 \triangleright f_3) \\
    c(f)^{vp} & = c(f^{vp}) & f_1 \triangleright^* (f_2 \triangleright^* f_3) & = (f_1 \triangleright^* f_2) \triangleright^* f_3 \\
    (f^{s_x})^{-s_x} & = \emptyset & f_1 \triangleright (f_2 \triangleright^* f_3) & = (f_1 \triangleright^* f_2) \triangleright^* f_3 \\
    (f^{s_x})^{s_y} & = \emptyset \quad (\text{if } s_x \neq s_y) & f_1 \triangleright^* (f_2 \triangleright f_3) & = (f_1 \triangleright f_2) \triangleright^* f_3 \\
    (f^{vp})^{vp} & = (f^{vp})^{vp} & f \triangleright \emptyset & = c(f) \\
    (f^{vp})^{-vp} & = f^{vp} & c(\emptyset) & = \emptyset \\
    f \triangleright^* \emptyset & = \emptyset & c(f_1 \triangleright f_2) & = f_1 \triangleright c(f_2) \\
    c(f_1 \triangleright f_2) & = f_1 \triangleright^* c(f_2) & c(f_1) \triangleright f_2 & = c(f_1) \\
    f & = (f^{s_x} + f^{-s_x}) & (f^{-s_x})^{s_y} & = f^{s_y} \quad (\text{if } s_x \neq s_y)
\end{align*}
\]

**Proof.** The identities follow straightforwardly from the definitions of the operations. \hfill $\square$

We then define the mapping $M$ from $CXQ^+$ expressions to forest patterns as follows. For expressions annotated with $\alpha \leq \cup$ we define $M$ such that:

- $M(\alpha s_x) = s_x^{out}$
- $M(\alpha a::n) = sdot(a::n^{out})$
- $M(\alpha ddo(\beta e)) = M(\beta e)$
- $M(\alpha \text{if } \beta e_1 \text{ then } \gamma e_2) = c(M(\beta e_1)) + M(\gamma e_2)$
- $M(\alpha \text{for } s_x \in \beta e_1 \text{ return } \gamma e_2) = M(\gamma e_2)^{-s_x} + (M(\beta e_1) \triangleleft M(\gamma e_2)^{s_x})$
- $M(\alpha \text{let } s_x := \beta e_1 \text{ return } \gamma e_2) = M(\gamma e_2)^{-s_x} + (M(\beta e_1) \triangleright^* M(\gamma e_2)^{s_x})$
For all rewrite rules it holds that if they rewrite expressions annotated with $\text{prop}$ then $M(e) = \text{prop}(e)$. We show this separately for each of the rules using Lemma 4.5 and Lemma 4.6.

**Lemma 4.6.** If $\$x \notin FV(e)$ then $M(e)^{\$x} = \emptyset$ and $M(e)^{-\$x} = M(e)$.

**Proof.** It can be proven with induction on the structure of $e$ that $M(e)$ will not contain any roots labeled with $\$x$.

**Lemma 4.7.** For all rewrite rules it holds that if they rewrite $e_1$ to $e_2$ then $M(e_1) = M(e_2)$.

**Proof.** We show this separately for each of the rules using Lemma 4.5 and Lemma 4.6.

**Annotation Introduction** Consider the two rules:

- By definition $M(d\text{do}(c)) = M(c) = M(\text{prop}(c)) = M(d\text{do}(\text{prop}(c)))$.
- Assume that $\alpha \preceq \cup$ then $M(\text{if } \beta_1 \text{ then } \gamma_2) = c(M(\beta_1)) + M(\gamma_2) = M(\gamma_1) + M(\gamma_2) = c(M(\gamma_1)) + M(\gamma_2) = M(\text{if } \gamma_1 \text{ then } \gamma_2) + M(\gamma_2)$.

**Annotation Propagation** Consider the four rules:

- Since $M(e_1) = M(\gamma_1)$ then $M(\text{for } \$x \text{ in } e_1 \text{ return } \gamma_2) = M(\text{for } \$x \text{ in } \beta_1 \text{ return } \gamma_2)$.
- First consider the case where $\alpha = \forall$ and therefore also $\gamma = \forall$. Then it follows that $M(\text{for } \$x \text{ in } \beta_1 \text{ return } \gamma_2) = M(\gamma_2)^{-\$x} + M(\beta_1) < M(\gamma_2)^{\$x} = M(\gamma_2)^{-\$x} + M(\beta_1) < M(\gamma_2)^{\$x}$. Next consider the case where $\alpha = \forall$. Then $M(\text{for } \$x \text{ in } \beta_1 \text{ return } \gamma_2) = c(M(\gamma_2)^{-\$x} + M(\beta_1) < M(\gamma_2)^{\$x}) = c(M(\gamma_2)^{-\$x} + c(M(\beta_1) < M(\gamma_2)^{\$x})) = c(M(\gamma_2)^{-\$x} + c(M(\beta_1) < M(\gamma_2)^{\$x})) = M(\gamma_2)^{-\$x} + M(\beta_1) < M(\gamma_2)^{\$x} = M(\text{for } \$x \text{ in } \beta_1 \text{ return } \gamma_2)$.
- By the same argument as in the previous point except replacing $\preceq$ with $\preceq^*$ it holds that $M(\text{let } \$x := \beta_1 \text{ return } \gamma_2) = M(\text{let } \$x := \beta_1 \text{ return } \gamma_2)$.
- First consider the case where $\alpha = \forall$ and therefore also $\gamma = \forall$, then $M(\gamma_2) = M(\gamma_2)$ and therefore $M(\text{if } \beta_1 \text{ then } \gamma_2) = M(\text{if } \beta_1 \text{ then } \gamma_2)$. Then consider the case where $\alpha = \forall$. Then $M(\gamma_2) = c(M(\gamma_2) + M(\gamma_2)) = c(M(\gamma_2) + M(\gamma_2)) = M(\gamma_2)^{-\$x} + M(\beta_1) < M(\gamma_2)^{\$x}$.

**Removal of $d\text{do}$** Assume that $\alpha = \forall$ and therefore $\beta = \forall$. Then $M(\text{for } \$x \text{ do } \beta_1)$ = $M(\beta_1)$ = $M(\alpha)$. Assume that $\alpha = \forall$. Then $M(\text{for } \$x \text{ do } \beta_1) = M(\beta_1) = M(\alpha)$.

**Substitution** We have to show that $M(\text{let } \$x := \beta_1 \text{ return } \gamma_2) = c(M(\text{for } \$x \text{ do } \beta_1) + M(\gamma_2))$. Since $M(\text{let } \$x := \beta_1 \text{ return } \gamma_2) = c(M(\text{for } \$x \text{ do } \beta_1) + M(\gamma_2))$, we get $M(\text{for } \$x \text{ do } \beta_1) + M(\gamma_2)$.

**Assume** $e_2 = y$. If $\$x = \$y$ then $M(e_2[\$x/\$y]) = M(e_1)$ and $M(\text{let } \$x := e_1 \text{ return } e_2) = (\text{let } \$x := e_1 \text{ return } e_2) = (\$y^{\text{out}})^{-\$x} + M(\gamma_1) < (\$y^{\text{out}})^{-\$x} = \emptyset + M(\gamma_1) < \emptyset = M(e_1)$. If $\$x \neq \$y$ then $M(e_2[\$x/\$y]) = M(\$y) = \$y^{\text{out}}$ and $M(\text{let } \$x := e_1 \text{ return } e_2) = M(\$y)^{-\$x} + M(\gamma_1) < M(\$y)^{-\$x} + M(\gamma_1) < \emptyset = \emptyset$. We do this by induction on the structure of $e_2$. For brevity we omit the annotations of the expressions. Consider the cases:

- Assume $e_2 = y$. If $\$x = \$y$ then $M(e_2[\$x/\$y]) = M(e_1)$ and $M(\text{let } \$x := e_1 \text{ return } e_2) = (\$y^{\text{out}})^{-\$x} + M(\gamma_1) < (\$y^{\text{out}})^{-\$x} = \emptyset + M(\gamma_1) < \emptyset = M(e_1)$. If $\$x \neq \$y$ then $M(e_2[\$x/\$y]) = M(\$y) = \$y^{\text{out}}$ and $M(\text{let } \$x := e_1 \text{ return } e_2) = M(\$y)^{-\$x} + M(\gamma_1) < M(\$y)^{-\$x} + M(\gamma_1) < \emptyset = \emptyset$. 27
• Assume $e_2 = a\cdot n$. Then $M(e_2[\$x/e_1]) = M(a\cdot n) = \$dot{a}\cdot n$ and $M(\ell \$x := e_1 return e_2) = (\$dot{a}\cdot n) - \$x + (M(e_1) \triangleleft (\$dot{a}\cdot n) - \$x)$. Since by definition of $\text{CXQ}$ it holds that $\$x \neq \$dot$ this equals $(\$dot{a}\cdot n) + (M(e_1) \triangleleft \emptyset) = \$dot{a}\cdot n + \emptyset = \$dot{a}\cdot n$.

• Assume $e_2 = d\circ d(e_3)$. Then $M(e_2[\$x/e_1]) = M(d\circ d(e_3[\$x/e_1])) = M(e_3[\$x/e_1])$ and $M(\ell \$x := e_1 return e_2) = M(\ell \$x := e_1 return e_3)$, and by induction $M(e_3[\$x/e_1]) = M(\ell \$x := e_1 return e_3)$.

• Assume $e_2 = (c_3 \text{ then } e_1)$. Then $M(e_2[\$x/e_1]) = M(if c_3[\$x/e_1] \text{ then } e_4[\$x/e_1]) = c(M(e_3[\$x/e_1]) + M(e_4[\$x/e_1]))$. By induction this is equal to $c(M(\ell \$x := e_1 return e_3)) + M(\ell \$x := e_1 return e_4)$.

• Assume $e_2 = if c_3 \text{ then } e_1$. Then $M(e_2[\$x/e_1]) = M(if c_3[\$x/e_1] \text{ then } e_4[\$x/e_1]) = c(M(e_3[\$x/e_1]) + M(e_4[\$x/e_1]))$. By induction this is equal to $c(M(\ell \$x := e_1 return e_3)) + M(\ell \$x := e_1 return e_4)$.

• Assume $e_2 = for \$y \text{ in } e_3 \text{ return } e_4$. If $\$x = \$y$ then it holds that $M(e_2[\$x/e_1]) = M(for \$y \text{ in } e_3[\$x/e_1] \text{ return } e_4) = M(e_4 - \$y + (M(e_3[\$x/e_1]) \triangleleft M(e_4)) \triangleleft \$y)$. By induction this is equal to $M(e_4 - \$y + (M(\ell \$x := e_1 return e_3) \triangleleft M(e_4)) \triangleleft \$y)$.

• Assume $e_2 = let \$y := e_3 \text{ return } e_4$. The proof is as for $e_2 = for \$y \text{ in } e_3 \text{ return } e_4$.

**Loop Fusion** We need to show that $M(a\cdot for \$dot \text{ in } (c\cdot for \$dot \text{ in } v_1 e_1 \text{ return } v_2 e_2) \text{ return } v_3 e_3)$ is equal to $M(a\cdot for \$dot \text{ in } v_1 e_1 \text{ return } v_2 e_2) \text{ return } v_3 e_3)$. Observe that $M(a\cdot for \$dot \text{ in } (c\cdot for \$dot \text{ in } v_1 e_1 \text{ return } v_2 e_2) \text{ return } v_3 e_3) = c(M(a\cdot for \$dot \text{ in } v_1 e_1 \text{ return } v_2 e_2) \text{ return } v_3 e_3)$ and $M(v_3 e_3)$.
c(M(\text{if } e_0 \text{ then } e_1 \text{ return } (\text{if } e_2 \text{ then } e_3)))$. Therefore it is sufficient to prove this for $\alpha = \emptyset$. For brevity we omit the $\cup$ annotations in the following.

$$
M(\text{for } e_0 \text{ in } e_1 \text{ return } e_2) = M(e_0) = \text{if } e_0 \text{ then } e_2
$$

Condition Detection

We need to show that under the assumption that $\not\exists x \in FV(e_2)$ it holds that $M(\text{if } e_0 \text{ in } e_1 \text{ return } e_2)$ equals $M(\text{if } e_0 \text{ then } e_2)$. Observe that $M(\text{if } e_0 \text{ in } e_1 \text{ return } e_2) = c(M(\text{if } e_0 \text{ in } e_1 \text{ return } c(M(\text{if } e_0 \text{ then } e_2))))$. Therefore it is sufficient to prove this for $\alpha = \emptyset$. For brevity we omit the $\cup$ annotations in the following.

$$
M(\text{for } e_0 \text{ in } e_1 \text{ return } e_2) = M(e_0) = \text{if } e_0 \text{ then } e_2
$$

Condition Shift

We need to show that $M(\text{if } e_0 \text{ then } e_2)$ equals $M(\text{if } e_0 \text{ in } e_1 \text{ return } e_2)$. Observe that $M(\text{if } e_0 \text{ then } e_2) = c(M(\text{if } e_0 \text{ then } e_2) \text{ then } e_3))$ and $M(\text{if } e_0 \text{ then } e_2) = c(M(\text{if } e_0 \text{ then } e_2))$. Therefore it is sufficient to prove this for $\alpha = \emptyset$. For brevity we omit the $\cup$ annotations in the following.

$$
M(\text{if } e_0 \text{ then } e_2) = c(M(\text{if } e_0 \text{ then } e_2) \text{ then } e_3))
$$

Return Condition Lift

We need to show that under the assumption that $\not\exists x \in FV(e_2)$ it holds that $M(\text{if } e_0 \text{ in } e_1 \text{ return } (\text{if } e_2 \text{ then } e_3))$ is equal to $M(\text{if } e_0 \text{ then } e_2)$. Observe that $M(\text{if } e_0 \text{ then } e_2) = c(M(\text{if } e_0 \text{ then } e_2) \text{ then } e_3))$. Therefore it is sufficient to prove this for $\alpha = \emptyset$. For brevity we omit the $\cup$ annotations in the following.

$$
M(\text{if } e_0 \text{ then } e_2) = c(M(\text{if } e_0 \text{ then } e_2) \text{ then } e_3))
$$

Nested Return Condition Lift

We need to show that under the assumption that $\not\exists x \in FV(e_n)$ it holds that $M(\text{if } e_0 \text{ in } e_1 \text{ return } (\text{if } e_2 \text{ \ldots } e_n \text{ then } e_{n+1}))$ is equal to $M(\text{if } e_0 \text{ then } e_2 \text{ \ldots } e_n \text{ then } e_{n+1})$. Observe that $M(\text{if } e_0 \text{ then } e_2 \text{ \ldots } e_n \text{ then } e_{n+1}) = c(M(\text{if } e_0 \text{ in } e_1 \text{ return } (\text{if } e_2 \text{ \ldots } e_n \text{ then } e_{n+1})))$. Therefore it is sufficient to prove this for $\alpha = \emptyset$. For brevity we omit the $\cup$ annotations in the following.
\[ M(\text{for } \$x \text{ in } e_1 \text{ return (if } e_2 \text{ then } e_3)) = M(\text{if } e_2 \land \ldots \land e_n \text{ then } e_{n+1})^{-\$z} + (M(e_1) \triangleleft M(\text{if } e_2 \land \ldots \land e_n \text{ then } e_{n+1})^{\$z}) \]
\[ = (c(M(e_2)) + c(M(e_3)))^{-\$z} + (M(e_1) \triangleleft (c(M(e_2)) + c(M(e_3)))^{\$z}) \]
\[ = c(M(e_2))^{-\$z} + \ldots + c(M(e_n))^{-\$z} + (M(e_1) \triangleleft (c(M(e_2))^{-\$z} + \ldots + c(M(e_n))^{-\$z})) + M(e_{n+1})^{\$z}) \]

Return Result Lift

\[ M(\text{for } \$x \text{ in } e_1 \text{ return (if } e_2 \text{ then } e_3)) \]
\[ = M(\text{if } e_2 \land \ldots \land e_{n-1} \text{ then } e_{n+1})^{-\$z} + (M(e_1) \triangleleft M(\text{if } e_2 \land \ldots \land e_{n-1} \text{ then } e_{n+1})^{\$z}) \]
\[ = c(M(e_1)) + M(\text{for } \$x \text{ in } e_1 \text{ return (if } e_1 \land \ldots \land e_{n-1} \text{ then } e_{n+1})) \]

Nested Return Result Lift

\[ M(\text{for } \$x \text{ in } e_1 \text{ return (if } e_2 \text{ then } e_3)) = M(\text{if } e_2 \land \ldots \land e_{n-1} \text{ then } e_{n+1})^{-\$z} + (M(e_1) \triangleleft M(\text{if } e_2 \land \ldots \land e_{n-1} \text{ then } e_{n+1})^{\$z}) \]
\[ = (c(M(e_2)) + c(M(e_3)))^{-\$z} + (M(e_1) \triangleleft (c(M(e_2)) + c(M(e_3)))^{\$z}) \]
\[ = c(M(e_2))^{-\$z} + \ldots + c(M(e_n))^{-\$z} + (M(e_1) \triangleleft (c(M(e_2))^{-\$z} + \ldots + c(M(e_n))^{-\$z})) + M(e_{n+1})^{\$z}) \]
Lemma 4.8. For every \( CXQ^+ \) expression \( \U e \) it holds that if \( F(\U e) \) is defined then \( F(\U e) = M(\U e) \).

Proof. We prove with induction upon the structure of \( \U e \) that for all \( CXQ^+ \) expressions \( \alpha e \) it holds that if \( F(\alpha e) \) is defined then \( F(\alpha e) = M(\alpha e) \).

- Assume \( \alpha e = \U x \). Then \( F(\U x) = x_{\text{out}} = M(\U x) \).
- Assume \( \alpha e = \U^e x \). Then \( F(\U^e x) = x = c(x_{\text{out}}) = c(M(\U x)) = M(\U^e x) \).
- Assume \( \alpha e = \U a:n \). Then \( F(\U a:n) = \text{Sdot}(a:n_{\text{out}}) = M(\U a:n) \).

For Condition Lift We have to show that \( M(\alpha x \text{for } x \text{ in } \U(\U e_1 \text{ then } \U e_2) \text{ return } \alpha e_3) \) is equal to \( M(\alpha x \text{if } \U e_1 \text{ then } \U(\U e_2 \text{ return } \alpha e_3)) \). Observe that it holds that \( M(\alpha x \text{if } \U e_1 \text{ then } \U e_2) \text{ return } \alpha e_3 = c(M(\U x \text{for } x \text{ in } \U(\U e_1 \text{ then } \U e_2) \text{ return } \alpha e_3)) \) and \( M(\alpha x \text{if } \U e_1 \text{ then } \U(\U e_2 \text{ return } \U e_3) = c(M(\U x \text{if } \U e_1 \text{ then } \U(\U e_2 \text{ return } \U e_3))) \). Therefore it is sufficient to prove this for \( \alpha = \emptyset \). For brevity we omit the \( \cup \) annotations in the following.

\[
M(\alpha x \text{if } \U e_1 \text{ then } \U e_2) \text{ return } \alpha e_3 = \alpha e \text{ for } \U e_1 \text{ if } \U e_2 \text{ then } \U e_3 \text{ return } \U e_3.
\]

For brevity we omit the \( \cup \) annotations in the following.

\[
M(\alpha x \text{if } \U e_1 \text{ then } \U e_2) \text{ return } \alpha e_3 = M(\alpha x \text{if } \U e_1 \text{ then } \U e_2) \text{ return } \alpha e_3.
\]

Trivial Dot Condition We have to show that \( M(\alpha x \text{if } \U \text{dot } \text{then } \alpha e_2) \) is equal to \( M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2) \). Observe that \( M(\alpha x \text{if } \U \text{dot } \text{then } \alpha e_2) = c(M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2)) \) and \( M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2) = c(M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2)) \). Therefore it is sufficient to prove this for \( \alpha = \emptyset \). For brevity we omit the \( \cup \) annotations in the following.

\[
M(\alpha x \text{if } \U \text{dot } \text{then } \alpha e_2) = M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2).
\]

Introduction of Dot We have to show that if \( \text{Sdot} \notin FV(e_2) \) then \( M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2) \) is equal to \( M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2 \cup \alpha (\U x / \text{Sdot})) \). Observe that \( M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2) = c(M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2)) \) and \( M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2 \cup \alpha (\U x / \text{Sdot})) = c(M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2 \cup \alpha (\U x / \text{Sdot}))) \). Therefore it is sufficient to prove this for \( \alpha = \emptyset \). For brevity we omit the \( \cup \) annotations in the following.

\[
M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2) = M(\alpha x \text{if } \U e_1 \text{ return } \alpha e_2 \cup \alpha (\U x / \text{Sdot}))
\]

Dot Loop We have to show that \( M(\alpha x \text{if } \U \text{dot } \text{then } \alpha e_1) \) is equal to \( M(\alpha x \text{if } \U e_1) \). Observe that \( M(\alpha x \text{if } \U \text{dot } \text{then } \alpha e_1) = c(M(\alpha x \text{if } \U e_1) \text{ return } \U e_1) \) and \( M(\alpha x \text{if } \U e_1) = c(M(\alpha x \text{if } \U e_1) \text{ return } \U e_1) \). Therefore it is sufficient to prove this for \( \alpha = \emptyset \). For brevity we omit the \( \cup \) annotations:

\[
M(\alpha x \text{if } \U \text{dot } \text{then } \alpha e_1) = M(\alpha x \text{if } \U e_1) \cup \alpha (\U x / \text{Sdot})
\]

Shortening Condition We have to show that \( M(\alpha x \text{if } \U e \text{ then } \U \text{dot } \text{then } \alpha e) = c(c(M(\alpha x \text{if } \U e) \text{ return } \alpha e)) \) for all \( \alpha e \) and \( \U e \). Observe that it holds that if \( F(\alpha e) \) is defined then \( F(\alpha e) = M(\alpha e) \).
• Assume $a: e$. Then $F(\langle a::n \rangle) = \text{\texttt{sdot}}\langle a::n\text{\texttt{out}} \rangle = c(\text{\texttt{sdot}}\langle a::n\text{\texttt{out}} \rangle) = c(M(\langle x:\bar{e} \rangle)) = M(\langle x:\bar{e} \rangle).

• Assume $a: e$. Then $F(\langle x: e \rangle) = F(\langle x: e \rangle)$ by induction equals $M(\langle x: e \rangle) + M(\langle e: e \rangle)$. Since the result of $F(\langle x: e \rangle)$ is a condition pattern this is equal to $c(M(\langle x: e \rangle)) + c(M(\langle e: e \rangle)) = M(\langle x: e \rangle)$.

Assume $a: e$. Then $F(\langle x: e \rangle) = F(\langle x: e \rangle)$ by induction equals $M(\langle x: e \rangle) + M(\langle e: e \rangle)$ when $\langle x: e \rangle$ is a condition pattern. Since the result of $F(\langle x: e \rangle)$ is a condition pattern this is equal to $c(M(\langle x: e \rangle)) + c(M(\langle e: e \rangle)) = M(\langle x: e \rangle)$.

• Assume $a: e$. Then $F(\langle x: e \rangle) = F(\langle x: e \rangle)$ by induction equals $M(\langle x: e \rangle) + M(\langle e: e \rangle)$ when $\langle x: e \rangle$ is a condition pattern. Since the result of $F(\langle x: e \rangle)$ is a condition pattern this is equal to $c(M(\langle x: e \rangle)) + c(M(\langle e: e \rangle)) = M(\langle x: e \rangle)$.

• Assume $a: e$. Then $F(\langle x: e \rangle) = F(\langle x: e \rangle)$ by induction equals $M(\langle x: e \rangle) + M(\langle e: e \rangle)$ when $\langle x: e \rangle$ is a condition pattern. Since the result of $F(\langle x: e \rangle)$ is a condition pattern this is equal to $c(M(\langle x: e \rangle)) + c(M(\langle e: e \rangle)) = M(\langle x: e \rangle)$.

• Assume $a: e$. Then $F(\langle x: e \rangle) = F(\langle x: e \rangle)$ by induction equals $M(\langle x: e \rangle) + M(\langle e: e \rangle)$ when $\langle x: e \rangle$ is a condition pattern. Since the result of $F(\langle x: e \rangle)$ is a condition pattern this is equal to $c(M(\langle x: e \rangle)) + c(M(\langle e: e \rangle)) = M(\langle x: e \rangle)$.

Theorem 4.4. If a CXQ* expressions $\langle x: e \rangle$ can be rewritten to the TPNF expressions $\langle x: e \rangle$ then $F(\langle x: e \rangle) = F(\langle x: e \rangle)$.

Proof. By Lemma 4.7 the result of mapping $M$ remains the same after every rewrite and therefore $M(\langle x: e \rangle) = M(\langle x: e \rangle) = M(\langle x: e \rangle)$. Then it follows by Lemma 4.8 that $F(\langle x: e \rangle) = F(\langle x: e \rangle)$.

5 An Alternative Normal Form

If we apply the loop fusion rules in the other direction we get an alternative normal form. To be exact we substitute the loop fusion rule with the following rules:
Loop Split
\[\begin{align*}
\text{α for } $\text{dot}$ in $\cup e_1 \\
&\text{return (α for } $\text{dot}$ in $\cup e_2 \text{ return } ^a e_3) \\
&\text{α for } $\text{dot}$ in $\cup e_1 \text{ return } ^a e_3
\end{align*}\]
If $\cup \preceq \alpha$.

Nested Loop Split
\[\begin{align*}
\text{α for } $\text{dot}$ in $\cup e_1 \\
&\text{return (α if } \forall e_2 \land \ldots \land \forall e_n \text{ then (α for } $\text{dot}$ in $\cup e_{n+1} \text{ return } ^a e_{n+2})) \\
&\text{α for } $\text{dot}$ in $\cup e_1 \text{ return } ^a e_{n+2}
\end{align*}\]
If $\cup \preceq \alpha$, $n > 1$.

Filter Fusion
\[\begin{align*}
\text{α for } $\text{dot}$ in $\cup e_1 \\
&\text{return (α for } $\text{dot}$ in $\cup e_1 \text{ return (α if } \forall e_2 \land \ldots \land \forall e_n \text{ then $\cup$ $\text{dot}}) \\
&\text{return } ^a e_{n+1}
\end{align*}\]
If $\cup \preceq \alpha$.

We conjecture that the resulting set of rules also terminates when applied exhaustively to a CXQ expression of the form $\cup e$ and results in an expression in the following normal form.

Definition 5.1 (TPNF'). Defined by the syntax:
\[\begin{align*}
fp &::= otp | ^a if tp then fp \\
tp &::= atp | $\forall x$ | otp &::= aotp | $\forall$ $\text{x}$ | $\forall$ $\text{dot}$ \\
&\text{\forall for } $\text{dot}$ in tp return rc \\
&\text{\forall for } $\text{dot}$ in tp return orc \\
atp &::= $\forall$ ax::nt | otp &::= $\forall$ ax::nt | $\forall$ $\text{dot}$ \\
&\text{\forall for } $\text{dot}$ in atp return rc \\
&\text{\forall for } $\text{dot}$ in atp return orc \\
rc &::= $\forall$ ax::nt | $\forall if atp then rc$ \\
&\text{\forall if atp then (orc | $\forall$ $\text{dot}$)
\end{align*}\]
where $\forall x$ refers to the set of variables minus $\text{dot}$.

6 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 [Jagadish et al., 2001] or Galax [Michiels et al., 2007] and second, a growing number of advanced evaluation strategies and accompanying indexing systems for tree patterns is being published, for instance the staircase join [Grust et al., 2003] and holistic twig joins [Bruno et al., 2002].

More closely related to our work, the framework presented in [Deutsch et al., 2004] and extended in [Wang et al., 2005], focuses 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 [Arion et al., 2006], 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 [Marian and Siméon, 2003].

Another seemingly useful and promising means for XQuery normalization in general, is the monoid calculus as described for object base query languages [Fegaras and Maier, 2000]. The use of this approach is the subject of further research. In more general terms, the relevance of our work is illustrated by [Wong, 1993], where normal forms open up the road to a better understanding of some formal properties of functional query languages, as well as further optimization opportunities.

7 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 subject of further research. The proposed strategy is complete for an important fragment of the XQuery language and it is complementary to other query optimization approaches like those in NEXT [Deutsch et al., 2004], Galax [Marian and Siméon, 2003] or the view-based rewrites in [Arion et al., 2006]. 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 approach towards the identification and normalization of tree pattern expressions. The presented techniques are designed to easily fit inside any XQuery compiler.

References


34


