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 focusses on tree patterns that correspond with XPath expressions and from now on, we also refer to tree patterns as XPath expressions, i.e., having exactly one node of the pattern as output node for which the result is returned in document order and without duplicates.

Determining whether an arbitrary expression is a tree pattern is far from trivial. For instance, **Q1b** and **Q1c** which are written using a combination of Path expressions and FLWOR expressions,

are equivalent to **Q1a** and therefore are tree patterns. In some cases, subtle changes in the query can affect its semantics in a way that makes it different from a tree pattern. For instance, **Q1n** is almost identical to **Q1b** but does not return the corresponding nodes in document order. As shown in [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
```
$d//person[emailaddress]/name
```

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

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

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

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

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

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

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 $\langle n_1, n_2, n_3 \rangle$. Concatenation of two values v_1 and v_2 is denoted as $v_1 \cdot v_2$.
- Sub/super-value: v is a sub-value of v' if $v' = \langle n_1, \ldots, n_k \rangle$, $\{i_1, \ldots, i_j\} \subseteq \{1, \ldots, k\}$ such that $i_1 < \ldots < i_j$ and $v = \langle n_{i_1}, \ldots, n_{i_j} \rangle$.
- Variable names ($x, $y etc., including a special variable $dot) denoted by variables $\$x, \$y$, $\$x', \ldots, \$x_1, \ldots$
- Variable assignment over a store S, a function that maps variable names to values over S, denoted by variable $\Gamma, \Gamma', \ldots, \Gamma_1, \ldots$. The variable $dot is always mapped to a single node.
- Sub/super-assignment. Variable assignment Γ is a sub-assignment of Γ' if $\Gamma(\$x)$ is a sub-value of $\Gamma'(\$x)$ for each variable $\$x$.

2.2 Forest Patterns

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

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

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

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

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

The semantics of a set of trees is defined given a store S and variable assignment Γ over S. It is defined by *embeddings* which are functions h from the nodes of the pattern to 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 $\Gamma(\mathsf{\$dot})$ 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

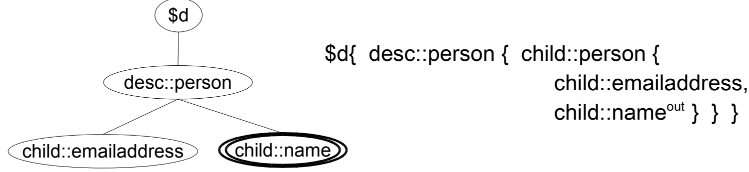

Figure 2: Query **Q1a** as a tree pattern and its textual representation.

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, Γ a variable assignment over S and x the value over S that is the result of evaluating f under S and Γ.

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:*

expr	::=	$\$x$ | *axis*::*ntest* | ddo(*expr*) | if *expr* then *expr* | for $\$x$ in *expr* return *expr*
		| let $\$x$:= *expr* return *expr*
ntest	::=	*label* | *
axis	::=	child | desc | d-o-s

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 $FV(a{::}n) = \{\$\mathsf{dot}\}$. The judgment $S, \Gamma \vdash e \Rightarrow x$ where S a store, Γ a variable assignment over S, e a CXQ expression and x a value over S denotes that x can be the result of the evaluation of e under S and Γ. Two expressions e and e' are said to be equivalent, denoted as $e \equiv e'$, if for every store S and variable assignment Γ over S it holds that $S, \Gamma \vdash e \Rightarrow x$ iff $S, \Gamma \vdash e' \Rightarrow x$.

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

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

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

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

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

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

- $(\textsf{for } \$y \textsf{ in } e_1 \textsf{ return } e_2)[\$x/e'] =$
  $\begin{cases} \textsf{for } \$y \textsf{ in } e_1[\$x/e'] \textsf{ return } e_2 & \textit{if } \$y = \$x \textit{ or } (e_2 = ax{::}nt \textit{ and } \$y = \textsf{\$dot}) \\ \textsf{for } \$z \textsf{ in } e_1[\$x/e'] \textsf{ return } (e_2[\$y/\$z])[\$x/e'] & \textit{otherwise} \end{cases}$
  with $\$z$ *some variable not in* $FV(e')$.

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

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_2 in an expression of the form for \$dot in e_1 return e_2 are replaced with the expression for \$dot in \$dot return $ax{::}nt$. It can then be shown with induction upon the expanded size of e that (1) $e[\$x/e']$ is well-defined and (2) the expanded size of $e[\$x/\$y]$ is equal to the expanded size of e.

3 Tree Pattern Decision for CXQ

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

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

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 $e = a{::}n$: Since Γ' is a super-assignment of Γ it holds that $\Gamma'(\textsf{\$dot}) = \Gamma(\textsf{\$dot})$, 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 $e = \textsf{ddo}(e_1)$: By induction the lemma holds for e_1, and clearly if v is a sub-value of v' then $\textsf{ddo}(v)$ is a sub-value of $\textsf{ddo}(v')$.

- Assume $e = \textsf{if } e_1 \textsf{ then } e_2$: 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 v_1 of e_1 under S and Γ is non-empty and v is the result of e_2 under S and Γ and (2) the result of e_1 under S and Γ is empty and $v = \langle\rangle$. If we assume (1) then, by induction it follows that the result of e_1 under S' and Γ' is non-empty and the result v' of e_2 under S' and Γ' is a super-value of v. By the semantics of e it then holds that v' is the value of e under S' and Γ'. If we assume (2) then the result of e under S' and Γ' is a super-value of v because every value is a super-value of $\langle\rangle$.

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

- Assume $e = \textsf{let } \$x := e_1 \textsf{ return } e_2$: By induction the result v' of e_1 under S' and Γ' is a super-value of the result v under S and Γ. 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 under S and $\Gamma[\$x \mapsto v]$.

□

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

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

- Assume $e = \$x$: If $\Gamma(\$x) = \langle\rangle$ then add a new node to $\Gamma(\$x)$ and to S as a singleton tree.

- Assume $e = a::n$: Then take the node n' in $\Gamma(\$\textsf{dot})$ and add a child under it in S that satisfies the node test n.

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

- Assume $e = \textsf{if } e_1 \textsf{ then } e_2$: By induction there is a super-store S_1 of S and super-assignment Γ_1 of Γ 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 Γ_2 of Γ_1 such that e_2 returns a non-empty result. Moreover, if Γ_2 differs from Γ for $\$x$ then either $\Gamma_1(\$x) = \langle\rangle$, in which case $\Gamma(\$x) = \langle\rangle$, or $\Gamma_1(\$x) \neq \langle\rangle$, in which case $\Gamma_2(\$x) = \Gamma_1(\$x)$ and, since Γ_2 differs from Γ for $\$x$, so does Γ_1 and therefore $\Gamma(\$x) = \langle\rangle$. By Lemma 3.2 it holds that the result of e_1 under S_2 and Γ_2 is also non-empty, so the result of e under S_2 and Γ_2 is indeed non-empty.

- Assume $e = \textsf{for } \$x \textsf{ in } e_1 \textsf{ return } e_2$: By induction there is a super-store S_1 of S and super-assignment Γ_1 of Γ 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 Γ_2 of $\Gamma_1[\$x \mapsto \langle n \rangle]$ such that the result of e_2 under S_2 and Γ_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 Γ for $\$x$ then Γ_1 differs from Γ for $\$x$ and therefore $\Gamma(\$x) = \langle\rangle$. If, on the other hand, $\Gamma_2[\$x \mapsto \Gamma_1(\$x)]$ differs from Γ for $\$y \neq \$x$ then either $\Gamma_1(\$y) = \langle\rangle$ and therefore also $\Gamma(\$y) = \langle\rangle$, or $\Gamma_1(\$y) \neq \langle\rangle$, in which case $\Gamma_2(\$y) = \Gamma_1(\$y)$ and, since Γ_2 differs from Γ for $\$y$, so does Γ_1 and therefore $\Gamma(\$y) = \langle\rangle$.

- Assume $e = \mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_2$: By induction there is a super-store S_1 of S and super-assignment Γ_1 of Γ 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 Γ_2 of $\Gamma_1[\$x \mapsto v_1]$ such that the result of e_2 under S_2 and Γ_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 Γ for $\$x$ then Γ_1 differs from Γ for $\$x$ and therefore $\Gamma(\$x) = \langle\rangle$. If, on the other hand, $\Gamma_2[\$x \mapsto \Gamma_1(\$x)]$ differs from Γ for $\$y \neq \$x$ then either $\Gamma_1(\$y) = \langle\rangle$ and therefore also $\Gamma(\$y) = \langle\rangle$, or $\Gamma_1(\$y) \neq \langle\rangle$, in which case $\Gamma_2(\$y) = \Gamma_1(\$y)$ and, since Γ_2 differs from Γ for $\$y$, so does Γ_1 and therefore $\Gamma(\$y) = \langle\rangle$.

□

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 wether that expression returns ordered and duplicate-free results. A property π holds for an expression e if for any store S and any variable assignment Γ, s.t. $S, \Gamma \vdash e \Rightarrow v$, the list of nodes in v satisfies π.

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

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

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

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 π under the property table T, denoted as $T \vdash e : \pi$, if it holds for every store S and every value assignment Γ over S that satisfies T that if $S, \Gamma \vdash e \Rightarrow x$ then $x : \pi$.*

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

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

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

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

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

- $rv(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_2) = \begin{cases} rv(e_1) & \textit{if}\ rv(e_2) = \$x \\ rv(e_2) & \textit{if}\ rv(e_2) \neq \$x \end{cases}$

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

Theorem 3.1. *For every CXQ expression e there is a CXQ expression e' such that (1)* $\mathsf{ddo}(e) \equiv \mathsf{ddo}(e')$ *and (2) $FV(e') = \{rv(e)\}$ or e' is of the form* $\mathsf{if}\ e_1'\ \mathsf{then}\ e_2'$ *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 premisse $rv(e_2) \neq \$x$, which is present in FORSETNORV and FORORDNORV1, is omitted because it is unnecessary.

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

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

This expression is normalized into CXQ as follows:

```
for $x in ddo(
    for $dot in ddo(
```

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

Assume $d: *no2d, nodup, then by* VarClos *we also know that* $d: *ord, gen. From* DdoStep *and* ChildStep *we know that* child::item: *ord, nodup, gen. From* ForSetRc, ForOrdRv2 *and* ForNodupRv2 *we know that*

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

All these properties are preserved by the surrounding ddo *operation (*DdoSet, DdoSeq*) and they also hold for the surrounding for expression because of* Dot, If, ForOrdRv1 *and* ForNodupRv2*. Similarly, we can derive that*

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

Finally, we use ForOrdRv2 *and* ForNodupRv1 *to derive ord and nodup for the entire expression.*

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

Theorem 3.2 (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 to a single node.

For completeness we consider each property:

no2d If *no2d* is not derived then $\$x \neq$ $dot 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 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 $\$x \neq \mathsf{\$dot}$ and $T(\$x)$ is a subset of $\{gen, nodup\}$ 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 $v : ord$ as follows. In S a tree with root n_1 and two children n_2 and n_3, and $v = \langle n_3, n_2 \rangle$.

nodup If *nodup* is not derived then $\$x \neq \mathsf{\$dot}$ and $T(\$x)$ is a subset of $\{no2d, gen, ord\}$ 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 $v : nodup$ as follows. In S a tree with root n_1, and $v = \langle n_1, n_1 \rangle$.

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 Γ that maps $dot to a value v over S and satisfies all tables T but the result of a::n under Γ does not have property *no2d* as follows. In S a tree with root n_1 and two children n_2 and n_3, and $v = \langle n_1 \rangle$.

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 Γ over S that maps $dot to a value v over S and satisfies all tables T but the result of a::n under Γ 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 $v = \langle n_1 \rangle$. 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 π 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 Γ over S that satisfies T we can always construct a super-store S' and a super-assignment Γ' over S' such that e_1 returns a non-empty result and Γ' also satisfies T. The second observation, as shown by Lemma 3.2, is that the result of e_2 under S' and Γ' will be a super-value of the result under S and Γ. 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 Γ then they also not hold for the result of e under S' and Γ'.

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 Γ 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 π indeed holds for the result of e_2 under S and $\Gamma[\$x \mapsto v]$, 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 (let $\$x := e_1$ return e_2) $\equiv (e_2[\$x/e_1])$ where $e_2[\$x/e_1]$ is expression e_2 with all free occurrences of $\$x$ replaced with e_1. The second observation is that for every property π it holds that the rules derive let $\$x := e_1$ return $e_2 : \pi$ iff 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

$$(T \vdash e_1 : \neg no2d \vee T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : \neg no2d) \\ \wedge \\ (rv(e_2) \neq \$x \vee T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : \neg no2d)$$

which is equivalent to

$$T \vdash e_1 : \neg no2d \wedge rv(e_2) \neq \$x \vee \\ T \vdash e_1 : \neg no2d \wedge T[\$x \mapsto \{\ldots\}] \vdash e_2 : \neg no2d \vee \\ T[\$x \mapsto \{\ldots\}] \vdash e_2 : \neg no2d \wedge rv(e_2) \neq \$x \vee \\ T[\$x \mapsto \{\ldots\}] \vdash e_2 : \neg no2d$$

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 Γ over S that satisfies T:

$$\text{for \$x in } e_1 \text{ return } e_2 : \neg no2d \wedge \\ T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : no2d \\ \Rightarrow \\ rv(e_2) = \$x \wedge T \vdash e_1 : \neg no2d$$

If e_1: *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 $rv(e_2) \neq \$x$, 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 $\$x(\text{/child::*})^n$ 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

$$\text{for \$x in } e_1 \text{ return } e_2 : \neg gen \wedge \\ T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : gen \\ \Rightarrow \\ rv(e_2) = \$x \wedge e_1 : \neg gen$$

If e_1: *gen*, then by Lemma 3.1 and by 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 $\$x(/\text{child::*})^n$ for some $n \geq 0$, it follows that the for expression also has the *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 *gen* property. So once again, the for expression has the *gen* property, which is a contradiction.

Property ***ord*****.**

Lemma 3.4. *For every instance S and an assignment Γ over S that satisfies T, if for a CXQ expression e it holds that $rv(e) = \$x$, where $T \vdash \$x : no2d, nodup$ and the rules derive that $T \vdash e : no2d$ 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 *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 $T \vdash e_2 : no2d$, 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 $T \vdash e_2 : no2d$, 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 Γ over S that satisfies T, if for a CXQ expression e it holds that $rv(e) = \$x$, where $T \vdash \$x : no2d, nodup$ then 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.

□

The *ord* property is derived by four rules: ForOrdRv1, ForOrdRv2, ForOrdNoRv1 and ForOrdNoRv2. Soundness of ForOrdRv1 follows by induction and the fact that if $rv(e_2) = \$x$ and the rules derive that $T[\$x \mapsto \{no2d, nodup\}] \vdash e_2 : no2d$ 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 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 $\$x/\text{d-o-s::*}$, see Lemma 3.5. Soundness of 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 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 ForOrdRv1, ForOrdRv2, ForOrdNoRv1 and ForOrdNoRv2, we need to show that

$$
\begin{aligned}
&(rv(e_2) \neq \$x \vee e_1 : \neg ord \vee e_2 : \neg ord)\\
&\wedge\ (rv(e_2) \neq \$x \vee e_1 : \neg ord \vee e_1 : \neg gen \vee e_1 : \neg nodup \vee e_2 : \neg ord)\\
&\wedge\ (rv(e_2) = \$x \vee e_2 : \neg no2d)\\
&\wedge\ (rv(e_2) = \$x \vee e_1 : \neg no2d \vee e_1 : \neg nodup \vee e_2 : \neg ord)\\
&\Rightarrow \quad (\text{for } \$x \text{ in } e_1 \text{ return } e_2) : \neg ord
\end{aligned}
$$

This can be rewritten as

$$
\begin{aligned}
&(rv(e_2) \neq \$x \wedge e_1 : \neg no2d \wedge e_2 : \neg no2d)\\
&\vee\ (rv(e_2) = \$x \wedge e_1 : \neg ord) \vee (e_1 : \neg ord \wedge e_2 : \neg no2d)\\
&\vee\ (e_1 : \neg gen \wedge e_2 : \neg no2d) \vee (e_1 : \neg nodup \wedge e_2 : \neg no2d)\\
&\vee\ e_2 : \neg ord\\
&\Rightarrow \quad (\text{for } \$x \text{ in } e_1 \text{ return } e_2) : \neg ord
\end{aligned}
$$

- if $rv(e_2) \neq \$x \wedge e_1 : \neg no2d \wedge e_2 : \neg no2d$ 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;
- if $rv(e_2) = \$x \wedge e_1 : \neg ord$ then either the nodes bound to $\$x$ or a subset of the nodes in the results of $\$x/\text{d-o-s::*}$ 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;
- if $e_1 : \neg ord \wedge e_2 : \neg no2d$, then there are two possibilities: if $rv(e_2) = \$x$, then the problem is a special case of the previous one, otherwise, since $e_1 : \neg ord$, the result of e_1 may contain several nodes, causing the result of e_2 to be concatenated several times;
- if $e_1 : \neg gen \wedge e_2 : \neg no2d$: analogous to the previous case;
- if $e_1 : \neg nodup \wedge e_2 : \neg no2d$: analogous to the previous case;
- if $e_2 : \neg ord$ then the result is trivially out of document order.

Property *nodup*. The *nodup* property is derived by three rules: ForNodupRv1, ForNodupRv2 and ForNodupNoRv. The soundness of 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 $\$x/\text{d-o-s::*}$. The soundness for 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 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 ForNodupRv1, ForNodupRv2 and ForNodup-NoRv, we need to show that

$$
\begin{aligned}
&(rv(e_2) \neq \$x \vee e_1 : \neg nodup \vee e_1 : \neg gen \vee e_2 : \neg nodup)\\
&\wedge\ (rv(e_2) \neq \$x \vee e_1 : \neg nodup \vee e_2 : \neg nodup \vee e_2 : \neg gen)
\end{aligned}
$$

$\wedge\ (rv(e_2) = \$x \vee e_1 : \neg no2d \vee e_1 : \neg nodup \vee e_2 : \neg nodup)$
$\Rightarrow \quad (\mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2) : \neg nodup$

which can be rewritten as
$(rv(e_2) \neq \$x \wedge e_1 : \neg no2d) \vee (e_1 : \neg gen \wedge e_1 : \neg gen)$
$\vee e_1 : \neg nodup \vee e_2 : \neg nodup$
$\Rightarrow \quad (\mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2) : \neg nodup$

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

From the definition of the semantics of forest patterns it is clear that the *ord* and *nodup* properties are necessary properties in order for an expression to be equivalent with a forest pattern. In fact, as will be shown in the following section, this is also a sufficient condition and therefore the following theorem holds.

Theorem 3.3. *A CXQ expresssion is equivalent with a forest pattern iff it has the ord and nodup properties.*

The proof of this theorem proceeds by demonstrating that every CXQ expressions of the form ddo(e) can be rewritten to a normal form that directly corresponds to and is equivalent with certain forest patterns. 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 *ord* and *nodup* correspond to a tree pattern and we give an algorithm to obtain this tree pattern. This algorithm is based on rewrite rules that reduce the expression to some normal form. These rewrite rules also derive information about to what extent the exact result of certain subexpression is relevant for the final result of the expression. For example, in an expression of the form ddo(e) the result of e can be changed by adding and/or remove duplicates or change the order of the nodes without affecting the result of the whole expression. Another example is an expression of the form if e_1 then e_2 where the result of e_1 can be changed as long as it is the empty sequence iff the original result of e_1 was empty without affecting the final result. To indicate these properties allow expressions to be annotated. An expression e annotated by α is denoted as ${}^{\alpha}e$ with α either $\cdot$, $\cup$ or $\vee$ which represent the *list concatenation*, *set union* and *boolean disjunction*, respectively. Informally they can be interpreted as saying that the value of the result of e may not be changed (for $\cdot$), the order may be changed and duplicates may be added and removed (for $\cup$), and the result may be changed as long as it stays non-empty iff it was non-empty (for $\vee$). We use variables α, β, etc. to range over annotations. We use variables e, e_1 etc. to range over the expression part of an annotated expression.

For each annotation α we define an *interpretation function* I^{α} that is defined such that (1) $I^{\cdot}(x) = x$, (2) $I^{\cup}(x)$ is the set that contains exactly all nodes in x and (3) $I^{\vee}(x)$ is **false** if x is

the empty sequence and **true** otherwise. These functions define for each annotation equivalence classes over sequences. Observe that for all annotations α it holds that $I^\alpha(e_1)\alpha I^\alpha(e_2) = I^\alpha(e_1 \cdot e_2)$. The semantics of ${}^\alpha e$ is then formally defined as the result of e which is mapped to an α-equivalent sequence. Note that this leads to a non-deterministic semantics.

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

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

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

4.1 The Tree Pattern Normal Form

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

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

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

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

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

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

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

2. $\mathcal{T}^\vee(l) = {}^\vee l$

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

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

5. $\mathcal{T}^\alpha(\$\mathsf{dot}\{t\}) = \mathcal{T}^\alpha(t)$

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

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

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

This expression is mapped by $\mathcal{X}$ to CXQ$^+$ as follows:

```
∪for $dot in ∪$x return (
   ∪for $dot in ∪desc::person return (
      ∪if ∨child::emailaddress
      then (
         ∪for $dot in ∪child::name return (
            ∪if ∨child::middlename then ∪$dot ) ) ) )
```

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

Theorem 4.1. *For all forest patterns f and values x, $S, \Gamma \vdash f \Rightarrow x$ iff $S, \Gamma \vdash \mathsf{ddo}(\mathcal{X}^{\cup}(f)) \Rightarrow x$.*

Proof. We prove with induction upon the structure of f that (1) $S, \Gamma \vdash f \Rightarrow x$ iff $S, \Gamma \vdash \mathsf{ddo}(\mathcal{X}^{\cup}(f)) \Rightarrow x$ and (2) there is an embedding of f under S and Γ iff $S, \Gamma \vdash \mathcal{X}^{\vee}(f) \Rightarrow x$ for seme non-empty sequence x. We extend the textual notation with explicit node identifiers as in $\mathsf{\$x}(n_1)\{\mathsf{desc::person}(n_2)\{\mathsf{child::emailaddress}(n_3), \mathsf{child::name}^{out}(n_4)\}\}$. Consider the cases for $\mathcal{X}^{\alpha}$:

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

2. $\mathcal{X}^{\vee}(\{t_1, \ldots, t_n\}) = {}^{\vee}\mathsf{if}\ \mathcal{T}^{\vee}(t_1)\ \mathsf{then}\ \mathcal{X}^{\vee}(\{t_2, \ldots, t_n\})$ if $n > 1$ and there is no output node in t_1. There is an embedding of $\{t_1, \ldots, t_n\}$ iff there is an embedding of t_1 and of $\{t_2, \ldots, t_n\}$. By induction it holds that this is true iff both $\mathcal{T}^{\vee}(t_1)$ and $\mathcal{X}^{\vee}(\{t_2, \ldots, t_n\})$ return a non-empty sequence. This is true iff ${}^{\vee}\mathsf{if}\ \mathcal{T}^{\vee}(t_1)\ \mathsf{then}\ \mathcal{X}^{\vee}(\{t_2, \ldots, t_n\})$ returns a non-empty sequence.

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

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

Consider the cases for $\mathcal{T}^{\alpha}$:

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

2. $\mathcal{T}^{\vee}(l(n)) = {}^{\vee}l$. If l is of the form $\$x$ then it is clear that the result of ${}^{\vee}\$x$ is non-empty iff there is am embedding of $\$x(n)$. If l is of the form $ax{::}nt$ then the result of ${}^{\vee}ax{::}nt$ 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 $\Gamma(\mathsf{\$dot})$. The latter is true iff there is am embedding of $ax{::}nt(n)$.

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

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

5. $\mathcal{T}^{\vee}(l(n)\{t_1,\ldots,t_n\}) = {}^{\vee}\mathsf{for\ \$dot\ in}\ {}^{\cup}l\ \mathsf{return}\ \mathcal{X}^{\vee}(\{t_1,\ldots,t_n\})$ if $n > 0$ and $l \neq \mathsf{\$dot}$. 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. $\mathcal{T}^{\cup}(\mathsf{\$dot}(n)\{t\}) = \mathcal{T}^{\cup}(t)$. Consider the proposition that there is an embedding of $\mathsf{\$dot}(n)\{t\}$ 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 $\mathcal{T}^{\cup}(t)$.

7. $\mathcal{T}^{\vee}(\mathsf{\$dot}(n)\{t\}) = \mathcal{T}^{\vee}(t)$. 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 $\mathcal{X}^{\cup}$. We will attempt to define this syntax such that (1) for every forrest pattern the result of $\mathcal{X}^{\cup}$ is in the syntax and (2) for every expression in the syntax there is forrest pattern that is mapped to it. That (1) holds can by readily observed by noting that the following holds for the non-terminals: fp describes the range of $\mathcal{X}^{\cup}$, tp describes the range of $\mathcal{T}^{\vee}$, otp describes the range of $\mathcal{T}^{\cup}$, atp describes the range of $\mathcal{T}^{\vee}$ restricted to trees with a $\mathsf{\$dot}$ root, and $aotp$ describes the range of $\mathcal{T}^{\cup}$ restricted to trees with other roots.

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

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

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

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

1. $\mathcal{F}({}^{\cup}\$x) = \$x^{out}$

2. $\mathcal{F}({}^{\vee}\$x) = \$x$

3. $\mathcal{F}({}^{\cup}a{::}n) = \mathsf{\$dot}\{a{::}n^{out}\}$

4. $\mathcal{F}({}^{\vee}a{::}n) = \mathsf{\$dot}\{a{::}n\}$

5. $\mathcal{F}({}^{\alpha}\mathsf{if}\ {}^{\vee}e_1\ \mathsf{then}\ {}^{\alpha}e_2) = \mathcal{F}({}^{\vee}e_1) + \mathcal{F}({}^{\alpha}e_2)$

6. $\mathcal{F}({}^{\alpha}\mathsf{for\ \$dot\ in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\alpha}e_1) = \$x\{t_1,\ldots,t_n\}$
   if $\mathcal{F}({}^{\alpha}e_1) = \{\mathsf{\$dot}\{t_1\},\ldots,\mathsf{\$dot}\{t_n\}\}$

7. $\mathcal{F}({}^{\cup}\textsf{for \$dot in } {}^{\cup}\$x \textsf{ return } {}^{\cup}e_1) = \$x^{out}\{t_1, \ldots, t_n\}$
   if $\mathcal{F}({}^{\cup}e_1) = \{\textsf{\$dot}\{t_1\}, \ldots, \textsf{\$dot}\{t_n\}, \textsf{\$dot}^{out}\}$

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

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

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

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

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

Proof. 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 $\mathcal{F}({}^{\alpha}e) = f$ iff $\mathcal{X}^{\alpha}(f) = {}^{\alpha}e$. 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 $\vee$.

We first consider the cases for fp:

- Assume $fp = otp$. Then by induction $\mathcal{F}^{\cup}(otp) = f$ iff $\mathcal{X}(f) = otp$.
- Assume $fp = {}^{\cup}\textsf{if } tp_1 \textsf{ then } fp_2$. Then $\mathcal{F}({}^{\cup}\textsf{if } tp_1 \textsf{ then } fp_2) = f$ iff it holds that there are f_1 and f_2 such that $f_1 = \mathcal{F}(tp_1)$ and $f_2 = \mathcal{F}(fp_2)$ 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 $\mathcal{F}(tp_1) = f_1$ iff $\mathcal{X}^{\vee}(f_1) = tp_1$ and $\mathcal{F}(fp_2) = f_2$ iff $\mathcal{X}^{\cup}(f_2) = fp_2$. It follows by the definition of $\mathcal{X}$ that this is then true iff $\mathcal{X}^{\cup}(f) = {}^{\cup}\textsf{if } \mathcal{X}^{\vee}(tp_1) \textsf{ then } \mathcal{X}^{\cup}(fp_2)$.

Next we consider the cases for otp:

- Assume $otp = aotp$. By induction $\mathcal{F}^{\cup}(aotp) = f$ iff $\mathcal{X}(f) = aotp$.
- Assume $otp = {}^{\cup}\$x$ such that $\$x \neq \textsf{\$dot}$. Then $\mathcal{F}^{\cup}({}^{\cup}\$x) = \$x^{out}$ and $\mathcal{X}^{\cup}(\$x^{out}) = {}^{\cup}\$x$.
- Assume $otp = {}^{\cup}\textsf{\$dot}$. Then $\mathcal{F}^{\cup}({}^{\cup}\textsf{\$dot}) = \textsf{\$dot}^{out}$ and $\mathcal{X}^{\cup}(\textsf{\$dot}^{out}) = {}^{\cup}\textsf{\$dot}$.
- Assume $otp = {}^{\cup}\textsf{for \$dot in } {}^{\cup}\$x \textsf{ return } orc_1$. Then $\mathcal{F}(orc_1)$ is either of the form $\{\textsf{\$dot}\{t_1\}, \ldots, \textsf{\$dot}\{t_n\}\}$ or $\{\textsf{\$dot}\{t_1\}, \ldots, \textsf{\$dot}\{t_n\}, \textsf{\$dot}^{out}\}$.

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

  In the second case $\mathcal{F}^{\cup}({}^{\cup}\textsf{for \$dot in } {}^{\cup}\$x \textsf{ return } orc_1) = \$x^{out}\{t_1, \ldots, t_n\}$. By induction this is equivalent with $\mathcal{X}^{\cup}(\{\textsf{\$dot}\{t_1\}, \ldots, \textsf{\$dot}\{t_n\}, \textsf{\$dot}^{out}\}) = orc_1$ and since $\mathcal{X}^{\cup}(\{\textsf{\$dot}\{t_1\}, \ldots, \textsf{\$dot}\{t_n\}\}) = \mathcal{X}^{\cup}(\{t_1, \ldots, t_n\})$ this is equivalent with $\mathcal{X}^{\cup}(\$x\{t_1, \ldots, t_n\}) = {}^{\cup}\textsf{for \$dot in } {}^{\cup}\$x \textsf{ return } orc_1$.

Next we consider the cases for $aotp$:

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

- Assume $aotp = {}^{\cup}$for \$dot in ${}^{\cup}a{::}n$ return orc_1. Then $\mathcal{F}(orc_1)$ is either of the form $\{\$\mathsf{dot}\{t_1\}, \ldots, \$\mathsf{dot}\{t_n\}\}$ or $\{\$\mathsf{dot}\{t_1\}, \ldots, \$\mathsf{dot}\{t_n\}, \$\mathsf{dot}^{out}\}$.

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

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

Next we consider the cases for tp:

- Assume $tp = atp$. Then by induction $\mathcal{F}(atp) = f$ iff $\mathcal{X}^{\vee}(f) = atp$.

- Assume $tp = {}^{\vee}\$x$ such that $\$x \neq \$\mathsf{dot}$. Then $\mathcal{F}({}^{\vee}\$x) = \$x$ and $\mathcal{X}^{\vee}(\$x) = {}^{\vee}\$x$.

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

Finally we consider the cases for atp:

- Assume $atp = {}^{\vee}a{::}n$. Then $\mathcal{F}({}^{\vee}a{::}n) = \$\mathsf{dot}\{a{::}n\}$ and $\mathcal{X}^{\vee}(\$\mathsf{dot}\{a{::}n\}) = \mathcal{X}^{\vee}(a{::}n) = {}^{\vee}a{::}n$.

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

□

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

4.2 Normalization Rules

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

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

Annotation Introduction and Propagation, and ddo Removal

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

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

rule may lead to duplication of expressions, and therefore a less efficient query plan, it is only applied to $\cup$ annotated CXQ expressions and therefore the result is guaranteed to be a forest pattern for which there is probably an efficient physical query plan. On the other hand it can be shown that a more conservative substitution rule that only substitutes when the variable appears once in e_2 is not sufficient. Consider, for example, an expression of the form

for \$x in \$y/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 $^{\cup}e$ where all subexpressions in e are annotated with $\cdot$ then the result is in the following syntax:*

se ::= $^{\cup}\$x$ | $^{\cup}ax$::nt | $^{\cup}$if be then se | $^{\cup}$for $\$x$ in se return se
be ::= $^{\vee}\$x$ | $^{\vee}ax$::nt | $^{\vee}$if be then be | $^{\vee}$for $\$x$ in se return be
nt ::= $label$ | *
ax ::=child | desc | 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 $^{\cup}e$ to an expression in TPNF.

Lemma 4.2. *If no rewrite rule applies to a CXQ$^+$ expression $^{\cup}e$ 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

Structural Manipulation

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

Figure 5: An overview of the structural rewrites.

- ${}^{\cup}\$x$ (OK)
- ${}^{\cup}ax{::}nt$ (OK)
- ${}^{\cup}$if be 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 ${}^{\vee}$if $tp_1 \wedge \ldots \wedge tp_n$ then tp_{n+1} (Condition Shift)
- ${}^{\cup}$for \$x in se_1 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 $tp_1 \wedge \ldots \wedge tp_n$ 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 $\$x = \$dot$
    - $\$y \neq \$dot$ and $\$y \in FV(se_2)$
      - se_2 is of form otp and $\$y$ is in $FV(otp)$ (Condition Detection)
      - se_2 is of the form if $tp_1 \wedge \ldots \wedge tp_n$ then otp and $\$y$ is in $FV(otp)$ ((Nested) Return Result Lift)
      - se_2 is of the form if $tp_1 \wedge \ldots \wedge tp_n$ then otp and $\$y$ is in $FV(tp_i), i \neq n$ ((Nested) Return Condition Lift)
    - $FV(se_2) = \{\$dot\}$, thus se_2 is an orc
      - se_1 is of form if $tp_1 \wedge \ldots \wedge tp_n$ 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)

- ${}^{\vee}\$x$
  - if $\$x \neq \$dot$ (OK)
  - else if $\$x = \$dot$ (OK)
- ${}^{\vee}$ax::nt (OK)
- ${}^{\vee}$if be_1 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 $tp_1 \wedge \ldots \wedge tp_n$ then tp_{n+1} (OK)
  - else be_1 is of the form if $tp_1 \wedge \ldots \wedge tp_n$ then tp_{n+1} (Condition Shift)
- ${}^{\vee}$for \$x in se retrun be

- let $\$x \neq \$dot$
  - if $\$dot \notin FV(be)$ then (Dot Introduction)
  - else if $\$x \notin FV(be)$ then (Condition Detection)
  - else, since $\{\$dot, \$x\} \subset FV(be)$ we know that se_2 is of the form if $tp_1 \wedge \ldots \wedge tp_n$ then tp_{n+1} and $\$x$ is either in $FV(tp_i)$ or $FV(otp)$
    - if $\$x \notin FV(tp_i)$ ((Nested) Return Condition Lift)
    - else if $\$x \notin FV(tp_{n+1})$ ((Nested) Return Result Lift)
- let $\$x = \$dot$
  - if be is a $\$dot$ (Trivial Loop)
  - else if $\$y \neq \$dot$ and $\$y \in FV(be)$
    - if be is of the form tp and $\$y \in FV(tp)$ (Condition Detection)
    - else if be is of the form if $tp_1 \wedge \ldots \wedge tp_n$ then tp_{n+1} and $\$y \in FV(tp_{n+1})$ ((Nested) Return Result Lift)
    - else be is of the form if $tp_1 \wedge \ldots \wedge tp_n$ then tp_{n+1} and $\$y \in FV(tp_i)$ ((Nested) Return Condition Lift)
  - if $FV(be) = \{\$dot\}$, then be is an rc
    - if se is of the form if $tp_1 \wedge \ldots \wedge tp_n$ then tp_{n+1} (For Condition Lift)
    - if se is of the form tp
      - if tp is a $\$y$ (OK)
      - else if tp is a ax::nt (OK)
      - else if tp is of the form for $\$dot$ in $\$x$ return orc (Loop Fusion)
      - else if tp is of the form for $\$dot$ in ax::nt return orc (Loop Fusion)

□

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, $\$dot$ is always the loop variable, and all rc/orc expressions have $\$dot$ 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 $\$dot$;
- **(Nested) Return Result Lift** In TPNF, the then-clause in a return-clause must be an rc, orc or $\$dot$. The free variable in an rc, orc, $\$dot$ always is $\$dot$;
- **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 $\$dot$;
- **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 $\$dot$;
- **Shortening Condition** In TPNF, $\$dot$ 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 $\langle w_1, \ldots, w_{15} \rangle$ of natural numbers and maps CXQ expressions to a natural number in the following way:

$$
\begin{array}{lcl}
c(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } e_2) & = & w_1 * c(e_1) + w_2 * c(e_2) + w_3 \\
c(\text{for \$dot in } e_1 \text{ return } e_2) & = & w_4 * c(e_1) + w_5 * c(e_2) + w_6 \\
c(\textsf{if } e_1 \textsf{ then } e_2) & = & w_7 * c(e_1) + w_8 * c(e_2) + w_9 \\
c(ax::nt) & = & w_{10} \\
c(\$x) & = & w_{11} \\
c(\$\text{dot}) & = & w_{12} \\
c(\textsf{let } \$x := e_1 \textsf{ return } e_2) & = & w_{13} * c(e_1) + w_{14} * c(e_2) + w_{15}
\end{array}
$$

where we assume that $\$x \neq \$\textsf{dot}$. Based on this notion of cost function, we define a combined cost function C by an n-tuple of cost functions $\langle c_1, \ldots, c_n \rangle$, where $C(e) = \langle c_1(e), \ldots, c_n(e) \rangle$.

We now give a combined cost function C define by a 5-tuple $\langle c_1, c_2, c_3, c_4, c_5 \rangle$ for which it holds that, when looking at the lexicographical order, the cost for all expressions e diminishes when applying a 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 $c_i(e_2) = c_i(e_1)$.

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

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

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

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

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

The third cost function c_3 is defined by $\langle 2, 1, 1, 2, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0 \rangle$ 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: $4 * c_3(e_1) + 2 * c_3(e_2) + c_3(e_3) + 3 \rightsquigarrow 2 * c_3(e_1) + 2 * c_3(e_2) + c_3(e_3) + 2$

- for condition lift: $c_3(e_1) + c_3(e_2) + c_3(e_3) + 1 \leadsto c_3(e_1) + c_3(e_2) + c_3(e_3) + 1$

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

- condition shift: $c_3(e_1) + c_3(e_2) + c_3(e_3) \leadsto c_3(e_1) + c_3(e_2) + c_3(e_3)$
- (nested) return condition lift: $2 * c_3(e_1) + c_3(e_2) + \ldots + c_3(e_n) + c_3(e_{n+1}) + 1 \leadsto 2 * c_3(e_1) + c_3(e_2) + \ldots + c_3(e_n) + c_3(e_{n+1}) + 1$
- (nested) return result lift: $2 * c_3(e_1) + c_3(e_2) + \ldots + c_3(e_n) + c_3(e_{n+1}) + 1 \leadsto 2 * c_3(e_1) + c_3(e_2) + \ldots + c_3(e_n) + c_3(e_{n+1}) + 1$

The fourth cost function c_4 is defined by $\langle 2, 2, 1, 2, 2, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0\rangle$ 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: $2 * c_4(e_1) + 2 * c_4(e_2) + \ldots + 2 * c_4(e_n) + 2 * c_4(e_{n+1}) + 1 \leadsto 2 * c_4(e_1) + 2 * c_4(e_2) + \ldots + 2 * c_4(e_{n-1}) + c_4(e_n) + 2 * c_4(e_{n+1}) + 1$
- (nested) return result lift: $2 * c_4(e_1) + 2 * c_4(e_2) + \ldots + 2 * c_4(e_n) + 2 * c_4(e_{n+1}) + 1 \leadsto 2 * c_4(e_1) + 2 * c_4(e_2) + \ldots + 2 * c_4(e_n) + c_4(e_{n+1}) + 1$

The value of $c_4(e)$ remains the same when applying:

- condition shift: $c_4(e_1) + c_4(e_2) + c_4(e_3) \leadsto c_4(e_1) + c_4(e_2) + c_4(e_3)$

Finally, the fifth cost function c_5 is defined by $\langle 1, 1, 1, 1, 1, 1, 2, 1, 0, 1, 1, 1, 1, 1, 0\rangle$, which intuitively states that we want to push as much as possible out of the if-clause. The value of $c_3(e)$ strictly diminishes when applying:

- condition shift: $4 * c_5(e_1) + 2 * c_5(e_2) + c_5(e_3) \leadsto 2 * c_5(e_1) + 2 * c_5(e_2) + c_5(e_3)$

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 $^\cup e$ 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 $^{\cdot}\mathsf{ddo}(^{\cdot}e)$ where $^{\cdot}e$ is equal to e except that all subexpressions are annotated with $\cdot$. The rewrite process will rewrite this to the equivalent $^{\cdot}\mathsf{ddo}(^{\cup}e')$ where $^{\cup}e'$ 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 $^{\cup}e'$ will be either of the form *otp* or of the form $\mathsf{if}\ tp_1 \wedge \ldots \wedge tp_n\ \mathsf{then}\ otp$. In the first case it holds that $FV(^{\cup}e')$ 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 $^{\cup}e'$ is either equivalent with $\mathsf{if}\ (\mathsf{if}\ tp_1 \wedge \ldots \wedge tp_{n-1}\ \mathsf{then}\ tp_n)\ \mathsf{then}\ otp$ if $n > 1$, or $\mathsf{if}\ tp_1\ \mathsf{then}\ otp$ 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 $FV(otp)$ 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 $\mathcal{F}$.

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 $\mathsf{\$dot}$ 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 $f_1 \triangleleft^* \{t_1, \ldots, t_n\} = f_1 \triangleleft \{t_1\} + \ldots + f_1 \triangleleft \{t_n\}$ and $f_1 \triangleleft^* \emptyset = \emptyset$.
- $f^{\$x}$ selects from f the trees roots labeled $\$x$.
- $f^{-\$x}$ selects from f the trees with a root not labeled $\$x$.

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

$$\begin{array}{rcl@{\qquad}rcl}
f_1 + f_2 &=& f_2 + f_1 & f_1 + \emptyset &=& f_1 \\
f_1 + (f_2 + f_3) &=& (f_1 + f_2) + f_3 & c(f_1 + f_2) &=& c(f_2) + c(f_1) \\
c(c(f)) &=& c(f) & (f_1 + f_2) \triangleleft f_3 &=& (f_1 \triangleleft f_3) + (f_2 \triangleleft f_3) \\
(f_1 \triangleleft f_2)^{vp} &=& f_1^{vp} \triangleleft f_2 & (f_1 + f_2) \triangleleft^* f_3 &=& (f_1 \triangleleft f_3) + (f_2 \triangleleft^* f_3) \\
(f_1 \triangleleft^* f_2)^{vp} &=& f_1^{vp} \triangleleft^* f_2 & f_1 \triangleleft^* (f_2 + f_3) &=& (f_1 \triangleleft^* f_2) + (f_1 \triangleleft^* f_3) \\
(f_1 + f_2)^{vp} &=& f_1^{vp} + f_2^{vp} & f_1 \triangleleft (f_2 \triangleleft f_3) &=& (f_1 \triangleleft f_2) \triangleleft f_3 \\
c(f)^{vp} &=& c(f^{vp}) & f_1 \triangleleft^* (f_2 \triangleleft^* f_3) &=& (f_1 \triangleleft^* f_2) \triangleleft^* f_3 \\
(f^{\$x})^{-\$x} &=& \emptyset & f_1 \triangleleft (f_2 \triangleleft^* f_3) &=& (f_1 \triangleleft f_2) \triangleleft^* f_3 \\
(f^{\$x})^{\$y} &=& \emptyset \ (\textit{if } \$x \neq \$y) & f_1 \triangleleft^* (f_2 \triangleleft f_3) &=& (f_1 \triangleleft^* f_2) \triangleleft f_3 \\
(f^{vp})^{vp'} &=& (f^{vp'})^{vp} & f \triangleleft \emptyset &=& c(f) \\
(f^{vp})^{vp} &=& f^{vp} & c(\emptyset) &=& \emptyset \\
f \triangleleft^* \emptyset &=& \emptyset & c(f_1 \triangleleft f_2) &=& f_1 \triangleleft c(f_2) \\
c(f_1 \triangleleft^* f_2) &=& f_1 \triangleleft^* c(f_2) & c(f_1) \triangleleft f_2 &=& c(f_1) \\
f &=& (f^{\$x} + f^{-\$x}) & (f^{-\$x})^{\$y} &=& f^{\$y} \qquad (\textit{if } \$x \neq \$y)
\end{array}$$

Proof. The identities follow straightforwardly from the definitions of the operations. □

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

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

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

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

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

Proof. It can be proven with induction on the structure of e that $\mathcal{M}(e)$ 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* $\mathcal{M}(e_1) = \mathcal{M}(e_2)$.

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 $\mathcal{M}({}^{\cdot}\textsf{ddo}({}^{\cdot}e)) = \mathcal{M}({}^{\cdot}e) = \mathcal{M}({}^{\cup}e) = \mathcal{M}({}^{\cdot}\textsf{ddo}({}^{\cup}e))$.
- Assume that $\alpha \preceq \cup$ then $\mathcal{M}({}^{\alpha}\textsf{if } {}^{\beta}e_1 \textsf{ then } {}^{\gamma}e_2) = c(\mathcal{M}({}^{\beta}e_1)) + \mathcal{M}({}^{\gamma}e_2) = \mathcal{M}({}^{\vee}e_1) + \mathcal{M}({}^{\gamma}e_2) = c(\mathcal{M}({}^{\vee}e_1)) + \mathcal{M}({}^{\gamma}e_2) = \mathcal{M}({}^{\alpha}\textsf{if } {}^{\vee}e_1 \textsf{ then } {}^{\gamma}e_2)$.

Annotation Propagation Consider the four rules:

- Since $\mathcal{M}({}^{\cdot}e_1) = \mathcal{M}({}^{\cup}e_1)$ then $\mathcal{M}({}^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\cdot}e_1 \textsf{ return } {}^{\gamma}e_2) = \mathcal{M}({}^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\gamma}e_2)$.
- First consider the case where $\alpha \prec \vee$ and therefore also $\gamma \prec \vee$. Then it follows that $\mathcal{M}({}^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\beta}e_1 \textsf{ return } {}^{\gamma}e_2) = \mathcal{M}({}^{\gamma}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd \mathcal{M}({}^{\gamma}e_2)^{\$x}) = \mathcal{M}({}^{\alpha}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd \mathcal{M}({}^{\alpha}e_2)^{\$x}) = \mathcal{M}({}^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\beta}e_1 \textsf{ return } {}^{\alpha}e_2)$. Next consider the case where $\alpha = \vee$. Then $\mathcal{M}({}^{\vee}\textsf{for } \$x \textsf{ in } {}^{\beta}e_1 \textsf{ return } {}^{\gamma}e_2) = c(\mathcal{M}({}^{\gamma}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd \mathcal{M}({}^{\gamma}e_2)^{\$x})) = c(c(\mathcal{M}({}^{\gamma}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd \mathcal{M}({}^{\gamma}e_2)^{\$x}))) = c(c(\mathcal{M}({}^{\gamma}e_2)^{-\$x}) + c((\mathcal{M}({}^{\beta}e_1) \lhd \mathcal{M}({}^{\gamma}e_2)^{\$x}))) = c(\mathcal{M}({}^{\vee}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd c(\mathcal{M}({}^{\gamma}e_2)^{\$x}))) = c(\mathcal{M}({}^{\vee}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd \mathcal{M}({}^{\vee}e_2)^{\$x})) =$
$\mathcal{M}({}^{\vee}\textsf{for } \$x \textsf{ in } {}^{\beta}e_1 \textsf{ return } {}^{\vee}e_2)$.
- By the same argument as in the previous point except replacing $\lhd$ with $\lhd^*$ it holds that $\mathcal{M}({}^{\alpha}\textsf{let } \$x := {}^{\beta}e_1 \textsf{ return } {}^{\gamma}e_2) = \mathcal{M}({}^{\alpha}\textsf{let } \$x := {}^{\beta}e_1 \textsf{ return } {}^{\alpha}e_2)$
- First consider the case where $\alpha \prec \vee$ and therefore also $\gamma \prec \vee$, then $\mathcal{M}({}^{\gamma}e_2) = \mathcal{M}({}^{\alpha}e_2)$ and therefore $\mathcal{M}({}^{\alpha}\textsf{if } {}^{\beta}e_1 \textsf{ then } {}^{\gamma}e_2) = \mathcal{M}({}^{\alpha}\textsf{if } {}^{\beta}e_1 \textsf{ then } {}^{\alpha}e_2)$. Then consider the case where $\alpha = \vee$. Then $\mathcal{M}({}^{\vee}\textsf{if } {}^{\beta}e_1 \textsf{ then } {}^{\gamma}e_2) = c(c(\mathcal{M}({}^{\beta}e_1)) + \mathcal{M}({}^{\gamma}e_2)) = c(c(\mathcal{M}({}^{\beta}e_1)) + c(\mathcal{M}({}^{\gamma}e_2))) = c(c(\mathcal{M}({}^{\beta}e_1)) + \mathcal{M}({}^{\vee}e_2)) = \mathcal{M}({}^{\vee}\textsf{if } {}^{\beta}e_1 \textsf{ then } {}^{\vee}e_2)$

Removal of ddo Assume that $\alpha \prec \vee$ and therefore $\beta \prec \vee$. Then $\mathcal{M}({}^{\alpha}\textsf{ddo}({}^{\beta}e)) = \mathcal{M}({}^{\beta}e) = \mathcal{M}({}^{\alpha}e)$. Assume that $\alpha = \vee$. Then $\mathcal{M}({}^{\vee}\textsf{ddo}({}^{\beta}e)) = c(\mathcal{M}({}^{\beta}e)) = \mathcal{M}({}^{\vee}e)$.

Substitution We have to show that $\mathcal{M}({}^{\alpha}\textsf{let } \$x := {}^{\beta}e_1 \textsf{ return } {}^{\alpha}e_2) = \mathcal{M}({}^{\alpha}e_2[\$x/{}^{\beta}e_1])$. Since $\mathcal{M}({}^{\vee}\textsf{let } \$x := {}^{\beta}e_1 \textsf{ return } {}^{\vee}e_2) = c(\mathcal{M}({}^{\vee}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd^* \mathcal{M}({}^{\vee}e_2)^{\$x})) = c(c(\mathcal{M}({}^{\cup}e_2))^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd^* c(\mathcal{M}({}^{\cup}e_2))^{\$x})) = c(\mathcal{M}({}^{\cup}e_2)^{-\$x} + (\mathcal{M}({}^{\beta}e_1) \lhd^* \mathcal{M}({}^{\cup}e_2)^{\$x})) =$
$c(\mathcal{M}({}^{\cup}\textsf{let } \$x := {}^{\beta}e_1 \textsf{ return } {}^{\cup}e_2))$ and $\mathcal{M}({}^{\vee}e_2[\$x/{}^{\beta}e_1]) = c(\mathcal{M}({}^{\cup}e_2[\$x/{}^{\beta}e_1]))$ it is sufficient to show this for $\alpha \prec \vee$. 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 $\mathcal{M}(e_2[\$x/e_1]) = \mathcal{M}(e_1)$ and $\mathcal{M}(\textsf{let } \$x := e_1 \textsf{ return } e_2) = (\$x^{out})^{-\$x} + (\mathcal{M}(e_1) \lhd^* (\$x^{out})^{\$x}) = \emptyset + (\mathcal{M}(e_1) \lhd (\$x^{out})) = \mathcal{M}(e_1)$. If $\$x \neq \$y$ then $\mathcal{M}(e_2[\$x/e_1]) = \mathcal{M}(\$y) = \$y^{out}$ and $\mathcal{M}(\textsf{let } \$x := e_1 \textsf{ return } e_2) = \mathcal{M}(\$y)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(\$y)^{\$x}) = \$y^{out} + (\mathcal{M}(e_1) \lhd^* \emptyset) = \$y^{out} + \emptyset = \$y^{out}$.

- Assume $e_2 = a{::}n$. Then $\mathcal{M}(e_2[\$x/e_1]) = \mathcal{M}(a{::}n) = \$\mathsf{dot}\{a{::}n^{out}\}$ and $\mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_2) = (\$\mathsf{dot}\{a{::}n^{out}\})^{-\$x} + (\mathcal{M}(e_1) \lhd^* (\$\mathsf{dot}\{a{::}n^{out}\})^{\$x})$. Since by definition of CXQ it holds that $\$x \neq \$\mathsf{dot}$ this equals $(\$\mathsf{dot}\{a{::}n^{out}\}) + (\mathcal{M}(e_1) \lhd^* \emptyset) = \$\mathsf{dot}\{a{::}n^{out}\} + \emptyset = \$\mathsf{dot}\{a{::}n^{out}\}$.

- Assume $e_2 = \mathsf{ddo}(e_3)$. Then $\mathcal{M}(e_2[\$x/e_1]) = \mathcal{M}(\mathsf{ddo}(e_3[\$x/e_1])) = \mathcal{M}(e_3[\$x/e_1])$ and $\mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_2) = \mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_3)$, and by induction $\mathcal{M}(e_3[\$x/e_1]) = \mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_3)$.

- Assume $e_2 = \mathsf{if}\ e_3\ \mathsf{then}\ e_4$. Then $\mathcal{M}(e_2[\$x/e_1]) = \mathcal{M}(\mathsf{if}\ e_3[\$x/e_1]\ \mathsf{then}\ e_4[\$x/e_1]) =$
  $c(\mathcal{M}(e_3[\$x/e_1])) + \mathcal{M}(e_4[\$x/e_1])$. By induction this is equal to $c(\mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_3)) +$
  $\mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_4)$
  $= c(\mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_3)^{\$x})) + (\mathcal{M}(e_4)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_4)^{\$x}))$
  $= c(\mathcal{M}(e_3))^{-\$x} + (\mathcal{M}(e_1) \lhd^* c(\mathcal{M}(e_3))^{\$x}) + \mathcal{M}(e_4)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_4)^{\$x})$
  $= c(\mathcal{M}(e_3))^{-\$x} + \mathcal{M}(e_4)^{-\$x} + (\mathcal{M}(e_1) \lhd^* c(\mathcal{M}(e_3))^{\$x}) + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_4)^{\$x})$
  $= c(\mathcal{M}(e_3))^{-\$x} + \mathcal{M}(e_4)^{-\$x} + (\mathcal{M}(e_1) \lhd^* (c(\mathcal{M}(e_3))^{\$x} + \mathcal{M}(e_4)^{\$x}))$
  $= (c(\mathcal{M}(e_3)) + \mathcal{M}(e_4))^{-\$x} + (\mathcal{M}(e_1) \lhd^* (c(\mathcal{M}(e_3)) + \mathcal{M}(e_4))^{\$x})$
  $= \mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_2)$.

- Assume $e_2 = \mathsf{for}\ \$y\ \mathsf{in}\ e_3\ \mathsf{return}\ e_4$. If $\$x = \$y$ then it holds that $\mathcal{M}(e_2[\$x/e_1]) =$
  $\mathcal{M}(\mathsf{for}\ \$y\ \mathsf{in}\ e_3[\$x/e_1]\ \mathsf{return}\ e_4) = \mathcal{M}(e_4)^{-\$y} + (\mathcal{M}(e_3[\$x/e_1]) \lhd \mathcal{M}(e_4)^{\$y})$. By induction this is equal to $\mathcal{M}(e_4)^{-\$y} + (\mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_3) \lhd (\mathcal{M}(e_4)^{\$y}))$
  $= \mathcal{M}(e_4)^{-\$y} + ((\mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_3)^{\$x})) \lhd (\mathcal{M}(e_4)^{\$y}))$
  $= \mathcal{M}(e_4)^{-\$y} + ((\mathcal{M}(e_3)^{-\$x} \lhd (\mathcal{M}(e_4)^{\$y})) + ((\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_3)^{\$x}) \lhd (\mathcal{M}(e_4)^{\$y})))$
  $= (\mathcal{M}(e_4)^{-\$y} + (\mathcal{M}(e_3) \lhd \mathcal{M}(e_4)^{\$y}))^{-\$x} + (\mathcal{M}(e_1) \lhd^* (\mathcal{M}(e_4)^{-\$y} + (\mathcal{M}(e_3) \lhd \mathcal{M}(e_4)^{\$y}))^{\$x})$
  $= \mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_2)$.

  If $\$x \neq \$y$ then we may assume w.l.o.g. that $\$y$ is not in $FV(e_1)$ since the mapping $\mathcal{M}$ is invariant under the choice of iteration variables. Then $\mathcal{M}(e_2[\$x/e_1]) =$
  $\mathcal{M}(\mathsf{for}\ \$y\ \mathsf{in}\ e_3[\$x/e_1]\ \mathsf{return}\ e_4[\$x/e_1]) = \mathcal{M}(e_4[\$x/e_1])^{-\$y} + (\mathcal{M}(e_3[\$x/e_1]) \lhd \mathcal{M}(e_4[\$x/e_1])^{\$y})$.
  By induction this is equal to $\mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_4)^{-\$y} + (\mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_3) \lhd$
  $(\mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_4)^{\$y}))$
  $= (\mathcal{M}(e_4)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_4)^{\$x}))^{-\$y} + ((\mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_3)^{\$x})) \lhd (\mathcal{M}(e_4)^{-\$x} +$
  $(\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_4)^{\$x}))^{\$y})$
  $= (\mathcal{M}(e_4)^{-\$x,-\$y} + (\mathcal{M}(e_1)^{-\$y} \lhd^* \mathcal{M}(e_4)^{\$x})) + ((\mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_3)^{\$x})) \lhd (\mathcal{M}(e_4)^{\$y} +$
  $(\mathcal{M}(e_1)^{\$y} \lhd^* \mathcal{M}(e_4)^{\$x})))$
  $= (\mathcal{M}(e_4)^{-\$x,-\$y} + (\mathcal{M}(e_1)^{-\$y} \lhd^* \mathcal{M}(e_4)^{\$x})) + ((\mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_3)^{\$x})) \lhd (\mathcal{M}(e_4)^{\$y} +$
  $(\emptyset \lhd^* \mathcal{M}(e_4)^{\$x})))$
  $= (\mathcal{M}(e_4)^{-\$x,-\$y} + (\mathcal{M}(e_1)^{-\$y} \lhd^* \mathcal{M}(e_4)^{\$x})) + ((\mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_3)^{\$x})) \lhd (\mathcal{M}(e_4)^{\$y} +$
  $\emptyset))$
  $= (\mathcal{M}(e_4)^{-\$x,-\$y} + (\mathcal{M}(e_1)^{-\$y} \lhd^* \mathcal{M}(e_4)^{\$x})) + ((\mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_3)^{\$x})) \lhd \mathcal{M}(e_4)^{\$y})$
  $= (\mathcal{M}(e_4)^{-\$x,-\$y} + (\mathcal{M}(e_1)^{-\$y} \lhd^* \mathcal{M}(e_4)^{\$x})) + (((\mathcal{M}(e_3)^{-\$x} \lhd \mathcal{M}(e_4)^{\$y}) + ((\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_3)^{\$x}) \lhd$
  $\mathcal{M}(e_4)^{\$y})))$
  $= (\mathcal{M}(e_4)^{-\$y,-\$x} + (\mathcal{M}(e_3)^{-\$x} \lhd \mathcal{M}(e_4)^{\$y})) + (((\mathcal{M}(e_1) \lhd^* \mathcal{M}(e_4)^{-\$y,\$x}) + (\mathcal{M}(e_1) \lhd^* (\mathcal{M}(e_3)^{\$x} \lhd$
  $\mathcal{M}(e_4)^{\$y}))))$
  $= (\mathcal{M}(e_4)^{-\$y,-\$x} + (\mathcal{M}(e_3)^{-\$x} \lhd \mathcal{M}(e_4)^{\$y})) + (\mathcal{M}(e_1) \lhd^* ((\mathcal{M}(e_4)^{-\$y,\$x}) + (\mathcal{M}(e_3)^{\$x} \lhd \mathcal{M}(e_4)^{\$y})))$
  $= (\mathcal{M}(e_4)^{-\$y} + (\mathcal{M}(e_3) \lhd \mathcal{M}(e_4)^{\$y}))^{-\$x} + (\mathcal{M}(e_1) \lhd^* (\mathcal{M}(e_4)^{-\$y} + (\mathcal{M}(e_3) \lhd \mathcal{M}(e_4)^{\$y}))^{\$x})$
  $= \mathcal{M}(\mathsf{let}\ \$x := e_1\ \mathsf{return}\ e_2)$.

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

Loop Fusion We need to show that $\mathcal{M}({}^{\alpha}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ ({}^{\cup}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\cup}e_2)\ \mathsf{return}\ {}^{\alpha}e_3)$ is equal to $\mathcal{M}({}^{\alpha}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ ({}^{\alpha}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_2\ \mathsf{return}\ {}^{\alpha}e_3))$. Observe that
$\mathcal{M}({}^{\vee}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ ({}^{\cup}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\cup}e_2)\ \mathsf{return}\ {}^{\vee}e_3) =$
$c(\mathcal{M}({}^{\cup}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ ({}^{\cup}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\cup}e_2)\ \mathsf{return}\ {}^{\cup}e_3))$ and
$\mathcal{M}({}^{\vee}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ ({}^{\vee}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_2\ \mathsf{return}\ {}^{\vee}e_3)) =$

$c(\mathcal{M}(^{\cup}\textsf{for \$dot in } {}^{\cup}e_1 \textsf{ return } (^{\cup}\textsf{for \$dot in } {}^{\cup}e_2 \textsf{ return } {}^{\cup}e_3)))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations in the following.

$\mathcal{M}(\textsf{for \$dot in } (\textsf{for \$dot in } e_1 \textsf{ return } e_2) \textsf{ return } e_3)$
$= \mathcal{M}(e_3)^{-\$dot} + (\mathcal{M}(\textsf{for \$dot in } e_1 \textsf{ return } e_2) \triangleleft \mathcal{M}(e_3)^{\$dot})$
$= \mathcal{M}(e_3)^{-\$dot} + ((\mathcal{M}(e_2)^{-\$dot} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_2)^{\$dot})) \triangleleft \mathcal{M}(e_3)^{\$dot})$
$= \mathcal{M}(e_3)^{-\$dot} + ((\mathcal{M}(e_2)^{-\$dot} \triangleleft \mathcal{M}(e_3)^{\$dot}) + ((\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_2)^{\$dot}) \triangleleft \mathcal{M}(e_3)^{\$dot}))$
$= \mathcal{M}(e_3)^{-\$dot} + (\mathcal{M}(e_2)^{-\$dot} \triangleleft \mathcal{M}(e_3)^{\$dot}) + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_2)^{\$dot} \triangleleft \mathcal{M}(e_3)^{\$dot})$
$= (\mathcal{M}(e_3)^{-\$dot} + (\mathcal{M}(e_2)^{-\$dot} \triangleleft \mathcal{M}(e_3)^{\$dot})) + (\mathcal{M}(e_1) \triangleleft ((\mathcal{M}(e_2)^{\$dot} \triangleleft \mathcal{M}(e_3)^{\$dot})))$
$= (\mathcal{M}(e_3)^{-\$dot,-\$dot} + (\mathcal{M}(e_2)^{-\$dot} \triangleleft \mathcal{M}(e_3)^{\$dot})) + (\mathcal{M}(e_1) \triangleleft (\mathcal{M}(e_3)^{-\$dot,\$dot} + (\mathcal{M}(e_2)^{\$dot} \triangleleft \mathcal{M}(e_3)^{\$dot})))$
$= (\mathcal{M}(e_3)^{-\$dot} + (\mathcal{M}(e_2) \triangleleft \mathcal{M}(e_3)^{\$dot}))^{-\$dot} + (\mathcal{M}(e_1) \triangleleft (\mathcal{M}(e_3)^{-\$dot} + (\mathcal{M}(e_2) \triangleleft \mathcal{M}(e_3)^{\$dot}))^{\$dot})$
$= \mathcal{M}(\textsf{for \$dot in } e_2 \textsf{ return } e_3)^{-\$dot} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(\textsf{for \$dot in } e_2 \textsf{ return } e_3)^{\$dot})$
$= \mathcal{M}(\textsf{for \$dot in } e_1 \textsf{ return } (\textsf{for \$dot in } e_2 \textsf{ return } e_3))$

Condition Detection We need to show that under the assumption that $\$x \notin FV(e_2)$ it holds that $\mathcal{M}(^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\alpha}e_2)$ equals $\mathcal{M}(^{\alpha}\textsf{if } {}^{\vee}e_1 \textsf{ then } {}^{\alpha}e_2)$. Observe that
$\mathcal{M}(^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\vee}e_2) = c(\mathcal{M}(^{\cup}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\cup}e_2))$ and $\mathcal{M}(^{\vee}\textsf{if } {}^{\vee}e_1 \textsf{ then } {}^{\vee}e_2) =$ $c(\mathcal{M}(^{\cup}\textsf{if } {}^{\cup}e_1 \textsf{ then } {}^{\cup}e_2))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations in the following.

$\mathcal{M}(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } e_2) = \mathcal{M}(e_2)^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_2)^{\$x}) = \mathcal{M}(e_2)^{-\$x} + (\mathcal{M}(e_1) \triangleleft \emptyset) = \mathcal{M}(e_2)^{-\$x} + c(\mathcal{M}(e_1)) = c(\mathcal{M}(e_1)) + \mathcal{M}(e_2) = \mathcal{M}(\textsf{if } e_1 \textsf{ then } e_2)$

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

$\mathcal{M}(\textsf{if } (\textsf{if } e_1 \textsf{ then } e_2) \textsf{ then } e_3) = c(c(\mathcal{M}(e_1)) + \mathcal{M}(e_2)) + \mathcal{M}(e_3) = (c(\mathcal{M}(e_1)) + c(\mathcal{M}(e_2))) + \mathcal{M}(e_3) = c(\mathcal{M}(e_1)) + (c(\mathcal{M}(e_2)) + \mathcal{M}(e_3)) = \mathcal{M}(\textsf{if } e_1 \textsf{ then } (\textsf{if } e_2 \textsf{ then } e_3))$

Return Condition Lift We need to show that under the assumption that $\$x \notin FV(e_2)$ it holds that $\mathcal{M}(^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\alpha}\textsf{if } {}^{\vee}e_2 \textsf{ then } {}^{\alpha}e_3)\ )$ is equal to
$\mathcal{M}(^{\alpha}\textsf{if } {}^{\vee}e_2 \textsf{ then } (^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\alpha}e_3))$. Observe that
$\mathcal{M}(^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\vee}\textsf{if } {}^{\vee}e_2 \textsf{ then } {}^{\vee}e_3)\ ) = c(\mathcal{M}(^{\cup}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\cup}\textsf{if } {}^{\cup}e_2 \textsf{ then } {}^{\cup}e_3)\ ))$ and
$\mathcal{M}(^{\vee}\textsf{if } {}^{\vee}e_2 \textsf{ then } (^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\vee}e_3)) = c(\mathcal{M}(^{\cup}\textsf{if } {}^{\cup}e_2 \textsf{ then } (^{\cup}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\cup}e_3)))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations in the following.

$\mathcal{M}(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } (\textsf{if } e_2 \textsf{ then } e_3))$
$= \mathcal{M}(\textsf{if } e_2 \textsf{ then } e_3)^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(\textsf{if } e_2 \textsf{ then } e_3)^{\$x})$
$= (c(\mathcal{M}(e_2)) + \mathcal{M}(e_3))^{-\$x} + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)) + \mathcal{M}(e_3))^{\$x})$
$= (c(\mathcal{M}(e_2)^{-\$x}) + \mathcal{M}(e_3)^{-\$x}) + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)^{\$x}) + \mathcal{M}(e_3)^{\$x}))$
$= (c(\mathcal{M}(e_2)) + \mathcal{M}(e_3)^{-\$x}) + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_3)^{\$x})$
$= c(\mathcal{M}(e_2)) + (\mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_3)^{\$x}))$
$= c(\mathcal{M}(e_2)) + \mathcal{M}(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } e_3)$
$= \mathcal{M}(\textsf{if } e_2 \textsf{ then } (\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } e_3))$

Nested Return Condition Lift We need to show that under the assumption that $\$x \notin FV(e_n)$ it holds that $\mathcal{M}(^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\alpha}\textsf{if } {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n \textsf{ then } {}^{\alpha}e_{n+1}))$ is equal to
$\mathcal{M}(^{\alpha}\textsf{if } {}^{\vee}e_n \textsf{ then } (^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\alpha}\textsf{if } {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_{n-1} \textsf{ then } {}^{\alpha}e_{n+1})))$. Observe that
$\mathcal{M}(^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\vee}\textsf{if } {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n \textsf{ then } {}^{\vee}e_{n+1})) =$
$c(\mathcal{M}(^{\cup}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\cup}\textsf{if } {}^{\cup}e_2 \wedge \ldots \wedge {}^{\cup}e_n \textsf{ then } {}^{\cup}e_{n+1})))$ and
$\mathcal{M}(^{\vee}\textsf{if } {}^{\vee}e_n \textsf{ then } (^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\vee}\textsf{if } {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_{n-1} \textsf{ then } {}^{\vee}e_{n+1}))) =$
$c(\mathcal{M}(^{\vee}\textsf{if } {}^{\vee}e_n \textsf{ then } (^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\vee}\textsf{if } {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_{n-1} \textsf{ then } {}^{\vee}e_{n+1}))))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations in the following.

$\mathcal{M}(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } (\textsf{if } e_2 \textsf{ then } e_3))$
$= \mathcal{M}(\textsf{if } e_2 \wedge \ldots \wedge e_n \textsf{ then } e_{n+1})^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(\textsf{if } e_2 \wedge \ldots \wedge e_n \textsf{ then } e_{n+1})^{\$x})$
$= (c(\mathcal{M}(e_2)) + \ldots + c(\mathcal{M}(e_n)) + \mathcal{M}(e_{n+1}))^{-\$x} + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)) + \ldots + c(\mathcal{M}(e_n)) + \mathcal{M}(e_{n+1}))^{\$x})$
$= (c(\mathcal{M}(e_2)^{-\$x}) + \ldots + c(\mathcal{M}(e_n)^{-\$x}) + \mathcal{M}(e_{n+1})^{-\$x}) + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)^{\$x})) + \ldots + c(\mathcal{M}(e_n)^{\$x})) + \mathcal{M}(e_{n+1})^{\$x}))$
$= (c(\mathcal{M}(e_2)^{-\$x}) + \ldots + c(\mathcal{M}(e_{n-1})^{-\$x} + c(\mathcal{M}(e_n)) + \mathcal{M}(e_{n+1})^{-\$x}) + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)^{\$x})) + \ldots + c(\mathcal{M}(e_{n-1})^{\$x})) + \mathcal{M}(e_{n+1})^{\$x}))$
$= c(\mathcal{M}(e_n)) + (c(\mathcal{M}(e_1)) + \ldots + c(\mathcal{M}(e_{n-1})) + \mathcal{M}(e_{n+1}))^{-\$x} + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_1)) + \ldots + c(\mathcal{M}(e_{n-1})) + \mathcal{M}(e_{n+1}))^{\$x})$
$= c(\mathcal{M}(e_n)) + \mathcal{M}(\textsf{if } e_1 \wedge \ldots \wedge e_{n-1} \textsf{ then } e_{n+1})^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(\textsf{if } e_1 \wedge \ldots \wedge e_{n-1} \textsf{ then } e_{n+1})^{\$x})$
$= c(\mathcal{M}(e_n)) + \mathcal{M}(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } (\textsf{if } e_1 \wedge \ldots \wedge e_{n-1} \textsf{ then } e_{n+1}))$
$= \mathcal{M}(\textsf{if } e_n \textsf{ then } (\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } (\textsf{if } e_1 \wedge \ldots \wedge e_{n-1} \textsf{ then } e_{n+1})))$

Return Result Lift We need to show that if $\$x \notin FV(e_3)$ then it holds that $\mathcal{M}(^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\alpha}\textsf{if } {}^{\vee}e_2 \textsf{ then } {}^{\alpha}e_3))$ is equal to $\mathcal{M}(^{\alpha}\textsf{if } (^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\vee}e_2) \textsf{ then } {}^{\alpha}e_3)$. Observe that $\mathcal{M}(^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\vee}\textsf{if } {}^{\vee}e_2 \textsf{ then } {}^{\vee}e_3)) =$ $c(\mathcal{M}(^{\cup}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\cup}\textsf{if } {}^{\cup}e_2 \textsf{ then } {}^{\cup}e_3)))$ and $\mathcal{M}(^{\vee}\textsf{if } (^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\vee}e_2) \textsf{ then } {}^{\vee}e_3) =$ $c(\mathcal{M}(^{\cup}\textsf{if } (^{\cup}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } {}^{\cup}e_2) \textsf{ then } {}^{\cup}e_3))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations in the following.

$\mathcal{M}(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } (\textsf{if } e_2 \textsf{ then } e_3))$
$= \mathcal{M}(\textsf{if } e_2 \textsf{ then } e_3)^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(\textsf{if } e_2 \textsf{ then } e_3)^{\$x})$
$= (c(\mathcal{M}(e_2)) + \mathcal{M}(e_3))^{-\$x} + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)) + \mathcal{M}(e_3))^{\$x})$
$= (c(\mathcal{M}(e_2)^{-\$x}) + \mathcal{M}(e_3)^{-\$x}) + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)^{\$x}) + \mathcal{M}(e_3)^{\$x}))$
$= (c(\mathcal{M}(e_2)^{-\$x}) + \mathcal{M}(e_3)) + (\mathcal{M}(e_1) \triangleleft c(\mathcal{M}(e_2)^{\$x}))$
$= c(\mathcal{M}(e_2)^{-\$x}) + (\mathcal{M}(e_1) \triangleleft c(\mathcal{M}(e_2)^{\$x})) + \mathcal{M}(e_3)$
$= c(\mathcal{M}(e_2)^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_2)^{\$x})) + \mathcal{M}(e_3)$
$= c(\mathcal{M}(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } e_2)) + \mathcal{M}(e_3)$
$= \mathcal{M}(\textsf{if } (\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } e_2) \textsf{ then } e_3)$

Nested Return Result Lift We have to show that if $\$x \notin FV(e_{n+1})$ then $\mathcal{M}(^{\alpha}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\alpha}\textsf{if } {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n \textsf{ then } {}^{\alpha}e_{n+1}))$ is the same output pattern as $\mathcal{M}(^{\alpha}\textsf{if } (^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\vee}\textsf{if } {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_{n-1} \textsf{ then } {}^{\vee}e_n)) \textsf{ then } {}^{\alpha}e_{n+1})$. Observe that it holds that $\mathcal{M}(^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\vee}\textsf{if } {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n \textsf{ then } {}^{\vee}e_{n+1})) =$ $c(\mathcal{M}(^{\cup}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\cup}\textsf{if } {}^{\cup}e_2 \wedge \ldots \wedge {}^{\cup}e_n \textsf{ then } {}^{\cup}e_{n+1})))$ and $\mathcal{M}(^{\vee}\textsf{if } (^{\vee}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\vee}\textsf{if } {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_{n-1} \textsf{ then } {}^{\vee}e_n)) \textsf{ then } {}^{\vee}e_{n+1}) =$ $c(\mathcal{M}(^{\cup}\textsf{if } (^{\cup}\textsf{for } \$x \textsf{ in } {}^{\cup}e_1 \textsf{ return } (^{\cup}\textsf{if } {}^{\cup}e_2 \wedge \ldots \wedge {}^{\cup}e_{n-1} \textsf{ then } {}^{\cup}e_n)) \textsf{ then } {}^{\cup}e_{n+1}))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations in the following.

$\mathcal{M}(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } (\textsf{if } e_2 \wedge \ldots \wedge e_n \textsf{ then } e_{n+1}))$
$= \mathcal{M}(\textsf{if } e_2 \wedge \ldots \wedge e_n \textsf{ then } e_{n+1})^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(\textsf{if } e_2 \wedge \ldots \wedge e_n \textsf{ then } e_{n+1})^{\$x})$
$= (c(\mathcal{M}(e_2)) + \ldots + c(\mathcal{M}(e_n)) + \mathcal{M}(e_{n+1}))^{-\$x} + (\mathcal{M}(e_1) \triangleleft (c(M(e_2)) + \ldots + c(\mathcal{M}(e_n)) + \mathcal{M}(e_{n+1}))^{\$x})$
$= c(\mathcal{M}(e_2)^{-\$x}) + \ldots + c(\mathcal{M}(e_n)^{-\$x}) + \mathcal{M}(e_{n+1})^{-\$x} + (\mathcal{M}(e_1) \triangleleft (c(M(e_2)^{\$x}) + \ldots + c(\mathcal{M}(e_n)^{\$x}) + \mathcal{M}(e_{n+1})^{\$x}))$
$= c(\mathcal{M}(e_2)^{-\$x}) + \ldots + c(\mathcal{M}(e_n)^{-\$x}) + \mathcal{M}(e_{n+1}) + (\mathcal{M}(e_1) \triangleleft (c(M(e_2)^{\$x}) + \ldots + c(\mathcal{M}(e_n)^{\$x})))$
$= c(\mathcal{M}(e_2)^{-\$x}) + \ldots + c(\mathcal{M}(e_n)^{-\$x}) + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)^{\$x}) + \ldots + c(\mathcal{M}(e_n)^{\$x}))) + \mathcal{M}(e_{n+1})$
$= c((c(\mathcal{M}(e_2)^{-\$x}) + \ldots + c(\mathcal{M}(e_{n-1})^{-\$x}) + \mathcal{M}(e_n)^{-\$x}) + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)^{\$x}) + \ldots + c(\mathcal{M}(e_{n-1})^{\$x}) + \mathcal{M}(e_n)^{\$x}))) + \mathcal{M}(e_{n+1})$
$= c((c(\mathcal{M}(e_2)) + \ldots + c(\mathcal{M}(e_{n-1})) + \mathcal{M}(e_n))^{-\$x} + (\mathcal{M}(e_1) \triangleleft (c(\mathcal{M}(e_2)) + \ldots + c(\mathcal{M}(e_{n-1})) + \mathcal{M}(e_n))^{\$x})) + \mathcal{M}(e_{n+1})$
$= c(\mathcal{M}(\textsf{if } e_2 \wedge \ldots \wedge e_{n-1} \textsf{ then } e_n)^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(\textsf{if } e_2 \wedge \ldots \wedge e_{n-1} \textsf{ then } e_n)^{\$x})) + \mathcal{M}(e_{n+1})$
$= c(\mathcal{M}(\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } (\textsf{if } e_2 \wedge \ldots \wedge e_{n-1} \textsf{ then } e_n))) + \mathcal{M}(e_{n+1})$
$= \mathcal{M}(\textsf{if } (\textsf{for } \$x \textsf{ in } e_1 \textsf{ return } (\textsf{if } e_2 \wedge \ldots \wedge e_{n-1} \textsf{ then } e_n)) \textsf{ then } e_{n+1})$.

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

$\mathcal{M}(\mathsf{for}\ \$x\ \mathsf{in}\ (\mathsf{if}\ e_1\ \mathsf{then}\ e_2)\ \mathsf{return}\ e_3) = \mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2) \triangleleft \mathcal{M}(e_3)^{\$x}) = \mathcal{M}(e_3)^{-\$x} + ((c(\mathcal{M}(e_1)) + \mathcal{M}(e_2)) \triangleleft \mathcal{M}(e_3)^{\$x}) = \mathcal{M}(e_3)^{-\$x} + ((c(\mathcal{M}(e_1)) \triangleleft \mathcal{M}(e_3)^{\$x}) + (\mathcal{M}(e_2) \triangleleft \mathcal{M}(e_3)^{\$x})) = \mathcal{M}(e_3)^{-\$x} + (c(\mathcal{M}(e_1)) + (\mathcal{M}(e_2) \triangleleft \mathcal{M}(e_3)^{\$x})) = c(\mathcal{M}(e_1)) + (\mathcal{M}(e_3)^{-\$x} + (\mathcal{M}(e_2) \triangleleft \mathcal{M}(e_3)^{\$x})) = c(\mathcal{M}(e_1)) + \mathcal{M}(\mathsf{for}\ \$x\ \mathsf{in}\ e_2\ \mathsf{return}\ e_3) = \mathcal{M}(\mathsf{if}\ e_1\ \mathsf{then}\ (\mathsf{for}\ \$x\ \mathsf{in}\ e_2\ \mathsf{return}\ e_3))$:

Trivial Dot Condition We have to show that $\mathcal{M}({}^{\alpha}\mathsf{if}\ {}^{\vee}\$\mathsf{dot}\ \mathsf{then}\ {}^{\alpha}e_2)$ is equal to $\mathcal{M}({}^{\alpha}e_2)$. Observe that $\mathcal{M}({}^{\vee}\mathsf{if}\ {}^{\vee}\$\mathsf{dot}\ \mathsf{then}\ {}^{\vee}e_2) = c(\mathcal{M}({}^{\cup}\mathsf{if}\ {}^{\cup}\$\mathsf{dot}\ \mathsf{then}\ {}^{\cup}e_2))$ and $\mathcal{M}({}^{\vee}e_2) = c(\mathcal{M}({}^{\cup}e_2))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations in the following.

$\mathcal{M}(\mathsf{if}\ \$\mathsf{dot}\ \mathsf{then}\ e_2) = c(\mathcal{M}(\$\mathsf{dot})) + \mathcal{M}(e_2) = c(\{\$\mathsf{dot}^{out}\}) + \mathcal{M}(e_2) = \emptyset + \mathcal{M}(e_2) = \mathcal{M}(e_2)$

Trivial Loop We have to show that $\mathcal{M}({}^{\alpha}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\cup}e\ \mathsf{return}\ {}^{\alpha}\$x)$ is equal to $\mathcal{M}({}^{\alpha}e)$. Observe that $\mathcal{M}({}^{\vee}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\cup}e\ \mathsf{return}\ {}^{\vee}\$x) = c(\mathcal{M}({}^{\cup}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\cup}e\ \mathsf{return}\ {}^{\cup}\$x))$ and $\mathcal{M}({}^{\vee}e) = c(\mathcal{M}({}^{\cup}e))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations in the following.

$\mathcal{M}(\mathsf{for}\ \$x\ \mathsf{in}\ e\ \mathsf{return}\ \$x) = \mathcal{M}(\$x)^{-\$x} + (\mathcal{M}(e) \triangleleft \mathcal{M}(\$x)^{\$x}) = \emptyset + (\mathcal{M}(e) \triangleleft \$x^{out}) = \mathcal{M}(e)$.

Introduction of Dot We have to show that if $\$\mathsf{dot} \notin FV(e_2)$ then $\mathcal{M}({}^{\alpha}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\alpha}e_2)$ is equal to $\mathcal{M}({}^{\alpha}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\alpha}e_2[\$x/{}^{\cup}\$\mathsf{dot}])$. Observe that $\mathcal{M}({}^{\vee}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\vee}e_2) = c(\mathcal{M}({}^{\cup}\mathsf{for}\ \$x\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\cup}e_2))$ and $\mathcal{M}({}^{\vee}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\vee}e_2[\$x/{}^{\cup}\$\mathsf{dot}]) = c(\mathcal{M}({}^{\cup}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\cup}e_2[\$x/{}^{\cup}\$\mathsf{dot}]))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations in the following.

It can be shown with induction upon the structure of e_2 that $\mathcal{M}(e_2[\$x/\$\mathsf{dot}])$ is equal to $\mathcal{M}(e_2)$ except that roots labeled with $\$x$ are relabeled with $\$\mathsf{dot}$. It then follows that:
$\mathcal{M}(\mathsf{for}\ \$x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2) = \mathcal{M}(e_2)^{-\$x} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_2)^{\$x}) = \mathcal{M}(e_2) + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_2)^{\$x}) = \mathcal{M}(e_2) + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_2[\$x/\$\mathsf{dot}])^{\$dot}) = \mathcal{M}(e_2[\$x/\$\mathsf{dot}])^{-\$dot} + (\mathcal{M}(e_1) \triangleleft \mathcal{M}(e_2[\$x/\$\mathsf{dot}])^{\$dot}) = \mathcal{M}(\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2[\$x/\$\mathsf{dot}])$.

Dot Loop We have to show that $\mathcal{M}({}^{\alpha}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}\$\mathsf{dot}\ \mathsf{return}\ {}^{\alpha}e_1)$ equals $\mathcal{M}({}^{\alpha}e_1)$. Observe that $\mathcal{M}({}^{\vee}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}\$\mathsf{dot}\ \mathsf{return}\ {}^{\vee}e_1) = c(\mathcal{M}({}^{\cup}\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ {}^{\cup}\$\mathsf{dot}\ \mathsf{return}\ {}^{\cup}e_1))$ and $\mathcal{M}({}^{\vee}e_1) = c(\mathcal{M}({}^{\cup}e_1))$. Therefore it is sufficient to prove this for $\alpha = \cup$. For brevity we omit the $\cup$ annotations: $\mathcal{M}(\mathsf{for}\ \$\mathsf{dot}\ \mathsf{in}\ \$\mathsf{dot}\ \mathsf{return}\ e_1) = \mathcal{M}(e_1)^{-\$dot} + (\mathcal{M}(\$\mathsf{dot}) \triangleleft \mathcal{M}(e_1)^{\$dot}) = \mathcal{M}(e_1)^{-\$dot} + \mathcal{M}(e_1)^{\$dot} \mathcal{M}(e_1)$

Shortening Condition We have to show that $\mathcal{M}({}^{\vee}\mathsf{if}\ {}^{\vee}e\ \mathsf{then}\ {}^{\vee}\$\mathsf{dot})$ equals $\mathcal{M}({}^{\vee}e)$:
$\mathcal{M}({}^{\vee}\mathsf{if}\ {}^{\vee}e\ \mathsf{then}\ {}^{\vee}\$\mathsf{dot}) = c(c(\mathcal{M}({}^{\vee}e)) + \mathcal{M}({}^{\vee}\$\mathsf{dot})) = c(\mathcal{M}({}^{\vee}e) + c(\mathcal{M}({}^{\vee}\$\mathsf{dot}))) = c(\mathcal{M}({}^{\vee}e) + c(c(\mathcal{M}({}^{\cup}\$\mathsf{dot})))) = \mathcal{M}({}^{\vee}e) + c(\emptyset) = \mathcal{M}({}^{\vee}e)$

□

Lemma 4.8. *For every CXQ$^+$ expression ${}^{\cup}e$ it holds that if $\mathcal{F}({}^{\cup}e)$ is defined then $\mathcal{F}({}^{\cup}e) = \mathcal{M}({}^{\cup}e)$.*

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

- Assume ${}^{\alpha}e = {}^{\cup}\$x$. Then $\mathcal{F}({}^{\cup}\$x) = \$x^{out} = \mathcal{M}({}^{\cup}\$x)$.
- Assume ${}^{\alpha}e = {}^{\vee}\$x$. Then $\mathcal{F}({}^{\vee}\$x) = \$x = c(\$x^{out})) = c(\mathcal{M}({}^{\cup}\$x)) = \mathcal{M}({}^{\vee}\$x)$.
- Assume ${}^{\alpha}e = {}^{\cup}a{::}n$. Then $\mathcal{F}({}^{\cup}a{::}n) = \$\mathsf{dot}\{a{::}n^{out}\} = \mathcal{M}({}^{\cup}\$x)$.

- Assume ${}^{\alpha}e = {}^{\vee}a{::}n$. Then $\mathcal{F}({}^{\vee}a{::}n) = \mathsf{\$dot}\{a{::}n\} = c(\mathsf{\$dot}\{a{::}n^{out}\}) = c(\mathcal{M}({}^{\cup}\$x)) = \mathcal{M}({}^{\vee}\$x)$.

- Assume ${}^{\alpha}e = {}^{\cup}\mathsf{if}\ {}^{\vee}e_1\ \mathsf{then}\ {}^{\cup}e_2$. Then $\mathcal{F}({}^{\cup}\mathsf{if}\ {}^{\vee}e_1\ \mathsf{then}\ {}^{\cup}e_2) = \mathcal{F}({}^{\vee}e_1) + \mathcal{F}({}^{\cup}e_2)$ which by induction equals $\mathcal{M}({}^{\cup}e_1) + \mathcal{M}({}^{\cup}e_2)$. Since the result of $\mathcal{F}({}^{\vee}e_1)$ is a condition pattern this is equal to $c(\mathcal{M}({}^{\vee}e_1)) + \mathcal{M}({}^{\cup}e_2) = \mathcal{M}({}^{\cup}\mathsf{if}\ {}^{\vee}e_1\ \mathsf{then}\ {}^{\cup}e_2)$

- Assume ${}^{\alpha}e = {}^{\vee}\mathsf{if}\ {}^{\vee}e_1\ \mathsf{then}\ {}^{\vee}e_2$. Then $\mathcal{F}({}^{\vee}\mathsf{if}\ {}^{\vee}e_1\ \mathsf{then}\ {}^{\vee}e_2) = \mathcal{F}({}^{\vee}e_1) + \mathcal{F}({}^{\cup}e_2)$ which by induction equals $\mathcal{M}({}^{\cup}e_1) + \mathcal{M}({}^{\cup}e_2)$. Since the result of $\mathcal{F}({}^{\vee}e_i)$ is a condition pattern this is equal to $c(\mathcal{M}({}^{\vee}e_1)) + c(\mathcal{M}({}^{\cup}e_2)) = c(c(\mathcal{M}({}^{\vee}e_1)) + \mathcal{M}({}^{\cup}e_2)) = \mathcal{M}({}^{\vee}\mathsf{if}\ {}^{\vee}e_1\ \mathsf{then}\ {}^{\cup}e_2)$

- Assume ${}^{\alpha}e = {}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\cup}e_1$ and $\mathcal{F}({}^{\cup}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}\}$. Then $\mathcal{F}({}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\cup}e_1) = \$x\{t_1, \ldots, t_n\}$. Since $\mathcal{F}({}^{\cup}e_1)$ produces an output pattern this is equal to $\$x^{out} \triangleleft \mathcal{F}({}^{\cup}e_1) = \mathcal{F}({}^{\cup}e_1)^{-\$dot} + (\$x^{out} \triangleleft \mathcal{F}({}^{\cup}e_1)^{\$dot})$. By induction this equals $\mathcal{M}({}^{\cup}e_1)^{-\$dot} + (\$x^{out} \triangleleft \mathcal{M}({}^{\cup}e_1)^{\$dot}) = \mathcal{M}({}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\cup}e_1)$.

- Assume ${}^{\alpha}e = {}^{\vee}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\vee}e_1$ and $\mathcal{F}({}^{\vee}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}\}$. Then $\mathcal{F}({}^{\vee}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\vee}e_1) = \$x\{t_1, \ldots, t_n\}$. Since $\mathcal{F}({}^{\vee}e_1)$ produces a condition pattern this is equal to $\$x^{out} \triangleleft c(\mathcal{F}({}^{\cup}e_1)) = c(\mathcal{F}({}^{\cup}e_1)^{-\$dot}) + (\$x^{out} \triangleleft c(\mathcal{F}({}^{\cup}e_1)^{\$dot}))$. By induction this equals $c(\mathcal{M}({}^{\cup}e_1)^{-\$dot}) + (\$x^{out} \triangleleft c(\mathcal{M}({}^{\cup}e_1)^{\$dot})) = c(\mathcal{M}({}^{\cup}e_1)^{-\$dot} + (\$x^{out} \triangleleft \mathcal{M}({}^{\cup}e_1)^{\$dot})) = \mathcal{M}({}^{\vee}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\vee}e_1)$.

- Assume ${}^{\alpha}e = {}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\cup}e_1$ and $\mathcal{F}({}^{\cup}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}\}$. Then $\mathcal{F}({}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\cup}e_1) = \mathsf{\$dot}\{a{::}n\{t_1, \ldots, t_n\}\}$. Since $\mathcal{F}({}^{\cup}e_1)$ produces an output pattern this is equal to $\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft \mathcal{F}({}^{\cup}e_1) = \mathcal{F}({}^{\cup}e_1)^{-\$dot} + (\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft \mathcal{F}({}^{\cup}e_1)^{\$dot})$. By induction this equals $\mathcal{M}({}^{\cup}e_1)^{-\$dot} + (\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft \mathcal{M}({}^{\cup}e_1)^{\$dot}) =$
$\mathcal{M}({}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\cup}e_1)$.

- Assume ${}^{\alpha}e = {}^{\vee}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\vee}e_1$ and $\mathcal{F}({}^{\vee}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}\}$. Then $\mathcal{F}({}^{\vee}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\vee}e_1) = \mathsf{\$dot}\{a{::}n\{t_1, \ldots, t_n\}\}$. Since $\mathcal{F}({}^{\vee}e_1)$ produces a condition pattern this is equal to $\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft c(\mathcal{F}({}^{\cup}e_1)) = c(\mathcal{F}({}^{\cup}e_1)^{-\$dot}) + (\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft c(\mathcal{F}({}^{\cup}e_1)^{\$dot}))$. By induction this equals $c(\mathcal{M}({}^{\cup}e_1)^{-\$dot}) + (\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft c(\mathcal{M}({}^{\cup}e_1)^{\$dot})) = c(\mathcal{M}({}^{\cup}e_1)^{-\$dot} + (\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft \mathcal{M}({}^{\cup}e_1)^{\$dot})) = \mathcal{M}({}^{\vee}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\vee}e_1)$.

- Assume ${}^{\alpha}e = {}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\cup}e_1$ and $\mathcal{F}({}^{\cup}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}, \mathsf{\$dot}^{out}\}$. Then $\mathcal{F}({}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\cup}e_1) = \$x^{out}\{t_1, \ldots, t_n\}$. Since $\mathcal{F}({}^{\cup}e_1)$ produces an output pattern where a root is output node this is equal to $\$x^{out} \triangleleft \mathcal{F}({}^{\cup}e_1) = \mathcal{F}({}^{\cup}e_1)^{-\$dot} + (\$x^{out} \triangleleft \mathcal{F}({}^{\cup}e_1)^{\$dot})$. By induction this equals $\mathcal{M}({}^{\cup}e_1)^{-\$dot} + (\$x^{out} \triangleleft \mathcal{M}({}^{\cup}e_1)^{\$dot}) = \mathcal{M}({}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}\$x\ \mathsf{return}\ {}^{\cup}e_1)$.

- Assume ${}^{\alpha}e = {}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\cup}e_1$ and $\mathcal{F}({}^{\cup}e_1) = \{\mathsf{\$dot}\{t_1\}, \ldots, \mathsf{\$dot}\{t_n\}, \mathsf{\$dot}^{out}\}$. Then $\mathcal{F}({}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\cup}e_1) = \mathsf{\$dot}\{a{::}n^{out}\{t_1, \ldots, t_n\}\}$. Since $\mathcal{F}({}^{\cup}e_1)$ produces an output pattern where a root is output node this is equal to $\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft \mathcal{F}({}^{\cup}e_1) = \mathcal{F}({}^{\cup}e_1)^{-\$dot} + (\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft \mathcal{F}({}^{\cup}e_1)^{\$dot})$. By induction this equals
$\mathcal{M}({}^{\cup}e_1)^{-\$dot} + (\mathsf{\$dot}\{a{::}n^{out}\} \triangleleft \mathcal{M}({}^{\cup}e_1)^{\$dot}) = \mathcal{M}({}^{\cup}\mathsf{for}\ \mathsf{\$dot}\ \mathsf{in}\ {}^{\cup}a{::}n\ \mathsf{return}\ {}^{\cup}e_1)$.

□

Theorem 4.4. *If a CXQ^{+} expressions ${}^{\cup}e$ can be rewritten to the TPNF expressions ${}^{\cup}e_1$ and ${}^{\cup}e_2$ then $\mathcal{F}({}^{\cup}e_1) = \mathcal{F}({}^{\cup}e_2)$.*

Proof. By Lemma 4.7 the result of mapping $\mathcal{M}$ remains the same after every rewrite and therefore $\mathcal{M}({}^{\cup}e) = \mathcal{M}({}^{\cup}e_1) = \mathcal{M}({}^{\cup}e_2)$. It then follows by Lemma 4.8 that $\mathcal{F}({}^{\cup}e_1) = \mathcal{F}({}^{\cup}e_2)$. □

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

$$\begin{array}{l} {}^{\alpha}\mathsf{for\ \$dot\ in}\ {}^{\cup}e_1 \\ \mathsf{return}\ ({}^{\alpha}\mathsf{for\ \$dot\ in}\ {}^{\cup}e_2\ \mathsf{return}\ {}^{\alpha}e_3) \end{array} \quad \rightsquigarrow \quad \begin{array}{l} {}^{\alpha}\mathsf{for\ \$dot\ in} \\ \quad ({}^{\cup}\mathsf{for\ \$dot\ in}\ {}^{\cup}e_1\ \mathsf{return}\ {}^{\cup}e_2) \\ \mathsf{return}\ {}^{\alpha}e_3 \end{array}$$

If $\cup \preceq \alpha$.

Nested Loop Split

$$\begin{array}{l} {}^{\alpha}\mathsf{for\ \$dot\ in}\ {}^{\cup}e_1 \\ \mathsf{return} \\ \quad ({}^{\alpha}\mathsf{if}\ {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n \\ \quad \mathsf{then} \\ \qquad ({}^{\alpha}\mathsf{for\ \$dot\ in}\ {}^{\cup}e_{n+1} \\ \qquad \mathsf{return}\ {}^{\alpha}e_{n+2})) \end{array} \quad \rightsquigarrow \quad \begin{array}{l} {}^{\alpha}\mathsf{for\ \$dot\ in} \\ \quad ({}^{\cup}\mathsf{for\ \$dot\ in}\ {}^{\cup}e_1 \\ \quad \mathsf{return} \\ \qquad ({}^{\cup}\mathsf{if}\ {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n \\ \qquad \mathsf{then}\ {}^{\cup}e_{n+1})) \\ \mathsf{return}\ {}^{\alpha}e_{n+2} \end{array}$$

If $\cup \preceq \alpha$, $n > 1$.

Filter Fusion

$$\begin{array}{l} {}^{\alpha}\mathsf{for\ \$dot\ in} \\ \quad ({}^{\cup}\mathsf{for\ \$dot\ in}\ {}^{\cup}e_1 \\ \quad \mathsf{return} \\ \qquad ({}^{\cup}\mathsf{if}\ {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n \\ \qquad \mathsf{then}\ {}^{\cup}\mathsf{\$dot})) \\ \mathsf{return}\ {}^{\alpha}e_{n+1} \end{array} \quad \rightsquigarrow \quad \begin{array}{l} {}^{\alpha}\mathsf{for\ \$dot\ in}\ {}^{\cup}e_1 \\ \mathsf{return}\ (\\ \quad {}^{\alpha}\mathsf{if}\ {}^{\vee}e_2 \wedge \ldots \wedge {}^{\vee}e_n \\ \quad \mathsf{then}\ {}^{\alpha}e_{n+1}) \end{array}$$

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:*

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

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

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], focusses on minimizing navigation within nested subqueries. In contrast to our work, they do not focus on discovering tree patterns inside queries and they ignore existential XPath queries. Hence their approach is fully complementary to our normalization strategy. Quite similarly, a proposed technique for identifying tree patterns [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 to 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 subjkect of further research. The proposed strategy is complete for an important fragment of the XQuery language and it is complementatry to other query optimization approaches like those in NEXT [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 aproach towards the identification and normalization of tree pattern expressions. The presented techniques are designed to easily fit inside any XQuery compiler.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Lu et al., 2005a] Lu, J., Ling, T. W., Chan, C. Y., and Chen, T. (2005a). From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *VLDB*, pages 193–204.

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

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

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

[Wang et al., 2005] Wang, S., Rundensteiner, E. A., and Mani, M. (2005). Optimization of nested XQuery expressions with orderby clauses. In *ICDE Workshops*, page 1277.

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