Advance Access publication on July 11, 2010
doi:10.1093/comjnl/bxq055

A Study of a Positive Fragment of Path Queries: Expressiveness, Normal Form and Minimization

YUQING WU[1], DIRK VAN GUCHT[1], MARC GYSSENS[2,*] AND JAN PAREDAENS[3]

[1]*School of Informatics and Computing, Lindley Hall, Indiana University, Bloomington, IN 47405, USA*
[2]*Faculty of Sciences, Hasselt University & Transnational University of Limburg, Agoralaan, Bldg D, 3590 Diepenbeek, Belgium*
[3]*Department of Mathematics & Computer Science, University of Antwerp, Bldg. G, Middelheimlaan 1, 2020 Antwerpen, Belgium*
**Corresponding author: marc.gyssens@uhasselt.be*

We study the expressiveness of a positive fragment of path queries, denoted Path$^+$, on documents that can be represented as node-labeled trees. The expressiveness of Path$^+$ is studied from two angles. First, we establish that Path$^+$ is equivalent in expressive power to two particular subfragments, as well as to the class of tree queries, a subclass of the first-order conjunctive queries defined over the label, parent–child and child–parent predicates. The translation algorithm from tree queries to Path$^+$ yields a normal form for Path$^+$ queries. Using this normal form, we can decompose a Path$^+$ query into subqueries that can be expressed in a very small fragment of Path$^+$ for which efficient evaluation strategies are available. Second, we characterize the expressiveness of Path$^+$ in terms of its ability to resolve nodes in a document. This result is used to show that each tree query can be translated to a unique, equivalent and minimal tree query. The combination of these results yields an effective strategy to evaluate a large class of path queries on documents.

Keywords: XML; path query; normal form; expressiveness; minimization

Received 4 January 2010; revised 15 April 2010
Handling editor: Leonid Libkin

1. INTRODUCTION

Over the last decade, XQuery [1] has become a standard for declarative querying of XML documents. For our purposes, an XML document is a finite, unordered, node-labeled tree, and a query is a function that associates to a document a set of paths between its nodes, which we represent as pairs of the start and end nodes. More in particular, we are interested in positive XQuery queries in which only the self, parent and child axes are used, and which constitute an important fragment of XQuery (see, e.g. [2, 3]). An example of such a query is shown in Fig. 1.[1]

We can express such queries in an algebraic path query language that we call *Path*$^+$. Path$^+$ allows returning the empty set, label examination, parent/child navigation, composition, first and second projections, intersection and inversion. More precisely, the expressions of Path$^+$ are

$$E := \emptyset \mid \varepsilon \mid \hat{\ell} \mid \downarrow \mid \uparrow \mid E;\, E \mid \Pi_1(E) \mid \Pi_2(E) \mid E \cap E \mid E^{-1},$$

where the primitives '$\emptyset$', 'ε', '$\hat{\ell}$', '$\downarrow$', '$\uparrow$', respectively, return the empty set, the set of paths of length 0, the set of paths of length 0 conditioned by a specific label, the set of paths of length 1 along the child axis, and the set of paths of length 1 along the parent axis, and the operations '$;$', 'Π_1', 'Π_2', '$\cap$' and '$^{-1}$', respectively, denote composition, first projection, second projection, intersection and inversion of sets of paths. The precise semantics of these operations is described in Fig. 4. Projections allow for a clean expression of predication in XQuery, which we do not explicitly add to our language. Path$^+$ is fully capable of expressing the XQuery query in Fig. 1 in an

[1]In the query of Fig. 1, the descendant axis is merely used to indicate that the query may be evaluated at any node of the document under consideration, and not only at the root. As mentioned, the descendant axis does not occur in the actual query.

```
for $i in doc(...)//a/b[*/a]
  for $j in $i/c/*/d[e]
    for $k in $j/*/f
return ($i, $k)
intersect
for $i in doc(...)//a/b
  for $j in $i/c/a/d
    for $k in $j/c/f
return ($i, $k)
```

FIGURE 1. Example of a positive query in XQuery.

algebraic form as

$$\Pi_2(\hat{a}; \downarrow); \hat{b}; \Pi_1(\downarrow; \downarrow; \hat{a}); \downarrow; \hat{c}; \downarrow; \downarrow; \hat{d}; \Pi_1(\downarrow; \hat{e}); \downarrow; \downarrow; \hat{f} \\ \cap \Pi_2(\hat{a}; \downarrow); \hat{b}; \downarrow; \hat{c}; \downarrow; \hat{a}; \downarrow; \hat{d}; \downarrow; \hat{c}; \downarrow; \hat{f},$$

in which we assume the intersection operation to have lower priority than the composition operation.

Besides this algebraic formalism, we shall also consider a declarative formalism, namely a particular class of *tree queries*. Two examples of such tree queries are shown in Fig. 2. Just like documents, tree queries are finite, unordered, labeled trees. Unlike documents, tree queries may contain nodes labeled by a wildcard that intuitively is supposed to be compatible with any label. In addition, two nodes (which may coincide) are marked as, respectively, **s**(ource) and **d**(estination). In general, the source node need not be an ancestor of the destination node. The semantics of a tree query given a document is as follows. We consider all mappings of the nodes of the tree query to the nodes of the document preserving labels and the parent–child relationship. For each such mapping, the tree query returns the pair of nodes of the document on which source and destination have been mapped. Notice that the source node need *not* be mapped to the root of the document. Both tree queries shown in Fig. 2 can be seen to be actually equivalent to the XQuery query in Fig. 1. The tree query in Fig. 2, *left*, is the one most faithfully translating the XQuery query in Fig. 1, but the tree query in Fig. 2, *right*, is obviously simpler.

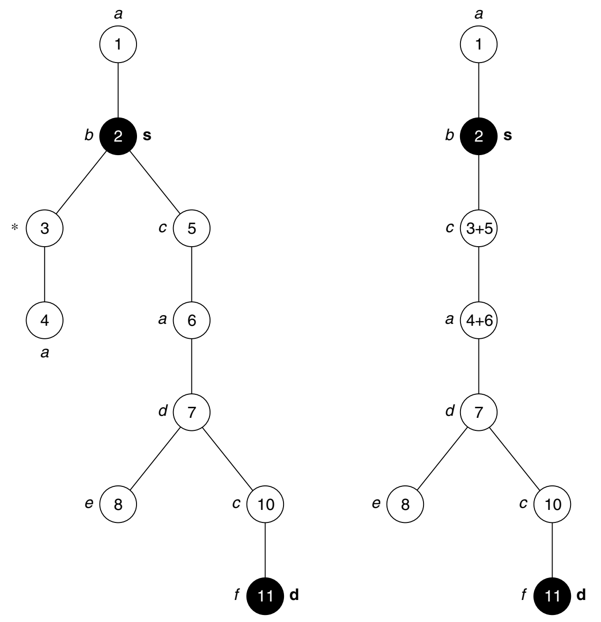


FIGURE 2. Two tree queries. The left one is the straightforward translation of the Path$^+$ expression most faithfully corresponding to the positive XQuery query in Fig. 1. The right one is obtained by reducing the left one to the unique minimal tree query equivalent to it.

In this paper, we study Path$^+$ from the perspective of query expressiveness, resolution expressiveness, minimization, optimization and evaluation. More in particular, we discuss and show the following.

In Section 2, we position our own work in the context of related work. In particular, we position Path$^+$ among related languages considered in the literature.

In Section 3, we present formal definitions of our document model, Path$^+$, and tree queries, and of some related notions.

In Section 4, we establish that Path$^+$ is equivalent in expressive power to two of its fragments, notably Path$^+(\cap)$ and Path$^+(\Pi_1, \Pi_2)$. Besides the primitives and the composition operation, the former fragment only contains the intersection operation, whereas the latter only contains the projection operations. For example, the Path$^+$ expression above is equivalent to the Path$^+(\cap)$ expression

$$(\uparrow; \hat{a}; \downarrow \cap \varepsilon); \hat{b}; (\downarrow; \downarrow; \hat{a}; \uparrow; \uparrow); \downarrow; \hat{c}; \downarrow; \downarrow; \hat{d}; (\downarrow; \hat{e}; \uparrow); \downarrow; \downarrow; \\ \hat{f} \cap (\uparrow; \hat{a}; \downarrow \cap \varepsilon); \hat{b}; \downarrow; \hat{c}; \downarrow; \hat{a}; \downarrow; \hat{d}; \downarrow; \hat{c}; \downarrow; \hat{f}$$

and to the Path$^+(\Pi_1, \Pi_2)$ expression

$$\Pi_2(\hat{a}; \downarrow); \hat{b}; \Pi_1(\downarrow; \downarrow; \hat{a}); \downarrow; \hat{c}; \downarrow; \hat{a}; \downarrow; \hat{d}; \\ \Pi_1(\downarrow; \hat{e}); \downarrow; \hat{c}; \downarrow; \hat{f}.$$

In the process, we also establish that Path$^+$ is equivalent to the class of tree queries we consider. In addition, we present translation algorithms between any pair of languages studied.

In Section 5, we observe and prove that the translation of tree queries into Path$^+(\Pi_1, \Pi_2)$ expressions yields a clean normal form for general Path$^+$ queries. In particular, the parent axis only occurs at the outermost level of nesting with respect to the projection operations. In addition, all parent axes precede all child axes at this level, and the second projection can only occur once, in between the subexpression with the parent axes and the subexpression with the child axes. Notice that this is indeed the case for the Path$^+(\Pi_1, \Pi_2)$ expression above, in which the parent axis does not occur. Finally, we connect our normal form with the structure of the tree query from which it can be derived.

In Section 6, we study the resolution or distinguishability expressiveness of Path$^+$ applied to a particular document. We propose the structural notions of perfect similarity and perfect bisimilarity of nodes of a document, and show that two nodes cannot be distinguished by a Path$^+$ expression if and only if they are perfectly similar or bisimilar, depending on the precise meaning we give to distinguishability. We bootstrap this result to the level of distinguishability of paths by Path$^+$ expressions. In particular, the result of a Path$^+$ expression on a document is the union of blocks of paths which are mutually perfectly bisimilar. We also use these results to show that certain path queries cannot be expressed in Path$^+$.

In Section 7, we present an algorithm to reduce tree queries, and show that this algorithm results in the unique minimal tree query (with respect to the number of nodes) that is equivalent to the given one. In particular, we relate the capability for reduction of a tree query to (a generalization of) perfect similarity between some of its nodes. We also show that the translation of a minimal tree query into a Path$^+$ expression is also minimal, this time with respect to the number of parent and child axes occurring in the expression. For example, the minimal tree query equivalent to the tree query in Fig. 2, *left*, is shown in Fig. 2, *right*. The corresponding minimal Path$^+$ expression is

$$\Pi_2(\hat{a}; \downarrow); \hat{b}; \downarrow; \hat{c}; \downarrow; \hat{a}; \downarrow; \hat{d}; \Pi_1(\downarrow; \hat{e}); \downarrow; \hat{c}; \downarrow; \hat{f}.$$

Finally, we establish a strong connection between minimal Path$^+$ expressions and the normal form presented in Section 5.

In Section 8, We also discuss the ramifications of our work on query decomposition and evaluation of Path$^+$ expressions and more general path queries. On the one hand, we show that there exist index-only evaluation plans to evaluate Path$^+$ queries. On the other hand, we argue that Path$^+$ expressions can be regarded as building blocks for more general path queries, such as those involving set union and set difference, and the ancestor and descendant axes. As such, Path$^+$ can be viewed to path queries, as select-project-join queries are viewed to relational queries.

In Section 9, we discuss some complexity-related issues concerning the algorithms presented in this paper. We argue that they all have low polynomial complexity, i.e. linear of quadratic.

In Section 10, we propose some directions for future work.

2. RELATED WORK

Many researchers have introduced algebraic and logical abstractions in order to study formal aspects of XQuery [1] or XPath [4] and several of its fragments [5–11]. Research on XPath and its sublanguages has been focusing on the expressiveness and the efficient evaluation of these languages. Tree queries or pattern trees are also natural to XML. They have been studied ever since XML and query languages on XML were introduced. Such studies cover areas from minimization [12, 14–18] to efficient evaluation of tree queries [3, 19, 20]. Most of these papers use the standard node-set semantics of XPath.[2] In our work, we focus on the path semantics which is more natural to XQuery, which has already been considered previously in [8–10, 21]. To our knowledge, Path$^+$ as such has not been studied previously. It is a strict subset of Core XPath under path semantics [8], which allows negation in predication, and a strict superset of the language $\mathcal{X}^{\uparrow}_{[\,]}$ of Benedikt *et al.* [6]. In earlier work of some of the present authors [9, 18], the languages under consideration either allowed for some form of negation, or disallowed the simultaneous occurrence of the child and parent axes.

Tree queries or tree patterns have been proposed as an alternative paradigm to query XML documents, but most of these proposals [6, 11–13], when interpreted in the path semantics as we do, only consider tree queries where the context or source node is the root, and, therefore, can only return ancestor–descendant pairs. To our knowledge, only Kimelfeld and Sagiv [16] and ten Cate [10] considered tree queries where the source node need not be the root. However, Kimelfeld and Sagiv did not link tree queries to fragments of XQuery or XPath, whereas ten Cate considered tree queries that are a strict generalization of ours, in that to each node a condition may be associated that contains forms of negation.

Besides proving that Path$^+$ is equivalent to the language of tree queries, we also show that Path$^+$ is equivalent to two of its fragments, Path$^+(\cap)$ (in which, besides the primitives and composition, only intersection is allowed) and Path$^+(\Pi_1, \Pi_2)$ (in which, besides the primitives and composition, only projections are allowed). The latter result can also be interpreted as a closure property under intersection of Path$^+(\Pi_1, \Pi_2)$. Closure under intersection has also been established for Core XPath under both node-set and path semantics [8]. Our result, however, does not follow from these results of Marx and de Rijke.

As far as we are aware, normal forms for XQuery or XPath fragments are only considered by Benedikt *et al.* [6], but for a language that does not consider the parent axis and predication, and by ten Cate [10], for Regular XPath, a language strictly richer than ours. It is interesting to observe that, in ten Cate's normal form, all outer-level parent axes precede all outer-level child axes, as is the case in our normal form. The possible presence of negation in the predicates, however, does not allow ten Cate to extend his normal form to the level of predicates. In contrast, the expressions that are the operands of projection operations in our normal form are of a very restricted form which, e.g. disallows the parent axis and the second projection.

With regard to tree queries, minimization results not unlike ours have been obtained for the special case where the source is the root [7, 12, 13]. Kimelfeld and Sagiv [16], on the other hand, did consider the case where the source node is not the root, and independently obtained the same minimization results as we do [22], however, in a completely different way. More specifically, Kimelfeld and Sagiv obtained their results

[2]The exception is the work of Amer-Yahia *et al.* [14], where tree structures are returned as the result of a query.

by a reduction to Boolean tree queries, whereas we obtain our results by establishing a connection between (a generalization of) perfect similarity, introduced to study the resolution expressiveness of Path$^+$ in Section 6, and the reducibility of tree queries. In addition, we bootstrap the minimization results for tree queries to minimization results for Path$^+$ expressions and establish a link between minimal Path$^+$ expressions and the normal form proposed in Section 5.

3. PRELIMINARIES

In this section, we give the definition of documents, a positive fragment of path queries and the query language of tree queries. Throughout the paper, we assume an infinitely enumerable set $\mathcal{L}$ of *labels*.

3.1. Documents, paths and queries

Throughout this paper, all trees are assumed to be finite and unordered, unless stated otherwise.

DEFINITION 3.1. *A* document *D is a labeled tree (V, Ed, λ), with V the set of nodes, $Ed \subseteq V \times V$ the set of edges, and $\lambda : V \to \mathcal{L}$ a node-labeling function.*

A *query* is a function that associates to a document a set of pairs of its nodes. Now observe that, given an arbitrary pair (m, n) of nodes of a document D, and ignoring the orientation of the edges, there is a unique shortest *path* from m to n, which we shall henceforth refer to as *the* path from m to n. Because of this correspondence, we can interpret a query as associating to a document the paths in the document that it allows.

Given arbitrary nodes m and n of a document D, the least common ancestor of m and n is the 'highest' node on the path from m to n. Therefore, this node will henceforth be denoted as top(m, n).

EXAMPLE 3.1. Figure 3 shows an example of a document that will be used throughout the paper. In this document, e.g. top$(n_8, n_{12}) = n_4$.

3.2. The positive path algebra

Here, we give the formal definition of the Positive Path Algebra, denoted Path$^+$ and its semantics.

DEFINITION 3.2. *Path$^+$ is an algebra which consists of the primitives $\emptyset$, ε, $\hat{\ell}$ ($\ell \in \mathcal{L}$), $\downarrow$ and $\uparrow$, together with the operations composition ($E_1; E_2$), first projection ($\Pi_1(E)$), second projection ($\Pi_2(E)$), intersection ($E_1 \cap E_2$) and inversion (E^{-1}), the semantics of which is described in Fig. 4 (E, E_1, and E_2 represent Path$^+$ expressions).*

With regard to bracketing, we assume that inverse takes precedence over composition and that composition takes precedence over intersection.

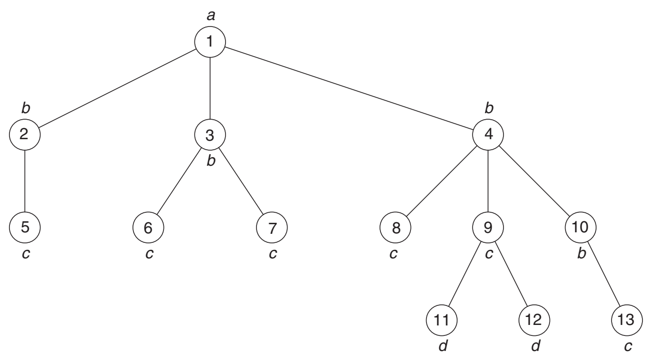


FIGURE 3. Example document.

$$
\begin{array}{lcl}
\emptyset(D) & = & \emptyset\,, \\
\varepsilon(D) & = & \{(n,n) \mid n \in V\}\,, \\
\hat{\ell}(D) & = & \{(n,n) \mid n \in V \;\&\; \lambda(n) = \ell\}\,, \\
\downarrow(D) & = & Ed\,, \\
\uparrow(D) & = & \{(m,n) \mid (n,m) \in Ed\}\,, \\
E_1; E_2(D) & = & \{(m,n) \mid \exists p : (m,p) \in E_1(D) \\
 & & \quad \& \; (p,n) \in E_2(D)\}\,, \\
\Pi_1(E)(D) & = & \{(m,m) \mid \exists n : (m,n) \in E(D)\}\,, \\
\Pi_2(E)(D) & = & \{(n,n) \mid \exists m : (m,n) \in E(D)\}\,, \\
E_1 \cap E_2(D) & = & E_1(D) \cap E_2(D)\,, \\
E^{-1}(D) & = & \{(m,n) \mid (n,m) \in E(D)\}\,.
\end{array}
$$

FIGURE 4. Recursive definition of the semantics of a Path$^+$ expression, given a document $D = (V, Ed, \lambda)$ (E, E_1 and E_2 represent Path$^+$ expressions).

In the remainder of the paper, equality signs between Path$^+$ expressions must be interpreted in the semantic sense rather than the syntactic sense, i.e. for two Path$^+$ expressions E_1 and E_2, we write $E_1 = E_2$ if, for each document D, $E_1(D) = E_2(D)$. For example, $\uparrow^2; \downarrow; \uparrow^4; \downarrow^3 = \uparrow^5; \downarrow^3$. Here, and in the sequel, exponentiation denotes repeated composition.[3]

By restricting the operations allowed in expressions, several subalgebras of Path$^+$ can be defined. The following are of special interest to us:

(1) Path$^+(\cap)$ is the subalgebra of Path$^+$ where, besides the primitives and the composition operation, only intersection is allowed.
(2) Path$^+(\Pi_1, \Pi_2)$ is the subalgebra of Path$^+$ where, besides the primitives and the composition operation, only the first and second projections are allowed.
(3) DPath$^+(\Pi_1)$ is the subalgebra of Path$^+$ where, besides the primitives $\emptyset$, ε, $\hat{\ell}$, $\downarrow$ and the composition operation, only the first projection is allowed.

[3]In particular, for a Path$^+$ expression E, $E^0 = \varepsilon$.

Example 3.2. The following is an example of a Path$^+$ expression:

$$\Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{b}; \downarrow; \hat{c}); \uparrow;$$
$$\Pi_2(\Pi_1((\downarrow; \hat{b}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow); \downarrow; \Pi_1(\hat{c}; \downarrow; \hat{d}); \hat{c}; \downarrow.$$

The semantics of this expression given the document in Fig. 3 is the following set of pairs of nodes of that document: $\{(n_9, n_{11}), (n_9, n_{12})\}$. The above expression is equivalent to the much simpler Path$^+(\Pi_1, \Pi_2)$ expression $\Pi_1(\downarrow; \hat{d}); \hat{c}; \uparrow; \Pi_2(\downarrow); \hat{b}; \downarrow; \Pi_1(\downarrow; \hat{d}); \hat{c}; \downarrow$. The subexpressions $\Pi_1(\downarrow; \hat{d}); \hat{c}$ and $\hat{b}; \downarrow; \Pi_1(\downarrow; \hat{d}); \hat{c}; \downarrow$ are in DPath$^+(\Pi_1)$.

3.3. Signature expressions

A special Path$^+$ expression, defined below, will be used throughout this paper.

Definition 3.3. *Let D be a document and let m and n be arbitrary nodes of D. Then, the* signature *of the pair (m, n), denoted sig(m, n), is the Path$^+$ expression $\uparrow^k; \downarrow^\ell$, with k the length of the path between top(m, n) and m, and ℓ the length of the path between top(m, n) and n.*

The signature of a pair of nodes of a document can be seen as a description of the unique path connecting these nodes, but also as an expression that can be applied to the document under consideration. We shall often exploit this duality.

Notice that, by definition, we have $(m, n) \in \text{sig}(m, n)(D)$, for every document D. Also, $\text{sig}(m, n) = \text{sig}(m, \text{top}(m, n)); \text{sig}(\text{top}(m, n), n)$.

Now, let (m_1, n_1) and (m_2, n_2) be two pairs of nodes in a document D. We say that (m_1, n_1) *subsumes* (m_2, n_2), denoted $(m_1, n_1) \geq (m_2, n_2)$, if (m_2, n_2) is in $\text{sig}(m_1, n_1)(D)$. We say that (m_1, n_1) are (m_2, n_2) *congruent*, denoted $(m_1, n_1) \equiv (m_2, n_2)$, if $(m_1, n_1) \geq (m_2, n_2)$ and $(m_2, n_2) \geq (m_1, n_1)$. It can be easily seen that, in this case, $\text{sig}(m_1, n_1) = \text{sig}(m_2, n_2)$. Informally speaking, the path from m_1 to n_1 has then the same shape as the path from m_2 to n_2.

Example 3.3. In the document in Fig. 3, $\text{sig}(n_5, n_6) = \text{sig}(n_7, n_8) = \uparrow^2; \downarrow^2$, whereas $\text{sig}(n_8, n_9) = \uparrow; \downarrow$. Hence, $(n_5, n_6) \equiv (n_7, n_8)$, whereas $(n_5, n_6) \geq (n_8, n_9)$, but not the other way around.

For later use, but also because they have some interest on their own, we finally note the following fundamental properties of subsumption and congruence.

Proposition 3.1. *Let m, n, m_1, n_1, p_1, m_2, n_2 and p_2 be nodes of a document D. Then the following properties hold.*

(1) $(m, m) \geq (n, n)$.
(2) $(m_1, n_1) \geq (m_2, n_2)$ *implies that* $(n_1, m_1) \geq (n_2, m_2)$.
(3) *If top(m_1, p_1) is also an ancestor of n_1, then $(m_1, n_1) \geq (m_2, n_2)$ and $(n_1, p_1) \geq (n_2, p_2)$ imply that $(m_1, p_1) \geq (m_2, p_2)$.*
(4) *All properties above also hold when subsumption is substituted by congruence.*

Proof. All properties are straightforward, except for Property (3). So, assume that $(m_1, n_1) \geq (m_2, n_2)$ and $(n_1, p_1) \geq (n_2, p_2)$. Hence, $(m_2, n_2) \in \text{sig}(m_1, n_1)(D)$ and $(n_2, p_2) \in \text{sig}(n_1, p_1)(D)$, whence $(m_2, p_2) \in \text{sig}(m_1, n_1)\ ;\ \text{sig}(n_1, p_1)(D)$. For the sake of abbreviation, let $t_1 := \text{top}(m_1, n_1)$ and $u_1 := \text{top}(n_1, p_1)$. Using these nodes, we can write $\text{sig}(m_1, n_1); \text{sig}(n_1, p_1) = \text{sig}(m_1, t_1); \text{sig}(t_1, n_1); \text{sig}(n_1, u_1); \text{sig}(u_1, p_1)$, which is equal to $\text{sig}(m_1, q_1); \text{sig}(q_1, p_1)$, where $q_1 = \text{top}(t_1, u_1)$. Notice that q_1 is a common ancestor of m_1 and p_1, whence it is also an ancestor of $\text{top}(m_1, p_1)$, the least common ancestor of m_1 and p_1. By assumption, $\text{top}(m_1, p_1)$ is a common ancestor of m_1, n_1 and p_1, whence also of $\text{top}(m_1, n_1)$ and $\text{top}(n_1, p_1)$, one of which is q_1. Thus, $q_1 = \text{top}(m_1, p_1)$, and, therefore, $\text{sig}(m_1, q_1); \text{sig}(q_1, p_1) = \text{sig}(m_1, p_1)$. In summary, $(m_2, p_2) \in \text{sig}(m_1, p_1)(D)$, whence $(m_1, p_1) \geq (m_2, p_2)$. □

Example 3.4. Consider again the document in Fig. 3. There, we have that $(n_6, n_7) \geq (n_8, n_8)$ and $(n_7, n_8) \geq (n_8, n_9)$. In addition, $\text{top}(n_6, n_8)$ is an ancestor of n_7. By Proposition 3.1(3), $(n_6, n_8) \geq (n_8, n_9)$, which is indeed the case. The additional condition cannot be omitted, however. To illustrate this, consider the subsumptions $(n_6, n_5) \geq (n_7, n_8)$ and $(n_5, n_7) \geq (n_8, n_9)$. In this case, $\text{top}(n_6, n_7)$ is *not* an ancestor of n_5. Notice that $(n_6, n_7) \ngeq (n_7, n_9)$.

3.4. Tree queries

Here, we define the tree query language, denoted **T**, and its semantics.

Definition 3.4. *A* tree query *is a 3-tuple (T, s, d), with T a labeled tree, and s and d nodes of T, called the* source *and* destination *nodes. The nodes of T are either labeled with a symbol of $\mathcal{L}$ or with a* wildcard *denoted '$*$', which is assumed not to be in $\mathcal{L}$. To the set of all tree queries, we add $\emptyset$. The resulting set of expressions is denoted* **T**.

In order to define the semantics of a tree query, we need the concept of a *containment mapping* between labeled trees, of which we shall provide a slightly more general definition than needed here, for later purposes.

Definition 3.5. *Let $\overline{\mathcal{L}}$ be a set of labels containing $\mathcal{L}$, and assume there is a partial order '$\geq$' on $\overline{\mathcal{L}}$ which is the identity on $\mathcal{L}$. Let $T_1 = (V_1, Ed_1, \lambda_1)$ and $T_2 = (V_2, Ed_2, \lambda_2)$ be labeled trees, with $\lambda_1 : V_1 \to \overline{\mathcal{L}}$ and $\lambda_2 : V_2 \to \overline{\mathcal{L}}$. A* containment mapping *of T_1 into T_2 is a mapping $h : V_1 \to V_2$ such that*

(1) $\forall m, n \in V_1 : (m, n) \in Ed_1 \Rightarrow (h(m), h(n)) \in Ed_2$, *and*
(2) $\forall n \in V_1 : \lambda_1(n) \geq \lambda_2(h(n))$.

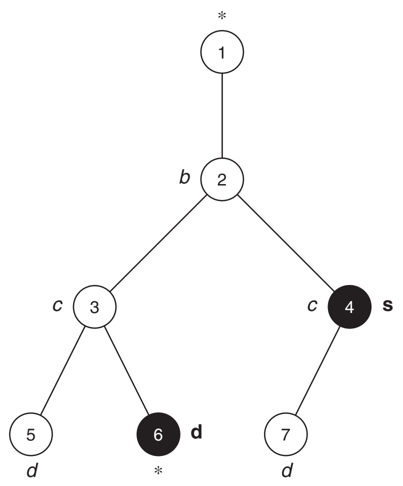


FIGURE 5. Example tree query.

Observe that a containment mapping is in fact a homomorphism with respect to the parent–child and label predicates if the labeled trees do not contain labels outside $\mathcal{L}$.

In general, two symbols $\ell_1, \ell_2 \in \overline{\mathcal{L}}$ for which $\ell_1 \geq \ell_2$ or $\ell_2 \geq \ell_1$ are called *comparable*. Two nodes labeled with comparable symbols are called *compatible*. Finally, if ℓ_1 and ℓ_2 are two comparable symbols of $\overline{\mathcal{L}}$, we denote by $\min(\ell_1, \ell_2)$ the smaller of the two.

In the present context, $\overline{\mathcal{L}} = \mathcal{L} \cup \{*\}$. The partial order on this set is defined as follows: for all $\ell_1, \ell_2 \in \mathcal{L}$, $\ell_1 \geq \ell_2$ if $\ell_1 = \ell_2$ or $\ell_1 = *$. This order enforces that, in the semantics of a tree query defined below, the symbol '$*$' does indeed act as a wildcard.

Definition 3.6. *Let $P = (T, s, d)$ be a tree query, and let D be a document. The semantics of P given D, denoted $P(D)$, is defined as the set*

$$\{(h(s), h(d)) \mid h \text{ is a containment mapping of } T \text{ into } D\}.$$

The semantics of $\emptyset$ on D, i.e. $\emptyset(D)$, is the empty set.

Example 3.5. Figure 5 shows an example of a tree query. The semantics of this tree query given the document in Fig. 3 is the same as the semantics of the Path$^+$ expressions given in Example 3.2, i.e. the set of pairs of nodes $\{(n_9, n_{11}), (n_9, n_{12})\}$. We will show later in the paper that this tree query is actually equivalent to the Path$^+$ expressions given in Example 3.2.

4. EQUIVALENCES OF QUERY LANGUAGES

In this section, we show that Path$^+$, Path$^+(\cap)$, Path$^+(\Pi_1, \Pi_2)$ and **T** are equivalent in expressive power by exhibiting translation algorithms that translate an expression in one language to an equivalent expression in one of the other languages.

Proposition 4.1. *The query languages Path$^+$ and Path$^+(\cap)$ are equivalent in expressive power, and there exists an algorithm translating an arbitrary Path$^+$ expression into an equivalent Path$^+(\cap)$ expression.*

Proof. We provide the translation algorithm, the correctness of which is obvious. This translation algorithm consists of two parts. The first part is a recursive translation of a Path$^+$ expression E to an equivalent expression $\tau(E)$ in the subalgebra Path$^+(\cap, ^{-1})$, where, besides the primitives and the composition operation, only intersection and inverse are allowed. It consists of the following rewriting rules:

$$\begin{aligned}
\tau(E) &= E \text{ if } E \text{ is a primitive expression,}\\
\tau(E_1; E_2) &= \tau(E_1); \tau(E_2),\\
\tau(\Pi_1(E)) &= \tau(E); \tau(E)^{-1} \cap \varepsilon,\\
\tau(\Pi_2(E)) &= \tau(E)^{-1}; \tau(E) \cap \varepsilon,\\
\tau(E^{-1}) &= \tau(E)^{-1},\\
\tau(E_1 \cap E_2) &= \tau(E_1) \cap \tau(E_2).
\end{aligned}$$

The second part of the translation algorithm eliminates the inversion operation using the following rewriting rules:

$$\begin{aligned}
\emptyset^{-1} &= \emptyset,\\
\varepsilon^{-1} &= \varepsilon,\\
\hat{\ell}^{-1} &= \hat{\ell}\ (\ell \in \mathcal{L}),\\
\uparrow^{-1} &= \downarrow,\\
\downarrow^{-1} &= \uparrow,\\
(E_1; E_2)^{-1} &= E_2^{-1}; E_1^{-1},\\
(E^{-1})^{-1} &= E,\\
(E_1 \cap E_2)^{-1} &= E_1^{-1} \cap E_2^{-1}.
\end{aligned}$$

□

Example 4.1. As a first example, consider the Path$^+$ expression $\uparrow; \Pi_1(\uparrow)$. Using the translation algorithm in the proof of Proposition 4.1, we find that this expression is equivalent to $\uparrow; (\uparrow; \downarrow \cap\ \varepsilon)$, an expression of Path$^+(\cap)$.

As a second example, consider again the more complicated Path$^+$ expression

$$\Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{b}; \downarrow; \hat{c}); \uparrow;\\ \Pi_2(\Pi_1((\downarrow; \hat{b}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow); \downarrow; \Pi_1(\hat{c}; \downarrow; \hat{d}); \hat{c}; \downarrow$$

of Example 3.2. Again using the translation algorithm in the proof of Proposition 4.1 and applying some straightforward simplifications, we find the above expression is equivalent to the Path$^+(\cap)$ expression

$$\downarrow; \uparrow; \downarrow; \hat{d}; \uparrow; \hat{c}; (\uparrow; \hat{b}; \downarrow; \hat{c} \cap \varepsilon); \uparrow; ((\uparrow; \uparrow;\\ (\downarrow; \hat{b}; \downarrow \cap \downarrow; \downarrow; \hat{c}) \cap \varepsilon); \uparrow; \uparrow); \downarrow); \downarrow; \hat{c}; \downarrow; \hat{d}; \uparrow; \hat{c}; \downarrow.$$

Next, we are going to show that every Path$^+$ expression can be translated into a tree query (or the empty set). Thereto, we need some preparatory work.

Using that containment mappings are edge-preserving, the following result follows from a straightforward inductive argument.

Lemma 4.1. *Let $T_1 = (V_1, Ed_1, \lambda_1)$, $T_2 = (V_2, Ed_2, \lambda_2)$ and $T = (V, Ed, \lambda)$ be labeled trees. Let $h_1 : V_1 \to V$ and $h_2 : V_2 \to V$ be containment mappings of T_1 into T, respectively, T_2 into T. Let $m_1 \in V_1$ and $m_2 \in V_2$ be nodes of T_1, respectively, T_2, such that $h_1(m_1) = h_2(m_2)$. Let $n_1 \in V_1$ be an ancestor of m_1 and $n_2 \in V_2$ be an ancestor of m_2 such that $sig(m_1, n_1) = sig(m_2, n_2)$. Then $h_1(n_1) = h_2(n_2)$.*

We next describe the algorithms **Merge1** and **Merge2**, exhibited in Figs 6 and 7. **Merge1** tries to transform two given labeled trees into a new one in which a given node from the first tree is merged with a given node of the second tree. **Merge2** is similar to **Merge1** and tries to transform one given labeled tree into a new one in which two nodes of the given tree are merged into a single node.

Example 4.2. Let T_1 and T_2 be the labeled trees in Fig. 8a and b, respectively. Then, **Merge1**(T_1, T_2, n_2, n_6) results in $\emptyset$, since n_2 and n_6 are not compatible. The nodes n_4 and n_8, however, are compatible, as are their parents and grandparents. Therefore, **Merge1**(T_1, T_2, n_4, n_8) results in a labeled tree, which is shown in Fig. 8c. Notice that there are 'canonical embeddings' i_1 and i_2 of T_1, respectively, T_2 into the resulting labeled tree. In our

```
Algorithm Merge1
Input: two disjoint labeled trees
  $T_1 = (V_1, Ed_1, \lambda_1)$; $T_2 = (V_2, Ed_2, \lambda_2)$;
  nodes $m_1 \in V_1$; $m_2 \in V_2$.
Output: a labeled tree or $\emptyset$.
Method:
  let $d_1$ = depth of $m_1$ in $T_1$;
  let $d_2$ = depth of $m_2$ in $T_2$;
  let $d = \min(d_1, d_2)$;
  for $k = 0, \ldots, d$,
    if the level-$k$ ancestors of $m_1$ and $m_2$
      are incompatible return $\emptyset$;
  let $\overline{T} = T_1 \cup T_2$;
  for $k = 0, \ldots, d$,
    in $\overline{T}$, merge the level-$k$ ancestors $m_1^k$
      of $m_1$ and $m_2^k$ of $m_2$ into a node
      $\overline{m}^k$ labeled $\min(\lambda_1(m_1^k), \lambda_2(m_2^k))$;
  return $\overline{T}$.
```

FIGURE 6. Algorithm **Merge1**.

```
Algorithm Merge2
Input: a labeled tree $T = (V, Ed, \lambda)$;
  nodes $m_1, m_2 \in V$;
Output: a labeled tree or $\emptyset$.
Method:
  let $n = \mathrm{top}(m_1, m_2)$;
  if $\mathrm{dist}(n, m_1) \neq \mathrm{dist}(n, m_2)$
    return $\emptyset$;
  let $d = \mathrm{dist}(n, m_1) = \mathrm{dist}(n, m_2)$;
  for $k = 0, \ldots, d-1$,
    if the level-$k$ ancestors of $m_1$ and $m_2$
      are incompatible return $\emptyset$;
  let $\overline{T} = T$;
  for $k = 0, \ldots, d-1$,
    in $\overline{T}$, merge the level-$k$ ancestors $m_1^k$
      of $m_1$ and $m_2^k$ of $m_2$ into a node
      $\overline{m}^k$ labeled $\min(\lambda_1(m_1^k), \lambda_2(m_2^k))$;
  return $\overline{T}$.
```

FIGURE 7. Algorithm **Merge2**.

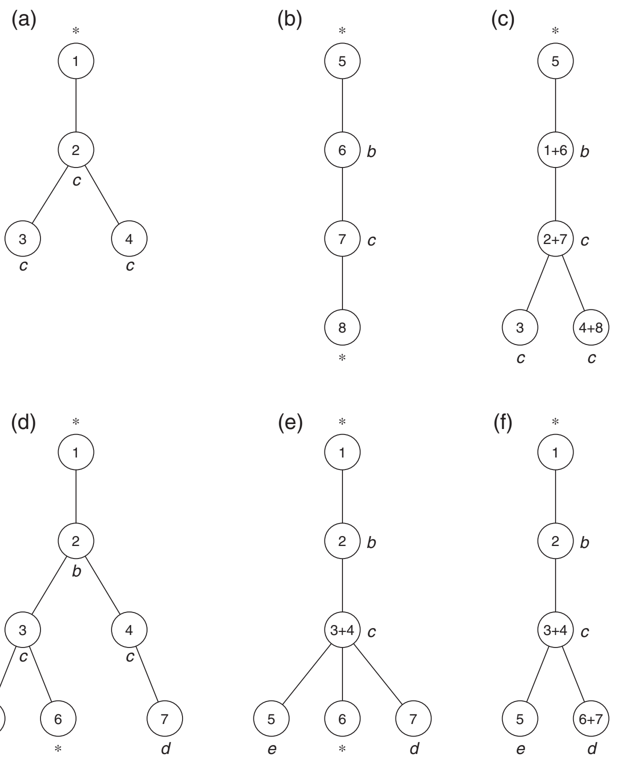


FIGURE 8. Example applications of the algorithms **Merge1** and **Merge2**. Details are provided in Example 4.2.

example, i_1 is defined by the following correspondences of node numbers: $1 \leftrightarrow 1{+}6$, $2 \leftrightarrow 2{+}7$, $3 \leftrightarrow 3$ and $4 \leftrightarrow 4{+}8$; i_2 is defined by the following correspondences of node numbers: $5 \leftrightarrow 5$, $6 \leftrightarrow 1{+}6$, $7 \leftrightarrow 2{+}7$ and $8 \leftrightarrow 4{+}8$.

Next, let T be the labeled tree in Fig. 8d. Then, **Merge2**(T, n_2, n_6) results in $\emptyset$, since n_2 and n_6 are at different depths in the tree, or equivalently, have different distances to their least common ancestor, $\text{top}(n_2, n_6) = n_2$. Also, **Merge2**$(T, n_5, n_7)$ results in $\emptyset$, since n_5 and n_7 are incompatible. The nodes n_3 and n_4, however, are at the same depth and compatible. Therefore, **Merge2**(T, n_3, n_4) results in a labeled tree, which is shown in Fig. 8e. Notice that there is a 'canonical covering' by T of the resulting labeled tree, defined by the following correspondences between node numbers: $1 \to 1$, $2 \to 2$, $3 \to 3{+}4$, $4 \to 3{+}4$, $5 \to 5$, $6 \to 6$ and $7 \to 7$. Similarly, the nodes n_6 and n_7 are at the same depth and compatible, and so are their parents, and, therefore, also **Merge2**(T, n_6, n_7) results in a labeled tree, which is shown in Fig. 8f. In this case, the canonical covering by T of resulting labeled tree is defined by the following correspondences between node numbers: $1 \to 1$, $2 \to 2$, $3 \to 3{+}4$, $4 \to 3{+}4$, $5 \to 5$, $6 \to 6{+}7$ and $7 \to 6{+}7$. Finally, notice that the labeled tree in Fig. 8f, can also be obtained by applying **Merge2** to the nodes n_6 and n_7 of the labeled tree in Fig. 8e.

Suppose **Merge1**(T_1, T_2, m_1, m_2) in Fig. 6 is a labeled tree $\overline{T}$. In Example 4.2, we already illustrated informally that there is a *canonical embedding* i_1 of T_1, respectively, i_2 of T_2 into $\overline{T}$. In general, these canonical embeddings are defined as follows, using the notation in Fig. 6. For $j = 1, 2$, and, for m in V_j,

$$\begin{cases} i_j(m) = m & \text{if } m \neq m_j^k, \text{ for } k = 0, \ldots, d, \\ i_j(m_j^k) = \overline{m}^k & \text{for } k = 0, \ldots, d. \end{cases}$$

From a straightforward argument, it follows that the image of T_1 under i_1, respectively, the image of T_2 under i_2 is a subtree[4] of $\overline{T}$ isomorphic to T_1, respectively, T_2, justifying the term 'embedding'. Together, these isomorphic images of T_1 and T_2 cover $\overline{T}$. We are now ready to prove the following key property of **Merge1**.

LEMMA 4.2. *Let $T_1 = (V_1, Ed_1, \lambda_1)$, $T_2 = (V_2, Ed_2, \lambda_2)$ and $T' = (V', Ed', \lambda')$ be labeled trees, and let $m_1 \in V_1$ and $m_2 \in V_2$ be nodes.*

(1) ***Merge1***(T_1, T_2, m_1, m_2) *is either the empty set or a labeled tree.*

(2) *If there exists a containment mapping $h_1 : V_1 \to V'$ of T_1 into T' and a containment mapping $h_2 : V_2 \to V'$ of T_2 into T' such that $h_1(m_1) = h_2(m_2)$, then* ***Merge1***(T_1, T_2, m_1, m_2) *is a labeled tree, say $\overline{T} = (\overline{V}, \overline{Ed}, \overline{\lambda})$, and there exists a containment mapping $\overline{h} : \overline{V} \to V'$ of $\overline{T}$ into T' such that, for each node $p \in V_1$, $\overline{h}(i_1(p)) = h_1(p)$, and, for each node $p \in V_2$, $\overline{h}(i_2(p)) = h_2(p)$, with $i_1 : V_1 \to \overline{V}$ and $i_2 : V_2 \to \overline{V}$ the canonical embeddings from T_1, respectively, T_2, into $\overline{T}$.*

(3) *If* ***Merge1***(T_1, T_2, m_1, m_2) *is a labeled tree, say $\overline{T} = (\overline{V}, \overline{Ed}, \overline{\lambda})$, and there exists a containment mapping $\overline{h} : \overline{V} \to V'$ of $\overline{T}$ into T', then the mappings $h_1 : V_1 \to V' : p \mapsto \overline{h}(i_1(p))$ and $h_2 : V_2 \to V' : p \mapsto \overline{h}(i_2(p))$, with $i_1 : V_1 \to \overline{V}$ and $i_2 : V_2 \to \overline{V}$ the canonical embeddings of T_1, respectively, T_2 into $\overline{T}$, are containment mappings of T_1 into T', respectively, T_2 into T', such that $h_1(m_1) = h_2(m_2)$.*

Proof. (1) Suppose it has already been established that, for $k = 0, \ldots, d$, the level-k ancestors of m_1 and m_2 are compatible, i.e. that **Merge1**(T_1, T_2, m_1, m_2) is not the empty set. Then, **Merge1** will output a graph $\overline{T}$. Initially, $\overline{T} = T_1 \cup T_2$. In this disconnected graph, every node has indegree 1, except for the two roots, which have indegree 0. If $d = 0$, then it is immediate that, after merging m_1 and m_2, $\overline{T}$ becomes connected and the number of nodes with indegree 0 has been reduced to one. If $d > 0$, let $\overline{T}^k$ be the value of T after the kth iteration of the for-loop. If $0 \leq k < d$, it follows from a straightforward induction that $\overline{T}^k$ is a connected graph in which all nodes have indegree 1, except for the roots of T_1 and T_2, which have indegree 0, and $\overline{m}^k$, which has indegree 2. In particular, this holds for $k = d - 1$. Now, by assumption, at least one of m_1^d and m_2^d has indegree 0, say, m_1^d. Hence, merging m_1^d and m_2^d into $\overline{m}^d$ will decrease the indegree of $\overline{m}^{d-1}$ by one, whereas the indegree of $\overline{m}^d$ will equal the indegree of m_2^d. Hence, also in this case, $\overline{T}^d$, the final result of **Merge1**, is connected, and all its nodes have indegree 1, except for one node, which has indegree 0. Such a graph is necessarily a tree.

(2) Suppose there exist a containment mapping $h_1 : V_1 \to V'$ of T_1 into T' and a containment mapping $h_2 : V_2 \to V'$ of T_2 into T' such that $h_1(m_1) = h_2(m_2)$. By Lemma 4.1, it follows that, for $k = 0, \ldots, d$, $h_1(m_1^k) = h_2(m_2^k)$. This is only possible if m_1^k and m_2^k are compatible. Hence, the result of **Merge1** is non-empty, and, therefore, by Lemma 4.2(1), a labeled tree, say $\overline{T} = (\overline{V}, \overline{Ed}, \overline{\lambda})$. We now define a mapping $\overline{h} : \overline{V} \to V'$, as follows. For $\overline{p} \in \overline{V}$,

$$\overline{h}(\overline{p}) = \begin{cases} h_1(i_1^{-1}(\overline{p})) & \text{if } \overline{p} \in i_1(V_1), \\ h_2(i_2^{-1}(\overline{p})) & \text{if } \overline{p} \in i_2(V_2), \end{cases}$$

with $i_1 : V_1 \to \overline{V}$ and $i_2 : V_2 \to \overline{V}$ the canonical embeddings from T_1, respectively, T_2 into $\overline{T}$. First, notice, that $i_1(V_1) \cup i_2(V_2) = \overline{V}$. Now, if $\overline{p} \in i_1(V_1) \cap i_2(V_2)$, then, for some k, $0 \leq k \leq d$, $\overline{p} = \overline{m}^k$, $i_1^{-1}(\overline{p}) = m_1^k$ and $i_2^{-1}(\overline{p}) = m_2^k$. Since we have established that $h_1(m_1^k) = h_2(m_2^k)$, $\overline{h}$ is well defined.

[4] In this paper, a *subtree* of a tree is a connected subgraph of that tree, which is weaker than the standard definition. To make the distinction clear, we shall refer to a subtree containing all descendants of some node of the tree as a *complete subtree*.

It remains to show that $\overline{h}$ is a containment mapping. Since i_1 and i_2 map T_1, respectively, T_2, to subtrees of $\overline{T}$, and both subtrees together cover $\overline{T}$, every edge of $\overline{T}$ is in at least one of these subtrees. So, let $\overline{p}, \overline{q} \in \overline{V}$ such that $(\overline{p}, \overline{q}) \in \overline{Ed}$. If this edge is, e.g. in the embedding of T_1 into $\overline{T}$, then $(i_1^{-1}(\overline{p}), i_1^{-1}(\overline{q})) \in Ed_1$. Hence, $(\overline{h}(\overline{p}), \overline{h}(\overline{q})) = (h_1(i_1^{-1}(\overline{p})), h_1(i_1^{-1}(\overline{q}))) \in Ed'$, since h_1 is a containment mapping of T_1 into T'. Finally, consider $\overline{p} \in \overline{V}$. If $\overline{p} \in i_1(V_1) - i_2(V_2)$, it follows that $\overline{\lambda}(\overline{p}) = \lambda_1(i_1^{-1}(\overline{p}))$. Since h_1 is a containment mapping of T_1 into T', $\overline{\lambda}(\overline{p}) = \lambda_1(i_1^{-1}(\overline{p})) \geq \lambda'(h_1(i_1^{-1}(\overline{p}))) = \lambda'(\overline{h}(\overline{p}))$. The case where $\overline{p} \in i_2(V_2) - i_1(V_1)$ is completely analogous, of course. So, consider the case where $\overline{p} \in i_1(V_1) \cap i_2(V_2)$. By construction, $\overline{\lambda}(\overline{p}) = \min(\lambda_1(i_1^{-1}(\overline{p})), \lambda_2(i_2^{-1}(\overline{p})))$. Without loss of generality, assume that the minimum equals the first component. As before, we then reach the desired conclusion.

(3) Suppose that the result of **Merge1** is non-empty and, therefore, a labeled tree, say $\overline{T} = (\overline{V}, \overline{Ed}, \overline{\lambda})$, and that there exists a containment mapping $\overline{h} : \overline{V} \to V'$ of $\overline{T}$ into T'. For $j = 1, 2$, define $h_j : V_j \to V' : p \mapsto \overline{h}(i_j(p))$, with $i_j : V_j \to \overline{V}$ the canonical embedding of T_j into $\overline{T}$. Notice that i_j is a containment mapping, by construction. Hence, h_j, as a composition of containment mappings is again a containment mapping, of T_j into T'. Since $i_1(m_1) = i_2(m_2) = \overline{m}^0$, $h_1(m_1) = h_2(m_2)$. □

Suppose **Merge2**(T, m_1, m_2) in Fig. 7 is a labeled tree $\overline{T}$. In Example 4.2, we already illustrated informally that there is a *canonical covering*, say f, by T of $\overline{T}$. In general, this canonical covering is defined as follows, using the notation in Fig. 7. For m in V,

$$\begin{cases} f(m) = m & \text{if } m \text{ is not an ancestor of } m_1 \text{ or } m_2, \\ f(m_j^k) = \overline{m}^k & \text{for } j = 1, 2, \text{ and } k = 0, \ldots, d. \end{cases}$$

From a straightforward argument, it follows that the image of T under f is precisely $\overline{T}$, justifying the term 'covering'. Finally, we are now ready to prove the analog of Lemma 4.2 for **Merge2**.

Lemma 4.3. *Let $T = (V, Ed, \lambda)$ and $T' = (V', Ed', \lambda')$ be labeled trees, and let $m_1, m_2 \in V$ be nodes.*

(1) ***Merge2**(T, m_1, m_2) is either the empty set or a labeled tree.*
(2) *If there exists a containment mapping $h : V \to V'$ of T into T' such that $h(m_1) = h(m_2)$, then **Merge2**(T, m_1, m_2) is a labeled tree, say $\overline{T} = (\overline{V}, \overline{Ed}, \overline{\lambda})$, and there exists a containment mapping $\overline{h} : \overline{V} \to V'$ of $\overline{T}$ into T' such that, for each node $p \in V$, $\overline{h}(f(p)) = h(p)$, with $f : V \to \overline{V}$ the canonical covering by T of $\overline{T}$.*
(3) *If **Merge2**(T, m_1, m_2) is a labeled tree, say $\overline{T} = (\overline{V}, \overline{Ed}, \overline{\lambda})$, and there exists a containment mapping $\overline{h} : \overline{V} \to V'$ of $\overline{T}$ into T, then the mapping $h : V \to V' : p \mapsto \overline{h}(f(p))$, with $f : V \to \overline{V}$ the canonical covering by T of $\overline{T}$, is a containment mapping of T into T' such that $h(m_1) = h(m_2)$.*

Proof. (1) Suppose it has already been established that the distance between m_1 and $n = \text{top}(m_1, m_2)$ equals the distance between m_2 and n, say d, and that, for $k = 0, \ldots, d-1$, the level-k ancestors of m_1 and m_2 are compatible, i.e. that **Merge2**(T, m_1, m_2) is not the empty set. Then, **Merge2** will output a graph $\overline{T}$. Initially, $\overline{T} = T$, a tree, in which every node has indegree 1, except for the root, which has indegree 0. If $d = 0$, $\overline{T} = T$. If $d = 1$, then it is immediate that, after merging m_1 and m_2, the indegree of the new node remains 1, whence $\overline{T}$ remains a tree. If $d > 1$, let $\overline{T}^k$ be the value of T after the kth iteration of the for-loop. If $0 \leq k < d-1$, it follows from a straightforward induction that $\overline{T}^k$ is a connected graph in which all nodes have indegree 1, except for the root of T, which has indegree 0, and $\overline{m}^k$, which has indegree 2. Finally, merging m_1^{d-1} and m_2^{d-1} into $\overline{m}^{d-1}$ will decrease the indegree of m^{d-2} by one, whereas the indegree of $\overline{m}^{d-1}$ will be 1 (as in the case $d = 1$). Hence, also in this case, $\overline{T}^{d-1}$, the final result of **Merge2**, is connected, and all its nodes have indegree 1, except for one node, which has indegree 0. Such a graph is necessarily a tree.

(2) Suppose there exists a containment mapping $h : V \to V'$ of T into T' such that $h(m_1) = h(m_2)$. Let $\overline{d}$ be the minimum of the depth of m_1 and the depth of m_2 in T. For $k = 0, \ldots, \overline{d}$, let m_1^k and m_2^k be the level-k ancestors of m_1 and m_2, respectively. By Lemma 4.1, it follows that, for $k = 0, \ldots, \overline{d}$, $h(m_1^k) = h(m_2^k)$. Since containment mappings preserve distance along directed paths in a tree, this is only possible if m_1 and m_2 have the same distance to $n = \text{top}(m_1, m_2)$. Additionally, m_1^k and m_2^k must be compatible. We may thus conclude that the result of **Merge2** is non-empty, and, therefore, by Lemma 4.3(1), a labeled tree, say $\overline{T} = (\overline{V}, \overline{Ed}, \overline{\lambda})$. We now define a mapping $\overline{h} : \overline{V} \to V$, as follows. For $\overline{p} \in \overline{V}$, let $p \in V$ be such that $\overline{p} = f(p)$. (Such a node always exists, since f is surjective.) Then, $\overline{h}(\overline{p}) = h(p)$. Now assume $\overline{p} = f(p_1) = f(p_2)$. Then, clearly, either $p_1 = p_2$, or, for some k, $0 \leq k < d$, $p_1 = m_1^k$ and $p_2 = m_2^k$, or vice-versa. In both cases, $h(p_1) = h(p_2)$, whence $\overline{h}$ is well defined. It remains to show that $\overline{h}$ is a containment mapping. Thus, let $\overline{p}, \overline{q} \in \overline{V}$ such that $(\overline{p}, \overline{q}) \in \overline{Ed}$. By construction, there exist $p, q \in V$ such that $f(p) = \overline{p}$, $f(q) = \overline{q}$ and $(p, q) \in Ed$. Hence, $(\overline{h}(\overline{p}), \overline{h}(\overline{q})) = (h(p), h(q)) \in Ed'$, since h is a containment mapping of T into T'.

Finally, consider $\overline{p} \in \overline{V}$. If $\overline{p} \notin \{\overline{m}^0, \ldots, \overline{m}^{d-1}\}$, let $p \in V$ the unique node such that $f(p) = \overline{p}$. Since h is a containment mapping of T into T', $\overline{\lambda}(\overline{p}) = \lambda(p) \geq \lambda'(h(p)) = \lambda'(\overline{h}(\overline{p}))$. If, for some k, $0 \leq k < d$, $\overline{p} = \overline{m}^k$, then, by construction, $\overline{\lambda}(\overline{p}) = \min(\lambda(m_1^k), \lambda(m_2^k))$. Without loss of generality, assume that the minimum equals the first component. As before, we then reach the desired conclusion.

(3) For the other direction, suppose that the result of **Merge2** is non-empty and, therefore, a labeled tree, say $\overline{T} = (\overline{V}, \overline{Ed}, \overline{\lambda})$,

```
Algorithm Path-T
Input:  a Path$^+$ expression $E$;
Output:  a tree query $P = (T, s, d)$ or $\emptyset$.
Method:
    if $E = \emptyset$ return $\emptyset$;
    if $E$ is another primitive of Path$^+$
        return the appropriate tree query of Figure 10;
    if $E = E_1; E_2$
        if Path-T$(E_1) = \emptyset$ return $\emptyset$;
        if Path-T$(E_2) = \emptyset$ return $\emptyset$;
        let $P_1 = (T_1, s_1, d_1) =$ Path-T$(E_1)$;
        let $P_2 = (T_2, s_2, d_2) =$ Path-T$(E_2)$;
        if Merge1$(T_1, T_2, d_1, s_2) = \emptyset$ return $\emptyset$;
        let $\overline{T} =$ Merge1$(T_1, T_2, d_1, s_2)$;
        let $i_1$ be the canonical embedding of $T_1$ in $\overline{T}$;
        let $i_2$ be the canonical embedding of $T_2$ in $\overline{T}$;
        return $\overline{P} = (\overline{T}, i_1(s_1), i_2(d_2))$;
    if $E = \Pi_1(E')$
        if Path-T$(E') = \emptyset$ return $\emptyset$;
        let $P = (T, s, d) =$ Path-T$(E')$;
        return $P = (T, s, s)$;
    if $E = \Pi_2(E')$
        if Path-T$(E') = \emptyset$ return $\emptyset$;
        let $P = (T, s, d) =$ Path-T$(E')$;
        return $P = (T, d, d)$;
    if $E = E_1 \cap E_2$
        if Path-T$(E_1) = \emptyset$ return $\emptyset$;
        if Path-T$(E_2) = \emptyset$ return $\emptyset$;
        let $P_1 = (T_1, s_1, d_1) =$ Path-T$(E_1)$;
        let $P_2 = (T_2, s_2, d_2) =$ Path-T$(E_2)$;
        if Merge1$(T_1, T_2, s_1, s_2) = \emptyset$ return $\emptyset$;
        let $\overline{T} =$ Merge1$(T_1, T_2, s_1, s_2)$;
        let $i_1$ be the canonical embedding of $T_1$ in $\overline{T}$;
        let $i_2$ be the canonical embedding of $T_2$ in $\overline{T}$;
        if Merge2$(\overline{T}, i_1(d_1), i_2(d_2)) = \emptyset$ return $\emptyset$;
        let $\overline{\overline{T}} =$ Merge2$(\overline{T}, i_1(d_1), i_2(d_2))$;
        let $f$ be the canonical covering by $\overline{T}$ of $\overline{\overline{T}}$;
        return $\overline{\overline{P}} = (\overline{\overline{T}}, f(i_1(s_1)), f(i_1(d_1)))$
                        $= (\overline{\overline{T}}, f(i_2(s_2)), f(i_2(d_2)))$;
    if $E = E'^{-1}$
        if Path-T$(E') = \emptyset$ return $\emptyset$;
        let $P = (T, s, d) =$ Path-T$(E')$;
        return $P = (T, d, s)$.
```

FIGURE 9. Algorithm **Path-T**.

and that there exists a containment mapping $\overline{h} : \overline{V} \to V'$ of $\overline{T}$ into T'. Define $h : V \to V' : p \mapsto \overline{h}(s(p))$, with $s : V \to \overline{V}$ the canonical covering by T of $\overline{T}$. Notice that s is a containment mapping, by construction. Hence, h, as a composition of containment mappings is again a containment mapping of T into T'. Since $s(m_1) = s(m_2) = \overline{m}^0$, $h(m_1) = h(m_2)$. □

We are now ready to translate Path$^+$ into tree queries (or the empty set). The actual algorithm is exhibited in Fig. 9. Notice that, by Proposition 4.1, it would have sufficed to exhibit the translations of the primitives, and the composition and intersection operations. However, we preferred to consider *all* Path$^+$ operations in order to provide a direct translation algorithm from Path$^+$ expressions into expressions of **T**, thus avoiding a costly translation from Path$^+$ to Path$^+(\cap)$ as a preprocessing step.

PROPOSITION 4.2. *Algorithm **Path-T** correctly translates an arbitrary Path$^+$ expression into an equivalent expression of **T** (i.e. a tree query or $\emptyset$).*

Proof. It is readily seen that Algorithm **Path-T** correctly translates $\emptyset$ and the other primitives of Path$^+$ into equivalent expressions of **T** (Fig. 10). Also, the cases for the projection and inverse operations are straightforward. We therefore focus on composition and intersection.

Composition. Let E_1 and E_2 be Path$^+$ expressions for which P_1 and P_2 are the equivalent expressions in **T**. If one of P_1 or P_2 equals $\emptyset$, then, obviously, $E_1; E_2$ must be translated into $\emptyset$. Otherwise, let $P_1 = (T_1, s_1, d_1)$ and $P_2 = (T_2, s_2, d_2)$ be the corresponding tree queries. Now, consider **Merge1**(T_1, T_2, d_1, s_2). If the result is $\emptyset$, so is the translation of $E_1; E_2$, since, in this case, $E_1(D)$ and $E_2(D)$ can never contain matching node pairs, whatever the document D under consideration. If the result of the algorithm is a labeled tree, say $\overline{T}$, then Lemma 4.2 states precisely that the tree query $\overline{P} = (\overline{T}, i_1(s_1), i_2(d_2))$ is the correct translation of $E_1; E_2$, where i_1 and i_2 are the canonical embeddings of T_1, respectively, T_2 into $\overline{T}$.

Intersection. Let E_1 and E_2 be Path$^+$ expressions for which P_1 and P_2 are the equivalent expressions in **T**. If one of P_1 or P_2 equals $\emptyset$, then, obviously, $E_1 \cap E_2$ must be translated into $\emptyset$. Otherwise, let $P_1 = (T_1, s_1, d_1)$ and $P_2 = (T_2, s_2, d_2)$ be the corresponding tree queries. Now, consider **Merge1**(T_1, T_2, s_1, s_2). If the result is $\emptyset$, so is the translation of $E_1 \cap E_2$, since, in this case, $E_1(D)$ and $E_2(D)$ can never contain node pairs that match on their first components—let alone share node pairs—whatever the document D under consideration. Thus, let the result of the algorithm be a labeled tree, say $\overline{T}$. Given an arbitrary document $D = (V', Ed', \lambda')$, we have that $E_1 \cap E_2(D) = E_1(D) \cap E_2(D)$ equals

$$\begin{aligned}
&\{(p, q) \mid (p, q) \in E_1(D) \ \&\ (p, q) \in E_2(D)\} \\
&\quad = \{(h_1(s_1), h_1(d_1)) \mid h_1 \text{ is a c.m.}^{5} \text{ of } T_1 \text{ into } D \\
&\qquad\quad \&\text{ there exists a c.m. } h_2 \text{ of } T_2 \text{ into } D \\
&\qquad\quad \&\ h_1(s_1) = h_2(s_2)\ \&\ h_1(d_1) = h_2(d_2)\} \\
&\quad = \{(\overline{h}(i_1(s_1)), \overline{h}(i_1(d_1))) \mid \overline{h} \text{ is a c.m. of } \overline{T} \\
&\qquad\quad \text{into } D\ \&\ \overline{h}(i_1(d_1)) = \overline{h}(i_2(d_2))\},
\end{aligned}$$

where i_1 and i_2 are the canonical embeddings of T_1, respectively, T_2, in $\overline{T}$. In the first step, we used that P_1 and P_2 are correct translations of E_1, respectively, E_2, and in the second step, we relied on Lemma 4.2. Next, consider **Merge2**$(\overline{T}, i_1(d_1), i_2(d_2))$. If the result is $\emptyset$, so is the translation of $E_1 \cap E_2$, since, in this case, the above set is always empty, as $i_1(d_1)$ and $i_2(d_2)$ can never be mapped onto the same node of D, whatever the document D under consideration. If the result of the algorithm

[5]Containment mapping.

is a labeled tree, say $\overline{\overline{T}}$, it follows from Lemma 4.3 that

$$\{(\overline{h}(i_1(s_1)), \overline{h}(i_1(d_1))) \mid \overline{h} \text{ is a c.m. of } \overline{T} \text{ into } D$$
$$\& \ \overline{h}(i_1(d_1)) = \overline{h}(i_2(d_2))\} =$$
$$\{(\overline{\overline{h}}(f(i_1(s_1))), \overline{\overline{h}}(f(i_1(d_1)))) \mid$$
$$\overline{\overline{h}} \text{ is a c.m. of } \overline{\overline{T}} \text{ into } D\},$$

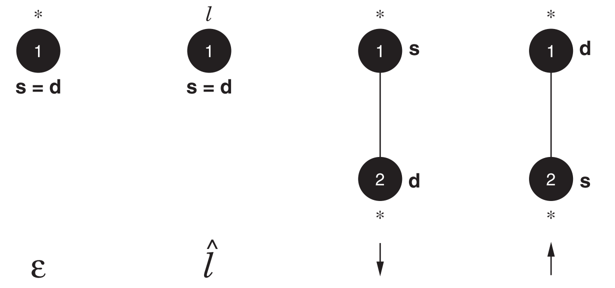
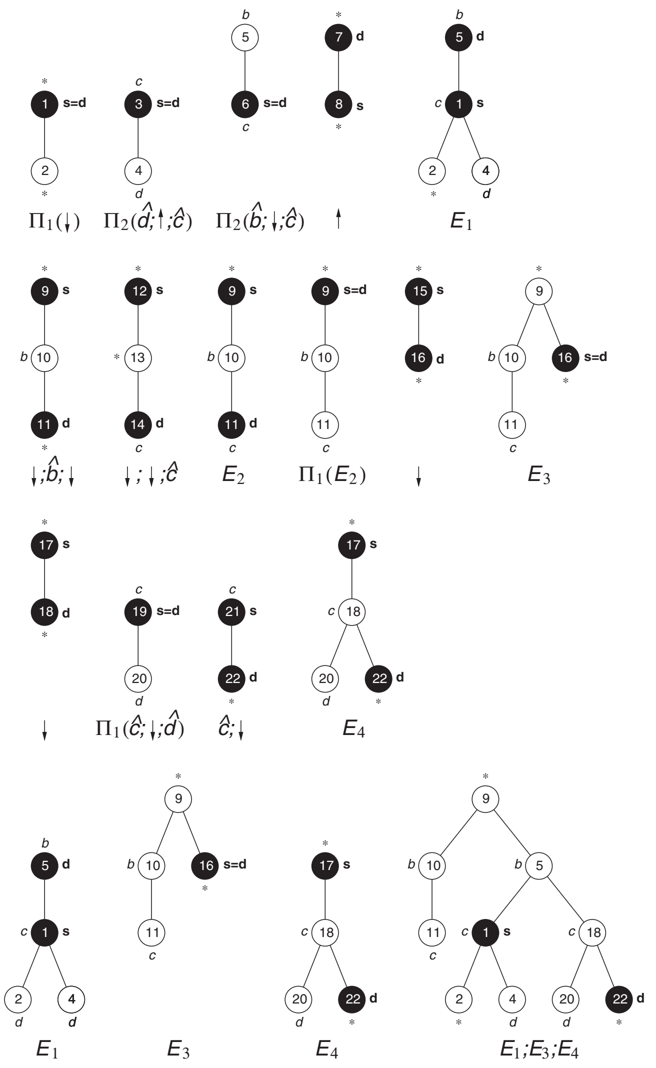


FIGURE 10. Translation of the primitives of Path$^+$ into tree queries.



$\Pi_1(\downarrow)$
$\Pi_2(\hat{d};\uparrow;\hat{c})$
$\Pi_2(\hat{b};\downarrow;\hat{c})$
↑
E_1
$\downarrow;\hat{b};\downarrow$
$\downarrow;\downarrow;\hat{c}$
E_2
$\Pi_1(E_2)$
↓
E_3
↓
$\Pi_1(\hat{c};\downarrow;\hat{d})$
$\hat{c};\downarrow$
E_4
E_1
E_3
E_4
$E_1;E_3;E_4$

FIGURE 11. Translation of the Path$^+$ expression considered in Example 4.3.

```
Algorithm T-Path
Input: a tree query $P = (T, s, d)$ or $\emptyset$;
Output: a Path$^+(\Pi_1, \Pi_2)$ expression $E$.
Method:
  if the input expression is $\emptyset$ % Case 0
    $E := \emptyset$;    let $r$ be the root of $T$;
  if $s = d$
    if $s \neq r$ % Case 1
      let $T_1$ be the complete subtree of $T$ rooted at $s$;
      let $T_2$ be $T$ with all strict descendants
            of $s$ removed;
      In $T_2$, replace $s$ by $s'$ with $\lambda(s') = *$;
      $E := \Pi_2(\textbf{T-Path}(T_2, r, s')); \textbf{T-Path}(T_1, s, s)$
    else % $s = r$
      if $T$ consists of a single node
        if $\lambda(s) = *$ % Case 2
          $E := \varepsilon$
        else % $\lambda(s) \in \mathcal{L}$, Case 3
          $E := \widehat{\lambda(s)}$
      else % T does not consist of a single node, Case 4
        let $s_1$ be a child of $s$;
        let $T' = T$;
        In $T'$, replace $s$ by $s'$ with $\lambda(s') = *$;
        let $T_s$ be the labeled tree consisting
              of the single labeled node $s$;
        $E := \Pi_1(\textbf{T-Path}(T', s', s_1)); \textbf{T-Path}(T_s, s, s)$
  else % $s \neq d$
    if $s$ is a strict ancestor of $d$ % Case 5
      let $s_1$ be the child of $s$ that is also an ancestor of $d$;
      let $T_1$ be the complete subtree rooted at $s_1$;
      let $T_2$ be $T$ with $s_1$ and all descendants
      of $s_1$ removed;
      $E := \textbf{T-Path}(T_2, s, s); \downarrow; \textbf{T-Path}(T_1, s_1, d)$;
    else % s is not an ancestor of d, Case 6
      let $s_1$ be the parent of $s$;
      let $T_1$ be the complete subtree rooted at $s$;
      let $T_2$ be $T$ with $s$ and all descendants
            of $s$ removed;
      $E := \textbf{T-Path}(T_1, s, s); \uparrow; \textbf{T-Path}(T_2, s_1, d)$;
  In $E$, remove all occurrences of "$;\varepsilon$" and "$\varepsilon;$";
  return $E$.
```

FIGURE 12. Algorithm **T-Path**.

where f is the canonical covering by $\overline{T}$ of $\overline{\overline{T}}$. Hence, the translation of $E_1 \cap E_2$ proposed in Algorithm **Path-T** is correct. □

Example 4.3. Consider again the Path$^+$ expression given in Example 3.2:

$$\Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{b}; \downarrow; \hat{c}); \uparrow;$$
$$\Pi_2(\Pi_1((\downarrow; \hat{b}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow); \downarrow; \Pi_1(\hat{c}; \downarrow; \hat{d}); \hat{c}; \downarrow.$$

We will now translate this expression into a tree query. This translation is illustrated in Fig. 11. Merged nodes are identified with the lowest of the numbers involved in the merging and intermediate results are labeled with the subexpressions to which they correspond. In particular,

$$E_1 = \Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{b}; \downarrow; \hat{c}); \uparrow,$$
$$E_3 = \Pi_2(\Pi_1(E_2); \downarrow),$$

with

$$E_2 = (\downarrow; \hat{b}; \downarrow) \cap (\downarrow; \downarrow; \hat{c}),$$
$$E_4 = \downarrow; \Pi_1(\hat{c}; \downarrow; \hat{d}); \hat{c}; \downarrow.$$

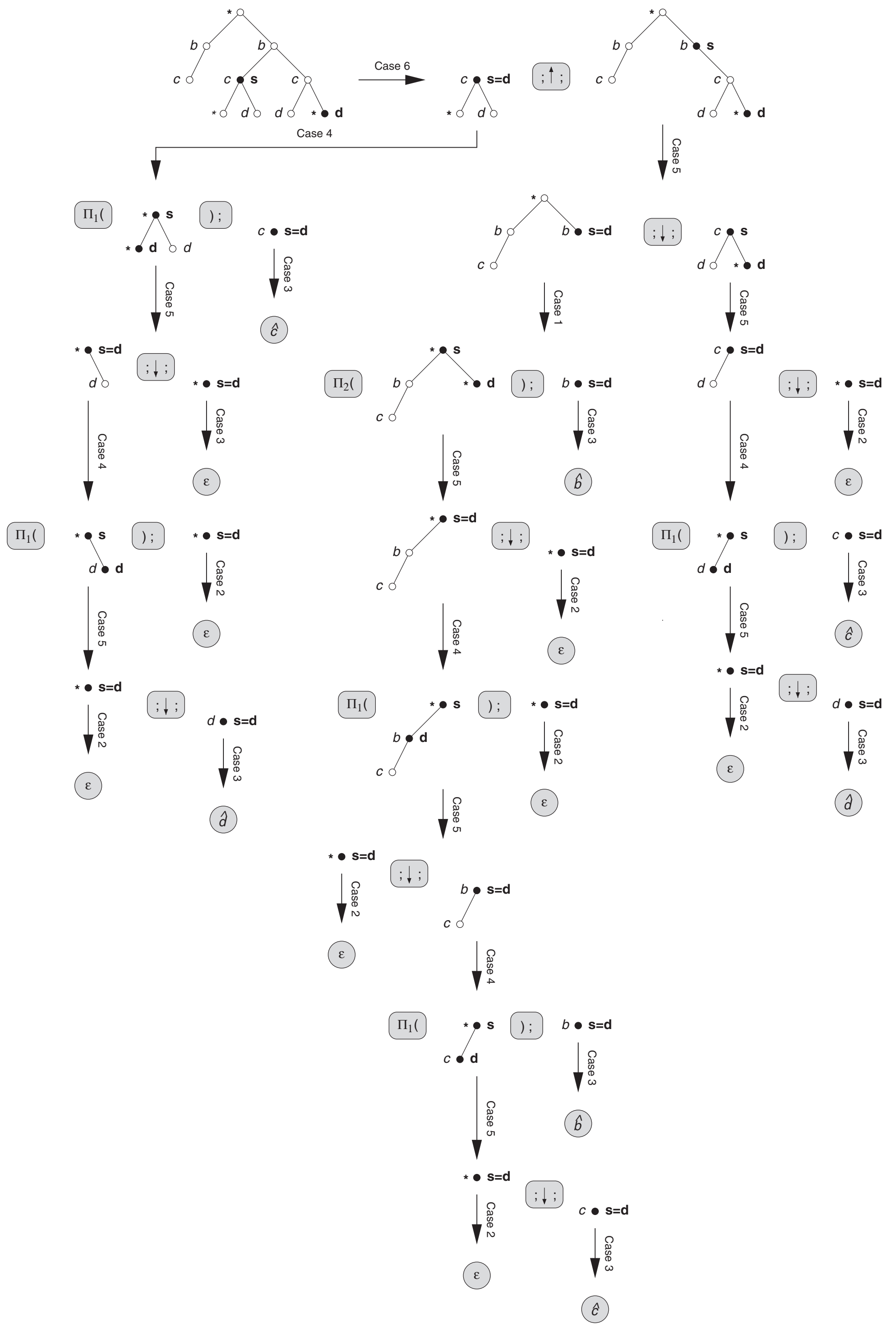


FIGURE 13. Schematic illustration of the translation of the rightmost tree query in Fig. 11 into an equivalent Path$^+(\Pi_1, \Pi_2)$ expression by Algorithm **T-Path** in Fig. 12.

Notice that the Path$^+$ expression under consideration equals the composition E_1; E_3; E_4. Its translation is exhibited as the last tree in Fig. 11.

Now that we have shown that a Path$^+$ expression can be translated into an expression of **T**, i.e. ∅ or tree query, and also exhibited an algorithm to perform this translation, we are next going to show that it is also possible to translate every expression of **T** into a Path$^+$ expression in an algorithmic fashion. Moreover, the translation algorithm, exhibited in Fig. 12, always returns expressions in Path$^+(\Pi_1, \Pi_2)$. We will analyze these expressions further in Section 5, where we shall propose a normal form for Path$^+$ expressions.

PROPOSITION 4.3. *Algorithm **T-Path** correctly translates an arbitrary **T** expression (i.e. a tree query or* $\emptyset$*) into an equivalent expression of Path*$^+(\Pi_1, \Pi_2)$.

Proof. Clearly, the **T** expression $\emptyset$ is correctly translated into the Path$^+$ expression $\emptyset$. Therefore, we may assume for the remainder of this proof that the input expression is a tree query $P = (T, s, d)$.

First, we prove that Algorithm **T-Path** always terminates. Cases 2 and 3 are base cases in which no recursive call occurs. In Cases 5 and 6, the labeled trees T_1 and T_2 both contain fewer nodes than T. This is also true in Case 1, except when s is a leaf, in which case T_2 and T have the same number of nodes.[6] However, the recursive call **T-Path**(T_2, r, s') is dealt with in Case 5, where the resulting labeled trees have strictly fewer nodes. Finally, in Case 4, the recursive call involves (a variation of) T. However, **T-Path**(T', s', s_1) is again dealt with in Case 5, as above. We may therefore conclude that the recursion must stop.

The correctness of Algorithm **T-Path** follows from a structural inductive argument. For the two base cases, Cases 2 and 3, this is obvious. For the other cases, the proposed translation of the tree query $P = (T, s, d)$ under consideration consists of a composition of one up to three subexpressions. Assume in each of these cases that, for each tree query $P' = (T', s', d')$ in one of these subexpressions, **T-Path**(T', s', d') is a correct translation of P'. If the subexpression is of the form Π_1(**T-Path**(T', s', d')) (Case 4), it follows from Proposition 4.2 that this subexpression is a correct translation of $P'' = (T', s', s')$. If the subexpression is of the form Π_2(**T-Path**(T', s', d')) (Case 1), it follows from Proposition 4.2 that this subexpression is a correct translation of $P'' = (T', d', d')$. Finally, if the subexpression is $\downarrow$ (Case 5) or $\uparrow$ (Case 6), it follows from Proposition 4.2 that this subexpression is a correct translation of the corresponding tree query in Fig. 10. All other subexpressions are of the form **T-Path**(T', s', d'). Thus, we have established that, in each of the Cases 1, 4, 5 and 6, the proposed translation of P is a composition of one up to three subexpressions, each of which is the translation of a particular tree query. By applying Algorithm **Path-T** (Fig. 9) to these subexpressions, we obtain the tree query P back in each of these cases, whence the proposed translation of P is correct. □

EXAMPLE 4.4. Consider the tree query obtained for

$$\Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{b}; \downarrow; \hat{c}); \uparrow;$$
$$\Pi_2(\Pi_1((\downarrow; \hat{b}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow); \downarrow; \Pi_1(\hat{c}; \downarrow; \hat{d}); \hat{c}; \downarrow$$

in Example 4.3, i.e. the final tree query in Fig. 11. If we translate this tree query into an equivalent Path$^+(\Pi_1, \Pi_2)$ expression using Algorithm **T-Path** in Fig. 12, we obtain

$$\Pi_1(\Pi_1(\downarrow); \downarrow; \hat{d}); \uparrow; \Pi_2(\Pi_1(\downarrow; \Pi_1(\downarrow; \hat{c}); \hat{b}); \downarrow); \hat{b}; \downarrow;$$
$$\Pi_1(\downarrow; \hat{d}); \hat{c}; \downarrow.$$

In Fig. 13, we exhibit the detailed steps of this translation.

We can now summarize Propositions 4.1–4.3.

THEOREM 4.1. *The query languages Path*$^+$*, Path*$^+(\cap)$*, Path*$^+(\Pi_1, \Pi_2)$ *and **T** are all equivalent in expressive power, and there exist translation algorithms between any two of them.*

The diagram in Fig. 14 illustrates graphically how the equivalences in Theorem 4.1 have been established.

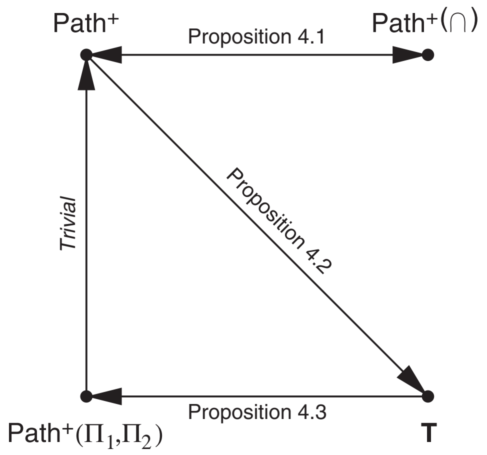


FIGURE 14. Schematic diagram illustrating how the equivalence of Path$^+$, Path$^+(\cap)$, Path$^+(\Pi_1, \Pi_2)$ and **T** in Theorem 4.1 has been established.

5. NORMAL FORM FOR EXPRESSIONS IN THE PATH$^+$ ALGEBRA

Normalization is frequently a critical step in rule-based query optimization. It serves the purpose of unifying queries with the same semantics, detecting containment among subqueries and establishing the foundation for cost-based query optimization, in which evaluation plans are to be generated. As it will turn out, using this normal form, we can decompose a Path$^+$ query into subqueries that can be expressed in DPath$^+(\Pi_1)$, a very small fragment of Path$^+$ for which efficient evaluation strategies are available [18]. The full query can then be evaluated by joining the results of these DPath$^+(\Pi_1)$ expressions.

When we revisit Section 4, where the translation algorithm **T-Path** from expressions in **T** to expressions in Path$^+(\Pi_1, \Pi_2)$ is described, notably in Fig. 12, we observe that the result of the translation is an expression E which conforms to the

[6]Actually, T_2 equals T in which the source node is relabeled by a wildcard.

following grammar:

$$\begin{aligned}E &= \emptyset \mid E_{\text{up}}\, E_{\text{top}}\, E_{\text{down}},\\ E_{\text{up}} &= (\Pi_1(F)^{?}\hat{\ell}^{?}\uparrow)^*,\\ E_{\text{top}} &= \Pi_2(F)^{?},\\ E_{\text{down}} &= F,\\ F &= \hat{\ell}^{?}(\downarrow\hat{\ell}^{?})^* \mid \Pi_1(F)^{?}\hat{\ell}^{?}(\downarrow\Pi_1(F)^{?}\hat{\ell}^{?})^*.\end{aligned}$$

For clarity, composition signs have been omitted in the regular expressions above. Also, notice that some subexpressions may equal ε and may therefore be omitted. Finally, notice that the base case for the recursive definition of F, i.e. $\hat{\ell}^{?}(\downarrow\hat{\ell}^{?})^*$, is actually subsumed by the general case, i.e. $\Pi_1(F)^{?}\hat{\ell}^{?}(\downarrow\Pi_1(F)^{?}\hat{\ell}^{?})^*$.

Obviously, the only non-primitive operations that occur in a formula that conforms to the above grammar are composition and the projections. Moreover, the subexpressions which conform to F are all in DPath$^+(\Pi_1)$. Hence, the second projection occurs at most once, and, if so, at the outer level. Also, the $\uparrow$ primitives occur only at the outer level. In addition, all $\uparrow$ primitives occur to the left of the Π_2 operation, if present, and of all $\downarrow$ primitives. We say that a Path$^+$ expression which conforms to the above grammar is in *normal form*.

EXAMPLE 5.1. Consider again Example 3.2. Clearly, the Path$^+$ expression

$$\Pi_1(\downarrow);\, \Pi_2(\hat{d};\, \uparrow;\, \hat{c});\, \Pi_2(\hat{b};\, \downarrow;\, \hat{c});\, \uparrow;$$
$$\Pi_2(\Pi_1((\downarrow;\, \hat{b};\, \downarrow) \cap (\downarrow;\, \downarrow;\, \hat{c}));\, \downarrow);\, \downarrow;\, \Pi_1(\hat{c};\, \downarrow;\, \hat{d});\, \hat{c};\, \downarrow$$

is not in normal form. In particular, notice that this expression contains an intersection and multiple occurrences of the second projection. In Example 4.3, we exhibited how this expression is translated into the final tree query shown in Fig. 11. In Example 4.4, we exhibited how this tree query is translated into the Path$^+(\Pi_1, \Pi_2)$ expression

$$\Pi_1(\Pi_1(\downarrow);\, \downarrow;\, \hat{d});\, \uparrow;\, \Pi_2(\Pi_1(\downarrow;\, \Pi_1(\downarrow;\, \hat{c});\, \hat{b});\, \downarrow);\, \hat{b};\, \downarrow;$$
$$\Pi_1(\downarrow;\, \hat{d});\, \hat{c};\, \downarrow.$$

This expression can indeed be written as E_{up}; E_{top}; E_{down}, with $E_{\text{up}} = \Pi_1(F_1); \uparrow$, $E_{\text{top}} = \Pi_2(F_2)$ and $E_{\text{down}} = F_3$, where

$$\begin{aligned}&F_1 = \Pi_1(F_4);\, \downarrow;\, \hat{d}, && F_4 = \downarrow, && F_7 = \downarrow;\, \hat{c}.\\ &F_2 = \Pi_1(F_5);\, \downarrow, && F_5 = \downarrow;\, \Pi_1(F_7);\, \hat{b}, &&\\ &F_3 = \hat{b};\, \downarrow;\, \Pi_1(F_6);\, \hat{c};\, \downarrow, && F_6 = \downarrow;\, \hat{d}. &&\end{aligned}$$

Hence, the expression is in normal form. It can be seen as the normalization of the original Path$^+$ expression in Example 3.2.

We formalize the above in the following theorem and corollary.

THEOREM 5.1. *The translation of a **T** expression into an equivalent Path$^+(\Pi_1, \Pi_2)$ expression by Algorithm **T-Path** of Fig. 12 is always in normal form.*

Proof. The **T** expression $\emptyset$ is translated into the Path$^+(\Pi_1, \Pi_2)$ expression $\emptyset$, which is assumed to be in normal form. Therefore, we may assume that the **T** expression under consideration is a tree query $P = (T, s, d)$. We shall trace Algorithm **T-Path**, and, in doing so, obtain that the generated translation is in normal form. Thereto, let $s = s_0, \ldots, s_{k-1}, s_k = t = d_l, d_{l-1}, \ldots, d_0 = d$ be the path from s to d in T, with $t = \text{top}(s, d)$. Notice that, possibly, $k = 0$, or $l = 0$, or both.

If $k > 0$, we are in Case 6 of the translation algorithm. There, in the last factor of the composition, the source is the parent of the source of P. For this last factor, we will remain in Case 6, until t is reached. We will then have obtained the following partial translation of P:

$$\overbrace{\textbf{T-Path}(T_0, s_0, s_0);\, \uparrow;\, \ldots\, \textbf{T-Path}(T_{k-1}, s_{k-1}, s_{k-1});\, \uparrow;}^{E_{\text{up}}}$$
$$\textbf{T-Path}(T_t, t, d).$$

Here, T_0 is the complete subtree of T rooted at $s_0 = s$; for $i = 1, \ldots, k-1$, T_i is the complete subtree of T rooted at s_i from which s_{i-1} and all its descendants are removed. Finally, T_t is the tree T from which s_{k-1} and all its descendant are removed. If $k = 0$, then $s = t$, and the above expression is still applicable, provided we put $E_{\text{up}} = \varepsilon$.

If $l > 0$, we are now in Case 5 of the translation algorithm for the translation of $P_t = (T_t, t, d)$. There, in the last factor of the composition, the source is the child of the source of P_t on the path to d. For this last factor, we will remain in Case 5, until d is reached. We will then have obtained the following partial translation of P_t:

$$\textbf{T-Path}(T^t, t, t);\, \downarrow;$$
$$\underbrace{\textbf{T-Path}(T^{l-1}, d_{l-1}, d_{l-1});\, \ldots :\, \downarrow;\, \textbf{T-Path}(T^0, d_0, d_0)}_{E'}.$$

Here, T^0 is the complete subtree of T_t rooted at $d_0 = d$; for $j = 1, \ldots, l-1$, T^j is the complete subtree of T_t rooted at d_j from which d_{j-1} and all its descendants are removed. Finally, T^t is the tree T_t from which d_{l-1} and all its descendants are removed. If $l = 0$, then $d = t$, and the above expression is still applicable, provided we put $E' = \varepsilon$.

Two possibilities may now occur.

(1) *The node t is the root of T^t.* Then, $E_{\text{top}} = \varepsilon$ and E_{down} equals **T-Path**(T^t, t, t); $\downarrow$; E'.
(2) *The node t is not the root of T^t.* Then, for the translation of $P^t = (T^t, t, t)$, we are in Case 1, and the following partial translation of P^t will result:

$$\Pi_2(\textbf{T-Path}(T^t_2, r, t'));\, \textbf{T-Path}(T^t_1, t, t).$$

Here, t' equals t relabeled by a wildcard, T_1^t is the complete subtree of T^t rooted at t, and T_2^t is the tree T^t from which the strict descendants of t are removed and in which t is replaced by t'. Then, $E_{\text{top}} = \Pi_2(\textbf{T-Path}(T_2^t, r, t'))$ and E_{down} equals

$$\textbf{T-Path}(T_1^t, t, t); \downarrow; E'.$$

Observe that the further translation of $P_2^t = (T_2^t, r, t)$ is analogous to the translation of P if $k = 0$ and t is the root of T. It has therefore the same form as E_{down} above. This concludes our treatment of the second possibility.

From the above analysis, we may conclude that the translation of P is of the form $E_{\text{up}}; E_{\text{top}}; E_{\text{down}}$, in which, omitting composition signs for clarity, these subexpressions have the general form $E_{\text{up}} = (G\uparrow)^*$, $E_{\text{top}} = \Pi_2(F)^?$ and $E_{\text{down}} = F$, where F stands for a DPath$^+(\Pi_1)$ expression of the form $G(\downarrow G)^*$ and G stands for the translation of a pattern of the form $P_r = (T, r, r)$, with r the root of the labeled tree T. (Superfluous ε primitives may of course be omitted.)

Hence, we have reduced the translation of a general tree query to the translation of a tree query of the form $P_r = (T, r, r)$, with which we shall deal next. If T is a single-node tree, we are in one of the base cases, i.e. Case 2, if r is labeled by a wildcard (in which case the translation is ε), or Case 3, if r is labeled by a symbol $\ell \in \mathcal{L}$ (in which case the translation is $\hat{\ell}$). Otherwise, we are in Case 4, whence the partial translation of P_r is of the form

$$\Pi_1(\textbf{T-Path}(T', r', r_1)); \textbf{T-Path}(T_r, r, r),$$

where T_r is the single-node-labeled tree consisting of the node r, r_1 is a child of r in T, r' is r relabeled by a wildcard and T' is the tree T in which r is replaced by r'. For the further translation of $P_{r'} = (T', r', r_1)$, we are again in the case where the source is a strict ancestor of the destination, whence its translation is of the general form of F. For the translation of the second factor above, we are again in a base case. We may therefore conclude, again omitting composition signs, that G is of the general form $\Pi_1(F)^?\hat{\ell}^?$, as was to be shown. □

An analysis of the proof of Theorem 5.1 reveals that, in the translation of the tree query $P = (T, s, d)$, the factors E_{up}, E_{top} and E_{down} can actually be identified with subtrees of T. Figure 15 shows T in its most generic form. In this figure, $t = \text{top}(s, d)$, and r is the root of T. Furthermore, T_r is the subtree of T obtained by removing all strict descendants of t, T_s is the subtree of T rooted at t which contains precisely all strict descendants of t on the path from t to s, as well as all their descendants, and, finally, T_d is the subtree of T rooted at t which contains all strict descendants of t not in T_s. (Notice that some or all of T_r, T_s and T_d may be single-node labeled trees). Let T_r' be T_r in which t is replaced by a wildcard-labeled node t', and let T_s' be T_s in which t is replaced by t'. Then, $E_{\text{up}} = \textbf{T-Path}(T_s', s, t')$, $E_{\text{top}} = \Pi_2(\textbf{T-Path}(T_r', r, t'))$ and $E_{\text{down}} = \textbf{T-Path}(T_d, t, d)$.

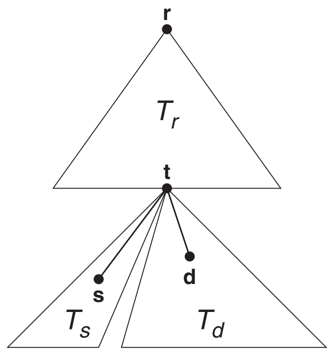


FIGURE 15. General structure of a tree query.

Corollary 5.1. *There exists an algorithm to translate an arbitrary Path$^+$ expression into an equivalent one in normal form.*

Proof. The algorithm first translates the given Path$^+$ expression into an equivalent **T** expression using the algorithm **Path-T** described in Fig. 9. By Theorem 5.1, the subsequent application of the algorithm **T-Path** described in Fig. 12 yields a Path$^+$ expression in normal form, equivalent to the original one. □

6. RESOLUTION EXPRESSIVENESS

So far, we have viewed Path$^+$ as a query language in which an expression associates to every document a binary relation on its nodes representing all paths in the document defined by that expression. We have referred to this view as the *query expressiveness* of Path$^+$. Alternatively, we can study the language's ability to distinguish a pair of nodes or a pair of paths in a given document, which we will refer to as the *resolution expressiveness* of Path$^+$.

Given a Path$^+$ expression E, a document D and a node m of D, we denote by $E(D)(m)$ the set $\{n \mid (m, n) \in E(D)\}$. We say that two nodes m and n in a document D cannot be resolved by a Path$^+$ expression if, for every Path$^+$ expression E, $E(D)(m)$ and $E(D)(n)$ are both empty or both nonempty. In this section, we first establish[7] that two nodes in a document cannot be resolved by a Path$^+$ expression if and only if the paths from the root of that document to these nodes have equal length, and corresponding nodes on these paths are *perfectly bisimilar* (Definition 6.4). For example, we will find that nodes n_2 and n_3 in the document D in Fig. 3 are perfectly bisimilar, whereas n_2 and n_4 are not. Consequently, by our results, we cannot find a Path$^+$ expression for which $E(D)(n_2) \neq \emptyset$ and $E(D)(n_3) = \emptyset$, or vice-versa, whereas an analogous conclusion does not hold for n_2 and n_4. Indeed, $\downarrow^2(D)(n_2) = \emptyset$, whereas $\downarrow^2(D)(n_4) \neq \emptyset$.

[7]The proof has the same structure as the proofs of similar properties for other fragments of the XPath Algebra in [9].

Next, we extend this result to the resolving power of Path$^+$ on pairs of paths in a document. Given a document and a pair of nodes, we will characterize which other pairs of nodes are always present in the output of a Path$^+$ expression returning the given pair upon input the given document. For example, this characterization will entail that every Path$^+$ expression returning the pair (n_6, n_7) upon input the document in Fig. 3 will also return the pair (n_6, n_6). Finally, we use these results to show that certain path queries cannot be expressed in Path$^+$. The previous example, e.g. implies that the query that returns all pairs of non-identical siblings in a document cannot be expressed in Path$^+$.

We first make precise what we mean by two nodes that cannot be resolved by a Path$^+$ expression.

Definition 6.1. *Let m_1 and m_2 be nodes of a document D.*

(1) *m_1 is* expression-related *to m_2, denoted $m_1 \geq_{exp} m_2$, if, for each Path$^+$ expression E, $E(D)(m_1) \neq \emptyset$ implies $E(D)(m_2) \neq \emptyset$.*
(2) *m_1 and m_2 are* expression-equivalent*, denoted $m_1 \equiv_{exp} m_2$, if $m_1 \geq_{exp} m_2$ and $m_2 \geq_{exp} m_1$.*

In fragments of the XPath Algebra containing a negation operator (e.g. set difference), expression-relatedness implies expression-equivalence. Unfortunately, this is not the case for Path$^+$. Clearly, expression-equivalence is an equivalence relation, but expression-relatedness is not.

Example 6.1. Consider again the document D in Fig. 3. Clearly, $n_4 \not\geq_{\rm exp} n_3$, as $\downarrow^2(D)(n_4) \neq \emptyset$, whereas $\downarrow^2(D)(n_3) = \emptyset$. In particular, $n_3 \not\equiv_{\rm exp} n_4$. It will follow from the results of this section, however, that $n_3 \geq_{\rm exp} n_4$. It will also follow from these results that, e.g. $n_6 \equiv_{\rm exp} n_7$.

Clearly, we need an instrument to establish expression-relatedness and expression-equivalence. Thereto, we intend to show that the semantic notions of expression-relatedness and expression-equivalence coincide with the syntactic notions of perfect similarity and perfect bisimilarity, respectively. Before we can give the formal definitions of perfect similarity and perfect bisimilarity of nodes, we need to define similarity of nodes.

Definition 6.2. *Similarity on the nodes of a document D is the smallest relation satisfying the following property. Let m_1 and m_2 be nodes of D. Then m_1 is* similar *to m_2, denoted $m_1 \geq_{\downarrow} m_2$, if $\lambda(m_1) = \lambda(m_2)$ and either*

(1) *m_1 is a leaf, or*
(2) *m_1 is not a leaf and, for each child n_1 of m_1, there exists a child n_2 of m_2 such that $n_1 \geq_{\downarrow} n_2$.*

(Two nodes are called *bisimilar*, denoted by the symbol '$\equiv_{\downarrow}$', if they are similar in both directions.)

Alternatively, Definition 6.2 can be interpreted as a recursive definition on the height of the nodes involved.

We now bootstrap Definition 6.2 to perfect similarity of nodes.

Definition 6.3. *Perfect similarity on the nodes of a document D is the smallest relation satisfying the following property. Let m_1 and m_2 be nodes of D. Then m_1 is* perfectly similar *to m_2, denoted $m_1 \geq_{\updownarrow} m_2$, if $m_1 \geq_{\downarrow} m_2$ and either*

(1) *$m_1 = m_2$ is the root, or*
(2) *m_1 and m_2 are both not the root, and $p_1 \geq_{\updownarrow} p_2$, with p_1 the parent of m_1 and p_2 the parent of m_2.*

Alternatively, Definition 6.3 can be interpreted as a recursive definition on the depth of the nodes involved.

Finally, we can define perfect bisimilarity of nodes.

Definition 6.4. *Let m_1 and m_2 be nodes of a document D. Then m_1 and m_2 are* perfectly bisimilar*, denoted $m_1 \equiv_{\updownarrow} m_2$, if $m_1 \geq_{\updownarrow} m_2$ and $m_2 \geq_{\updownarrow} m_1$.*

Example 6.2. Consider again the document in Fig. 3.

Of course, two identical nodes trivially satisfy all the relationships defined above in the present section. We therefore focus on relationships between different nodes.

For perfect bisimilarity, we have $n_2 \equiv_{\updownarrow} n_3$, $n_5 \equiv_{\updownarrow} n_6 \equiv_{\updownarrow} n_7$ and $n_{11} \equiv_{\updownarrow} n_{12}$, and, for bisimilarity, we have $n_2 \equiv_{\downarrow} n_3 \equiv_{\downarrow} n_{10}$, $n_5 \equiv_{\downarrow} n_6 \equiv_{\downarrow} n_7 \equiv_{\downarrow} n_8 \equiv_{\downarrow} n_{13}$ and $n_{11} \equiv_{\downarrow} n_{12}$. (Perfect) bisimilarity implies (perfect) similarity. In addition, the following perfect similarity relationships hold: $n_2 \geq_{\updownarrow} n_4$; $n_3 \geq_{\updownarrow} n_4$; $n_5 \geq_{\updownarrow} n_8$; $n_6 \geq_{\updownarrow} n_8$; $n_7 \geq_{\updownarrow} n_8$; $n_5 \geq_{\updownarrow} n_9$; $n_6 \geq_{\updownarrow} n_9$; $n_7 \geq_{\updownarrow} n_9$ and $n_8 \geq_{\updownarrow} n_9$. All of these are also similarity relationships. Besides these, also the similarity relationships $n_{10} \geq_{\downarrow} n_4$ and $n_{13} \geq_{\downarrow} n_9$ hold.

We mention the following useful properties of perfect (bi)similarity.

Proposition 6.1. *Let m_1 and m_2 be nodes of a document D such that $m_1 \geq_{\updownarrow} m_2$ (respectively, $m_1 \equiv_{\updownarrow} m_2$).*

(1) *If m_1 has an ancestor n_1, then m_2 has an ancestor n_2 such that $(m_1, n_1) \equiv (m_2, n_2)$ and $n_1 \geq_{\updownarrow} n_2$ (respectively, $n_1 \equiv_{\updownarrow} n_2$).*
(2) *If m_1 has a descendant n_1, then m_2 has a descendant n_2 such that $(m_1, n_1) \equiv (m_2, n_2)$ and $n_1 \geq_{\updownarrow} n_2$ (respectively, $n_1 \equiv_{\updownarrow} n_2$).*

Proof. The case for perfect bisimilarity follows immediately from the case for perfect similarity, by Definition 6.4. We will therefore only consider the latter case. The proof goes by induction on the length of the path between m_1 and n_1. In the base case, we have $n_1 = m_1$, whence $n_2 = m_2$, and both statements follow trivially.

We next consider the induction step.

To show the first statement, let n_1 be a strict ancestor of m_1. Then n_1 is also an ancestor of p_1, the parent of m_1. By Definition 6.3, m_2 has a parent, say p_2, and $p_1 \geq_{\updownarrow} p_2$. The statement now follows from the induction hypothesis.

To show the second statement, let n_1 be a strict descendant of m_1. Then n_1 is also a descendant of some child of m_1, say p_1. By Definition 6.3, $m_1 \geq_{\updownarrow} m_2$ implies $m_1 \geq_{\downarrow} m_2$. Hence, by Definition 6.2, m_2 has a child p_2 such that $p_1 \geq_{\downarrow} p_2$. By Definition 6.3, $p_1 \geq_{\updownarrow} p_2$. The statement now follows from the induction hypothesis. □

For the purpose of abbreviation, we extend perfect similarity and perfect bisimilarity to paths, i.e. pairs of nodes. Thereto, we need the notions of *subsumption* ('$\geq$') and *congruence* ('$\equiv$') between pairs of nodes introduced in Section 3. Thus, let m_1, m_2, n_1 and n_2 be nodes of a document D. We say that $(m_1, n_1) \geq_{\updownarrow} (m_2, n_2)$ (respectively, $(m_1, n_1) \equiv_{\updownarrow} (m_2, n_2)$) whenever $(m_1, n_1) \geq (m_2, n_2)$ (respectively, $(m_1, n_1) \equiv (m_2, n_2)$), $m_1 \geq_{\updownarrow} m_2$ (respectively, $m_1 \equiv_{\updownarrow} m_2$) and $n_1 \geq_{\updownarrow} n_2$ (respectively, $n_1 \equiv_{\updownarrow} n_2$).

EXAMPLE 6.3. Consider again Example 6.2. There, we established that $n_5 \geq_{\updownarrow} n_8$ and $n_6 \geq_{\updownarrow} n_9$. In addition, $(n_5, n_6) \geq (n_8, n_9)$. None of these relationships can be reversed. Hence, $(n_5, n_6) \geq_{\updownarrow} (n_8, n_9)$, but not the other way around. Also, $n_5 \equiv_{\updownarrow} n_5$ and $n_6 \equiv_{\updownarrow} n_7$, and $(n_5, n_6) \equiv (n_5, n_7)$ yield that $(n_5, n_6) \equiv_{\updownarrow} (n_5, n_7)$.

The abbreviation we have introduced is justified by the following property.

PROPOSITION 6.2. *Let m_1, m_2, n_1 and n_2 be nodes of a document D such that $(m_1, n_1) \geq_{\updownarrow} (m_2, n_2)$ (respectively, $(m_1, n_1) \equiv_{\updownarrow} (m_2, n_2)$). Now, let p_1 be any node on the path between m_1 and n_1. Let p_2 be the unique node that is either on the path between m_2 and n_2 or an ancestor of top(m_2, n_2) such that $(m_1, p_1) \geq (m_2, p_2)$ and $(p_1, n_1) \geq (p_2, n_2)$*[8] *(respectively, $(m_1, p_1) \equiv_{\updownarrow} (m_2, p_2)$ and $(p_1, n_1) \equiv_{\updownarrow} (p_2, n_2)$*[9]*). Then $p_1 \geq_{\updownarrow} p_2$ (respectively, $p_1 \equiv_{\updownarrow} p_2$).*

Proof. Clearly, p_1 is either an ancestor of m_1 or an ancestor of n_1 (or both). First, consider the case where p_1 is an ancestor of m_1. Then, $(m_1, p_1) \geq (m_2, p_2)$ implies that $(m_1, p_1) \equiv (m_2, p_2)$. Since p_2 is obviously the only node with this property, it follows immediately from Proposition 6.1(1), that $p_1 \geq_{\updownarrow} p_2$. The case where p_1 is an ancestor of n_1 is completely analogous. □

EXAMPLE 6.4. Continuing with Example 6.3, Proposition 6.2 states that, for the perfect similarity $(n_5, n_6) \geq_{\updownarrow} (n_8, n_9)$, that $n_5 \geq_{\updownarrow} n_8$, $n_2 \geq_{\updownarrow} n_4$, $n_1 \geq_{\updownarrow} n_1$, $n_3 \geq_{\updownarrow} n_4$ and $n_6 \geq_{\updownarrow} n_9$, which is indeed the case.

If one path subsumes another (respectively, if two paths are congruent), then each node encountered when navigating along the first path is perfectly similar (respectively, perfectly bisimilar) to the corresponding node encountered when doing the same navigation between the two nodes of the second path.

[8]Actually, one of these two conditions suffices; the first if p_1 is an ancestor of m_1, and the second if p_1 is an ancestor of n_1.

[9]Actually, one of these two conditions suffices.

Using the abbreviation, we can restate the condition *such that $(m_1, n_1) \equiv (m_2, n_2)$ and $n_1 \geq_{\updownarrow} n_2$* (respectively, $n_1 \equiv_{\updownarrow} n_2$) in the statement of Proposition 6.1(1) and (2), as *such that $(m_1, n_1) \geq_{\updownarrow} (m_2, n_2)$* (respectively, $(m_1, n_1) \equiv_{\updownarrow} (m_2, n_2)$). Actually, Proposition 6.1 can be generalized, as follows.

PROPOSITION 6.3. *Let m_1, m_2 and n_1 be nodes of a document D such that $m_1 \geq_{\updownarrow} m_2$ (respectively, $m_1 \equiv_{\updownarrow} m_2$). Then there exists a node n_2 such that $(m_1, n_1) \geq_{\updownarrow} (m_2, n_2)$ (respectively, $(m_1, n_1) \equiv_{\updownarrow} (m_2, n_2)$).*

Proof. Obviously, it suffices to prove the case for perfect similarity. If n_1 is an ancestor of m_1, then Proposition 6.3 follows immediately from Proposition 6.1(1). Otherwise, consider top(m_1, n_1), the least common ancestor of m_1 and n_1. Again by Proposition 6.1(1), there is an ancestor t_2 of m_2 such that $(m_1, \text{top}(m_1, n_1)) \geq_{\updownarrow} (m_2, t_2)$. Since n_1 is a descendant of top(m_1, n_1), we have by Proposition 6.1(2), that there exists a descendant n_2 of t_2 such that $(\text{top}(m_1, n_1), n_1) \geq_{\updownarrow} (t_2, n_2)$. Using Proposition 3.1(3), it readily follows that $(m_1, n_1) \geq_{\updownarrow} (m_2, n_2)$. □

EXAMPLE 6.5. Continuing with Example 6.4, Proposition 6.3 states that, given the perfect similarity $n_6 \geq_{\updownarrow} n_8$ and the node n_7, there exists a node n such that $(n_6, n_7) \geq_{\updownarrow} (n_8, n)$. This is indeed the case; all possible choices for n are n_8 itself and n_9.

We are now ready to establish the link between the semantic properties of expression-relatedness and expression-equivalence on the one hand, and perfect similarity and perfect bisimilarity on the other hand. First, we will show that a Path$^+$ expression cannot distinguish a path from one that is perfectly similar to it. To make the case for composition, we need the following technical lemma.

LEMMA 6.1. *Let m_1, m_2, n_1, n_2 and p_1 be nodes of a document D such that $(m_1, n_1) \geq_{\updownarrow} (m_2, n_2)$. Then, there exists a node p_2 such that $(m_1, p_1) \geq_{\updownarrow} (m_2, p_2)$ and $(p_1, n_1) \geq_{\updownarrow} (p_2, n_2)$.*

Proof. Let q_1 be top(m_1, p_1) or top(p_1, n_1), whichever is closer to p_1. Without loss of generality, assume the former.[10] By Proposition 6.1(1), m_2 has an ancestor q_2 such that $(m_1, q_1) \geq_{\updownarrow} (m_2, q_2)$. By Proposition 6.1(2), q_2 has a descendant p_2 such that $(q_1, p_1) \geq_{\updownarrow} (q_2, p_2)$. Using Proposition 3.1(3), it readily follows that $(m_1, p_1) \geq_{\updownarrow} (m_2, p_2)$, whence also $(p_1, m_1) \geq_{\updownarrow} (p_2, m_2)$ (Proposition 3.1(2)). By assumption, $(m_1, n_1) \geq_{\updownarrow} (m_2, n_2)$. Also by assumption, top(p_1, n_1) is an ancestor of $q_1 = \text{top}(p_1, m_1)$, whence also of m_1. Hence, using once more Proposition 3.1(3), it follows that $(p_1, n_1) \geq_{\updownarrow} (p_2, n_2)$, which concludes the proof. □

EXAMPLE 6.6. Continuing with Examples 6.2 and 6.5, consider the perfect similarity $(n_5, n_6) \geq_{\updownarrow} (n_8, n_9)$. Given

[10]The proof of the other case is completely analogous, with the roles of m_1 and n_1, respectively, m_2 and n_2, interchanged.

node n_7, Lemma 6.1 states that there exists a node p such that $(n_5, n_7) \geq_\updownarrow (n_8, p)$ and $(n_7, n_6) \geq_\updownarrow (p, n_9)$. In addition, the proof provides a method to find such a node. Thereto, we must consider $\text{top}(n_5, n_7) = n_1$ and $\text{top}(n_7, n_6) = n_3$, and select the one which is closer to n_7, which is n_3. Since n_3 is an ancestor of n_6, we must consider the ancestor of n_9 at the same distance, which is n_4. Finally, we must choose a descendant p of n_4 such that $(n_3, n_7) \geq_\updownarrow (n_4, p)$. Two choices for p satisfy this requirement: $p = n_8$ or $p = n_9$. It is readily verified that $(n_5, n_7) \geq_\updownarrow (n_8, n_8)$ (respectively, $(n_5, n_7) \geq_\updownarrow (n_8, n_9)$) and that $(n_7, n_6) \geq_\updownarrow (n_8, n_9)$ (respectively, $(n_7, n_6) \geq_\updownarrow (n_9, n_9)$).

Lemma 6.2. *Let E be a Path$^+$ expression, and let m_1, m_2, n_1 and n_2 be nodes of a document D such that $(m_1, n_1) \geq_\updownarrow (m_2, n_2)$. If $(m_1, n_1) \in E(D)$, then $(m_2, n_2) \in E(D)$.*

Proof. By Theorem 4.1, we may assume without loss of generality that E is a Path$^+(\cap)$ expression. The proof now proceeds by structural induction. For the primitives $\emptyset$, ε, $\hat{\ell}$ ($\ell \in \mathcal{L}$), $\downarrow$ and $\uparrow$, it is straightforward that the lemma holds. This settles the base case. We now turn to the inductive step:

Composition. Let $E := E_1; E_2$, with E_1 and E_2 satisfying the lemma. Assume $(m_1, n_1) \in E(D)$. Then there exists a node p_1 such that $(m_1, p_1) \in E_1(D)$ and $(p_1, n_1) \in E_2(D)$. By Lemma 6.1, there exists a node p_2 such that $(m_1, p_1) \geq_\updownarrow (m_2, p_2)$ and $(p_1, n_1) \geq_\updownarrow (p_2, n_2)$. By the induction hypothesis, $(m_2, p_2) \in E_1(D)$ and $(p_2, n_2) \in E_2(D)$. Thus, $(m_2, n_2) \in E(D)$.

Intersection. Let $E := E_1 \cap E_2$, with E_1 and E_2 satisfying the lemma. Assume $(m_1, n_1) \in E(D)$. Then, $(m_1, n_1) \in E_1(D)$ and $(m_1, n_1) \in E_2(D)$. Then, by the induction hypothesis, $(m_2, n_2) \in E_1(D)$ and $(m_2, n_2) \in E_2(D)$. Thus, $(m_2, n_2) \in E(D)$. □

Notice that Lemma 6.2 implies, for every Path$^+$ expression E and document D, that $E(D)$ is a union of equivalence classes on the set of pairs of nodes of D under perfect bisimilarity.

We are now ready to develop an argument that shows that the semantic property of expression-equivalence and the syntactic property of perfect bisimilarity coincide. The argument is divided into the following three lemmas.

Lemma 6.3. *Let m_1 and m_2 be nodes of a document D. If $m_1 \geq_\updownarrow m_2$, then $m_1 \geq_{exp} m_2$.*

Proof. Assume that $m_1 \geq_\updownarrow m_2$. We show that $m_1 \geq_{\text{exp}} m_2$. Thereto, let E be a Path$^+$ expression. Suppose that $E(D)(m_1) \neq \emptyset$. Hence, there exists a node n_1 such that $(m_1, n_1) \in E(D)$. By Proposition 6.3, there exists a node n_2 such that $(m_1, n_1) \geq_\updownarrow (m_2, n_2)$. By Lemma 6.2, $(m_2, n_2) \in E(D)$. Hence, $E(D)(m_2) \neq \emptyset$. □

Lemma 6.4. *Let m_1 and m_2 be nodes of a document D. If $m_1 \geq_{exp} m_2$, then $m_1 \geq_\downarrow m_2$.*

Proof. The proof is by induction on the height of the complete subtree rooted at m_1. First observe that $m_1 \geq_{\text{exp}} m_2$ implies that $\lambda(m_1) = \lambda(m_2)$. Otherwise, $\widehat{\lambda(m_1)}(D)(m_1) \neq \emptyset$ and $\widehat{\lambda(m_1)}(D)(m_2) = \emptyset$, a contradiction. This settles the base case, in which m_1 is a leaf.

Thus, assume that m_1 is not a leaf. Let n_1 be a child of m_1. Then m_2 cannot be a leaf, for, otherwise, $\downarrow(D)(m_1) \neq \emptyset$ and $\downarrow(D)(m_2) = \emptyset$, a contradiction. Let $n_2^1, \ldots, n_2^k$ be all children of m_2. Suppose that, for all i, $1 \leq i \leq k$, $n_1 \not\geq_{\text{exp}} n_2^i$. Hence, for all i, $1 \leq i \leq k$, there exists a Path$^+$ expression E_i such that $E_i(D)(n_1) \neq \emptyset$ and $E_i(D)(n_2^i) = \emptyset$. Let $F := \Pi_1(E_1) \cap \ldots \cap \Pi_1(E_k)$. Then, $\downarrow; F(D)(m_1) \neq \emptyset$ and $\downarrow; F(D)(m_2) = \emptyset$, a contradiction. Hence, there exists a child n_2^i of m_2, $1 \leq i \leq k$, such that $n_1 \geq_{\text{exp}} n_2^i$. By the induction hypothesis, $n_1 \geq_\downarrow n_2^i$. By Definition 6.2, $m_1 \geq_\downarrow m_2$. □

Lemma 6.5. *Let m_1 and m_2 be nodes of a document D. If $m_1 \geq_{exp} m_2$, then $m_1 \geq_\updownarrow m_2$.*

Proof. The proof is by induction on the depth of m_1. First, observe that $m_1 \geq_{\text{exp}} m_2$ implies that $m_1 \geq_\downarrow m_2$, by Lemma 6.4. Now assume that m_1 is the root. Let d be the height of D. Then $\downarrow^d(D)(m_1) \neq \emptyset$. Hence, $\downarrow^d(D)(m_2) \neq \emptyset$, which implies that m_2 is also the root, i.e. $m_1 = m_2$. This settles the base case.

Thus, assume that m_1 is not the root. Hence $\uparrow(D)(m_1) \neq \emptyset$, which implies in turn that $\uparrow(D)(m_2) \neq \emptyset$. It follows that m_2 is not the root either. Let p_1 be the parent of m_1 and p_2 the parent of m_2. Let E be a Path$^+$ expression such that $E(D)(p_1) \neq \emptyset$. Then $\uparrow; E(D)(m_1) \neq \emptyset$, which implies that $\uparrow; E(D)(m_2) \neq \emptyset$. Hence $E(D)(p_2) \neq \emptyset$. We have thus shown that $p_1 \geq_{\text{exp}} p_2$. By the induction hypothesis, $p_1 \geq_\updownarrow p_2$. By Definition 6.3, $m_1 \geq_\updownarrow m_2$. □

Lemmas 6.3 and 6.5 now immediately yield the following.

Theorem 6.1. *Let m_1 and m_2 be nodes of a document D. Then, $m_1 \geq_{exp} m_2$ if and only if $m_1 \geq_\updownarrow m_2$.*

Theorem 6.1 now immediately yields the desired result.

Corollary 6.1. *Let m_1 and m_2 be nodes of a document D. Then, $m_1 \equiv_{exp} m_2$ if and only if $m_1 \equiv_\updownarrow m_2$.*

In words, two nodes in a document cannot be resolved by a Path$^+$ expression if and only if these nodes are perfectly bisimilar.

Next, we extend this result to the resolving power of Path$^+$ to pairs of paths in a document. The following theorem now states the main result about the resolution expressiveness of Path$^+$.

Theorem 6.2. *Let m_1, m_2, n_1, n_2 be nodes of a document D. Then, the property that, for each Path$^+$ expression E, $(m_1, n_1) \in E(D)$ implies $(m_2, n_2) \in E(D)$ is equivalent to the property $(m_1, n_1) \geq_\updownarrow (m_2, n_2)$.*

Proof. We only need to show the equivalence from left to right, as the other direction was already shown in Lemma 6.2. Thus, assume that, for each Path$^+$ expression E, $(m_1, n_1) \in E(D)$ implies $(m_2, n_2) \in E(D)$. In particular, $(m_1, n_1) \in \text{sig}(m_1, n_1)(D)$ implies $(m_2, n_2) \in \text{sig}(m_1, n_1)(D)$, whence

$(m_1, n_1) \geq (m_2, n_2)$. Next, assume that, for some Path$^+$ expression E, $E(D)(m_1) \neq \emptyset$. Then, necessarily, $(m_1, n_1) \in \Pi_1(E); \text{sig}(m_1, n_1)(D)$. From the assumption, it follows that $(m_2, n_2) \in \Pi_1(E); \text{sig}(m_1, n_1)(D)$, whence $E(D)(m_2) \neq \emptyset$. Thus, we have shown that $m_1 \geq_{\text{exp}} m_2$, or, by Theorem 6.1, that $m_1 \geq_{\updownarrow} m_2$. Finally, assume that, for some Path$^+$ expression E, $E(D)(n_1) \neq \emptyset$. Then, $(m_1, n_1) \in \text{sig}(m_1, n_1); \Pi_1(E)(D)$, whence $(m_2, n_2) \in \text{sig}(m_1, n_1); \Pi_1(E)(D)$. Again, this is only possible if $E(D)(n_2) \neq \emptyset$. As before, it follows that also $n_1 \geq_{\updownarrow} n_2$, which concludes the proof. □

The theorem states that, whenever we find a pair (m_1, n_1) in the result of a query in Path$^+$, then we are guaranteed that any pair (m_2, n_2) such that $(m_1, n_1) \geq_{\updownarrow} (m_2, n_2)$, will also be in the result of the query, and vice-versa.

We illustrate some of the concepts and results introduced above.

Example 6.7. Let D be the document shown in Fig. 3. In Example 6.2, we established that $n_2 \equiv_{\updownarrow} n_3$, whence, by Corollary 6.1, $n_2 \equiv_{\text{exp}} n_3$. Hence, there is no Path$^+$ expression that can distinguish these two nodes. In Example 6.3, we established that $(n_5, n_6) \geq_{\updownarrow} (n_8, n_9)$, but not vice-versa. By Theorem 6.2, there is no Path$^+$ expression E such that $(n_5, n_6) \in E(D)$, but $(n_8, n_9) \notin E(D)$. Also by Theorem 6.2, there does exist a Path$^+$ expression F such that $(n_8, n_9) \in F(D)$, but $(n_5, n_6) \notin F(D)$. An example of such an expression is $F := \uparrow; \downarrow$. As a last example, we established in Example 6.2 that $n_5 \equiv_{\updownarrow} n_7$. Hence, $(n_5, n_6) \geq_{\updownarrow} (n_7, n_9)$, but not vice-versa. By Theorem 6.2, there is no Path$^+$ expression E such that $(n_5, n_6) \in E(D)$, but $(n_7, n_9) \notin E(D)$, although there does exist a Path$^+$ expression F such that $(n_7, n_9) \in F(D)$, but $(n_5, n_6) \notin F(D)$. An example of such an expression is $F := \uparrow; \uparrow; \downarrow; \downarrow; \Pi_1(\downarrow)$.

To conclude this section, we observe that some of the results, which strictly speaking deal with the resolution expressiveness, can be lifted to the level of query expressiveness. We illustrate this in a final example.

Example 6.8. We give some examples of queries that cannot be expressed in Path$^+$.

(1) *Return all pairs of non-identical siblings in a document.* Let D be a document in which the nodes m and n are non-identical siblings such that $m \geq_{\updownarrow} n$. Obviously, $(m, n) \geq_{\updownarrow} (n, n)$. So, by Theorem 6.2, the output of any Path$^+$ query returning (m, n) must also contain (n, n), an identical pair.
(2) *Return all pairs of different nodes in a document.* It follows from the previous example that this is in general not possible in Path$^+$.
(3) *Return all identical pairs of nodes having at least two children.* In a document in which all nodes have the same label, two nodes at the same height and depth in the document are perfectly bisimilar, regardless of how many children these nodes have. Hence, by Theorem 6.1, they cannot be separated by a Path$^+$ query.

We also invite the reader to establish the inexpressibility in Path$^+$ of the proposed queries by analyzing the perfect similarity relations in our running example document (Fig. 3).

7. MINIMIZATION OF TREE QUERIES AND PATH$^+$ EXPRESSIONS

Minimizing algebraic expressions aimed at optimizing query evaluation is a practice used extensively in relational database systems. It is natural that the same principle and procedure is followed in XML query processing and optimization.

7.1. Minimization of tree queries

The results on minimization of tree queries are derived using the theory developed in Section 6. In particular, we show that each tree query can be translated into an equivalent unique minimal tree query.

In order to achieve these results, we first extend the notion of containment mapping (Section 3.4). Thereto, let $P_1 = (T_1, s_1, d_1)$ and $P_2 = (T_2, s_2, d_2)$ be tree queries, with $T_1 = (V_1, Ed_1, \lambda_1)$ and $T_2 = (V_2, Ed_2, \lambda_2)$. A *query containment mapping* of P_1 into P_2 is a mapping $h : V_1 \to V_2$ such that

(1) h is a containment mapping of T_1 into T_2,
(2) $h(s_1) = s_2$,
(3) $h(d_1) = d_2$.

We observe the following, which is a variation on a well-known result by Chandra and Merlin [23].

Proposition 7.1. *Let P_1 and P_2 be tree queries. Then P_2 is contained*[11] *in P_1 if and only if there is a query containment mapping of P_1 into P_2.*

Proof. Let $P_1 = (T_1, s_1, d_1)$ and $P_2 = (T_2, s_2, d_2)$. First, assume that P_2 is contained in P_1. Let D_2 be the document obtained from T_2 by replacing every wildcard by some fixed label of $\mathcal{L}$ that is neither in T_1 nor in T_2. The canonical embedding i_2 of T_2 in D_2 is a containment mapping, and, therefore, P_2 contained in P_1 implies that there is a containment mapping h from T_1 into D_2 mapping s_1 to s_2 and d_1 to d_2. Clearly, $i_2^{-1} \circ h$ is the desired query containment mapping.[12] Finally, assume that there exists a query containment mapping h_1 of P_1 into P_2. Let D be a document and let h_2 be a containment mapping of P_2 into D. Then $h_2 \circ h_1$ is a containment mapping of P_1 into D satisfying $h_2 \circ h_1(s_1) = h_2(s_2)$ and $h_2 \circ h_1(d_1) = h_2(d_2)$, yielding the desired containment. □

[11] In the sense that, for each document D, $P_2(D) \subseteq P_1(D)$.
[12] For two mappings $f : X \to Y$ and $g : Y \to Z$, the composition $g \circ f : X \to Z$ is the mapping for which, for all x in X, $g \circ f(x) = g(f(x))$.

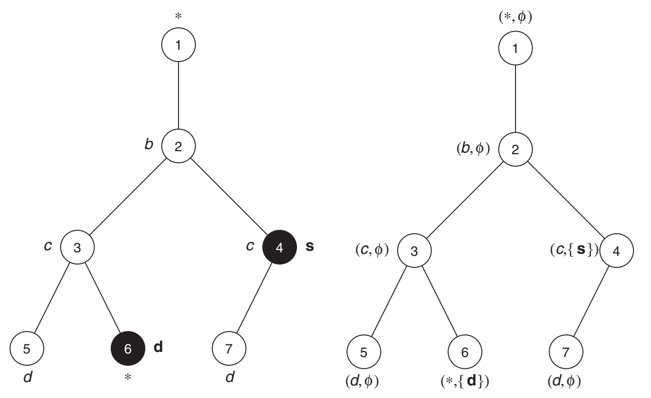


FIGURE 16. A tree query and its encoding as a labeled tree.

We are now going to describe an algorithm that reduces an arbitrary tree query to an equivalent minimal tree query. In order to do so, we must introduce some terminology and concepts. Given a tree query $P = (T, s, d)$ with $T = (V, Ed, \lambda)$, we wish to *encode* P as a labeled tree, i.e. incorporate the source and destination information into T. This encoding, denoted T_P, is a relabeling of T, i.e. $T_P = (V, Ed, \lambda_P)$ with $\lambda_P : V \to \overline{\mathcal{L}}$. Here[13], $\overline{\mathcal{L}} = (\mathcal{L} \cup \{*\}) \times 2^{\{s,d\}}$. It remains to define λ_P. For $n \in V$,

$$\lambda_P(n) = \begin{cases} (\lambda(n), \{s, d\}) & \text{if } n = s = d, \\ (\lambda(n), \{s\}) & \text{if } n = s \text{ and } s \neq d, \\ (\lambda(n), \{d\}) & \text{if } n = d \text{ and } s \neq d, \\ (\lambda(n), \emptyset) & \text{if } n \neq s \text{ and } n \neq d. \end{cases}$$

The encoding of the tree query in Fig. 5 in shown in Fig. 16.

We now define an order on $\overline{\mathcal{L}}$ (cf. Section 3.4), which is the reflexive–transitive closure of the following:

(1) For all $\ell \in \mathcal{L} \cup \{*\}$, and for all $S_1, S_2 \in 2^{\{s,d\}}$, if $S_1 \subseteq S_2$, then $(\ell, S_1) \geq (\ell, S_2)$.
(2) For all $\ell \in \mathcal{L}$, and for all $S \in 2^{\{s,d\}}$, $(*, S) \geq (\ell, S)$.

This order on $\overline{\mathcal{L}}$ is the one we use when talking about containment mappings between tree query encodings (Definition 3.5). The following result is straightforward.

Lemma 7.1. *Let P_1 and P_2 be two tree queries, and let T_{P_1} and T_{P_2} be their respective encodings. A query containment mapping of P_1 into P_2 is also a containment mapping of T_{P_1} into T_{P_2}, and vice-versa.*

This result actually justifies the term 'encoding'.

We now interpret labeled tree encodings of tree queries as documents and consider a variation on the similarity (Definition 6.2) and perfect similarity (Definition 6.3) relation between nodes, respectively, denoted as '$\geq_\downarrow$' and '$\geq_\updownarrow$'. Notably, we relax the condition that the first node has the same label as the second node to the condition that the fist node has a label that is at least the second label with respect to the order defined above. We will call this variation on (perfect) similarity *generalized* (*perfect*) *similarity* and denote it by '$\trianglerighteq_\downarrow$' (respectively, '$\trianglerighteq_\updownarrow$'). Most of the results proved for (perfect) similarity extend to generalized (perfect) similarity, in particular Proposition 6.1. The following lemma links tree query containment with generalized perfect similarity.

[13]By $2^{\{s,d\}}$, we mean the set of all subsets of $\{s, d\}$, i.e., $\{\emptyset, \{s\}, \{d\}, \{s, d\}\}$.

Lemma 7.2. *Let $T = (V, Ed, \lambda)$ be a labeled tree, with $\lambda : V \to \overline{\mathcal{L}}$, and let $i : V \to V$ be a containment mapping of T into itself. Then, for all $n \in V$, $n \trianglerighteq_\updownarrow i(n)$.*

Proof. We first show, by induction on the height of n, that $n \trianglerighteq_\downarrow i(n)$. For all $n \in V$, $\lambda(n) \geq \lambda(i(n))$, because i is a containment mapping. This settles the base case where n is a leaf. Now let n be an interior node, and let m be a child of n. Then $i(m)$ is a child of $i(n)$, because i is a containment mapping. By the induction hypothesis, $m \trianglerighteq_\downarrow i(m)$. By Definition 6.2, we may now conclude that $n \trianglerighteq_\downarrow i(n)$.

We next bootstrap this result, and show, by induction on the depth of n, that $n \trianglerighteq_\updownarrow i(n)$. First observe that, if n is the root, then $i(n) = n$. Indeed, let d be the height of T, and let $n = n_1, \ldots, n_d$ be a path from n to a leaf at maximal length. Then, $i(n) = i(n_1), \ldots, i(n_d)$ is in turn a path in T, because i is a containment mapping. Hence, $i(n)$ must be the root, whence $i(n) = n$. This settles the base case. Now suppose that n is not the root, and that p is the parent of n. Then, $i(p)$ is the parent of $i(n)$, because i is a containment mapping. By the induction hypothesis, $p \trianglerighteq_\updownarrow i(p)$. By Definition 6.3, we may now conclude that $n \trianglerighteq_\updownarrow i(n)$. □

If we combine Lemmas 7.1 and 7.2, we may conclude that, given a query containment mapping of a tree query into itself, each node of that tree query is generalized perfectly similar to its image, provided these nodes are interpreted in the context of the labeled tree encoding of the tree query.

Finally, notice that Algorithm **Merge2**, described in Fig. 7, also works if 'comparability' and 'compatibility' are defined with respect to the label order defined above.

We are now ready to prove the following key lemma.

Lemma 7.3. *Let P be a tree query and let T_P be its encoding as a labeled tree. Let m and n be different nodes of T_P such that $m \trianglerighteq_\updownarrow n$. Let **Merge2**$(T_P, m, n)$ be the result of applying Algorithm **Merge2** to T_P, m and n. Then, the following statements hold.*

(1) ***Merge2**(T_P, m, n) is again a labeled tree.*
(2) *There exists a tree query P' such that $T_{P'}$ = **Merge2**(T_P, m, n).*
(3) *The tree queries P and P' are equivalent.*

Proof. (1) By definition, perfectly similar nodes are compatible. Also, different perfectly similar nodes both have a parent, and these parents are again perfectly similar (Definition 6.3). From this, it follows in particular that the two nodes have the same distance to their least common ancestor. By Lemma 4.3, Algorithm **Merge2** will return a tree under these conditions.

(2) Let $P = (T, s, d)$, $T = (V, Ed, \lambda)$, $T_P = (V, Ed, \lambda_P)$, $\textbf{Merge2}(T_P, m, n) = T' = (V', Ed', \lambda')$ and $f : V \to V'$ be the canonical covering by T of T' (cf. Lemma 4.3). First notice that, if $s = d$, then this node is labeled in T_P by $(\lambda(s), \{s, d\})$, whereas all other nodes in T_P will have a label of the form $(\ell, \emptyset)$ with $\ell \in \mathcal{L} \cup \{*\}$. Obviously, all labels in T' also occur in T_P. Moreover, $f(s)$ in T' is still labeled by $(\lambda(s), \{s, d\})$. This is trivially so, if s is neither m nor n; in the other case, $f(s)$ will be labeled by $\min(\lambda_P(m), \lambda_P(n))$, which is $(\lambda(s), \{s, d\})$. Now, let $T'' = (V', Ed', \lambda'')$ with $\lambda'' : V' \to \mathcal{L} \cup \{*\}$ be defined by $\lambda''(f(s)) = \lambda(s)$ and $\lambda''(p') = \lambda'(p')$ for all other nodes $p' \in V'$. Clearly, $T' = T_{P'}$, with $P' = (T'', f(s), f(s))$. If $s \neq d$, then, in T_P, s is labeled by $(\lambda(s), \{s\})$ and d is labeled by $(\lambda(d), \{d\})$, whereas all other nodes in T_P will have a label of the form $(\ell, \emptyset)$ with $\ell \in \mathcal{L} \cup \{*\}$. The existence of a tree query P' for which $T' = T_{P'}$ can now be established by an argument completely analogous to the one in the case $s = d$.

(3) By Lemma 7.1, it suffices to establish the existence of containment mappings in both directions between T_P and T', using the notation introduced in the proof of the previous item. Obviously, the canonical covering f by T_P of T' is a containment mapping of T_P into T'. For the reverse direction, we will show by induction on the height of T_P that there exists a containment mapping $h' : V' \to V$ of T' into T_P satisfying the following conditions:

(1) if $n' = f(m) = f(n)$, then $h'(n') = n$, and
(2) for all $p \in V : p \trianglerighteq_{\updownarrow} h'(f(p))$.

We observe in this context that T_P and T' always have the same depth, as **Merge2** is depth-preserving. The existence of the desired containment mapping is trivial in the base case, where T_P and T' are both single-node trees. Now consider the inductive case. Let $\overline{T}_P$ and $\overline{T}'$ be the subtrees of T_P, respectively, T', obtained by cutting of all leaves at maximal depth. Since generalized perfectly similar nodes have the same depth, m and n are either both in $\overline{T}_P$ or both not in $\overline{T}_P$. In the first case, it readily follows that $\overline{T}' = \textbf{Merge2}(\overline{T}_P, m, n)$. In the second case, let p_m and p_n be the parents of m and n, respectively. By definition, $p_m \trianglerighteq_{\updownarrow} p_n$. It readily follows that $\overline{T}' = \textbf{Merge2}(\overline{T}_P, p_m, p_n)$. From the induction hypothesis, it follows in both cases that there exists a containment mapping h' of $\overline{T}'$ into $\overline{T}_P$ satisfying conditions (1) and (2) above.[14] Thereto, we extend h' to a containment mapping of T' into T_P, as follows. Let p' be a node of T' not in $\overline{T}'$, and let q' be its parent. We distinguish two cases.

(a) $p' = f(m) = f(n)$. By Condition (1), we must put $h'(p') := n$. Since, in this case, $q' = f(p_m) = f(p_n)$, it follows, again by Condition (1), that $h'(q') = p_n$. Hence, there is an edge between $h'(p')$ and $h'(q')$, so the proposed extension of h' is consistent with the first condition of the definition of containment mapping (Definition 3.5). Since, by assumption, $m \trianglerighteq_{\updownarrow} n$, and, trivially, $n \trianglerighteq_{\updownarrow} n$, m and n satisfy Condition (2) above. From $m \trianglerighteq_{\updownarrow} n$, it follows that $\lambda_P(m) \geq \lambda_P(n)$. By construction, $\lambda'(p') = \min(\lambda_P(m), \lambda_P(n)) = \lambda_P(n)$. It follows that $\lambda'(p') = \lambda_P(h'(p'))$, whence the proposed extension of h' is also consistent with the second condition of Definition 3.5.

(b) $p' \neq f(m) = f(n)$. In this case, there is a unique node $p \in V$ such that $f(p) = p'$. Let q be the parent of p in T_P. Since f is a containment mapping of T_P into T', $f(q) = q'$. By Condition (2) above, $q \trianglerighteq_{\updownarrow} h'(q')$. By Proposition 6.1 (2), $h'(q')$ has a child, say c, such that $p \trianglerighteq_{\updownarrow} c$. We put $h'(p') := c$. In doing so, we ensure that p satisfies Condition (2) above. In particular, there is an edge between $h'(p')$ and $h'(q')$, so the proposed extension of h' is consistent with the first condition of the definition of Definition 3.5. From $p \trianglerighteq_{\updownarrow} h'(p')$, it also follows that $\lambda_P(p) \geq \lambda_P(h'(p'))$. Hence, $\lambda'(p') = \lambda_P(p) \geq \lambda_P(h'(p'))$, whence the proposed extension of h' is also consistent with the second condition of Definition 3.5.

□

We are now ready to present our minimization algorithm **Reduce**. It is described in Fig. 17. Notice that this algorithm is non-deterministic, as, in each pass of the while loop, an arbitrary pair of generalized perfectly similar nodes must be chosen. The end result, however, turns out to be unique up to isomorphism, as is shown in Theorem 7.1.

We say that a tree query is *minimal* if there is no equivalent tree query with fewer nodes. The **T** expression $\emptyset$ is also assumed to be minimal. We now have the following.

THEOREM 7.1. *Let P be a **T** expression.*

(1) *Applying Algorithm **Reduce** to P always results in a **T** expression equivalent to P.*
(2) *Let P_{red} be a possible output of Algorithm **Reduce** upon input P. Then, P_{red} is isomorphic to any minimal **T** expression equivalent to P.*

Proof. Obviously, $P_{\text{red}} = \emptyset$ if and only if $P = \emptyset$, and the theorem holds trivially in this case. For the remainder of the proof, we therefore assume that both P and P_{red} are tree queries.

[14]Notice in this regard that the canonical covering by $\overline{T}_P$ of $\overline{T}'$ is the restriction of f to the nodes of $\overline{T}_P$.

```
Algorithm Reduce
Input: a T expression $P$;
Output: a T expression $P_{\mathrm{red}}$.
Method:
   if $P = \emptyset$ return $\emptyset$;
   let $P = (T, s, d)$;
   let $T_P$ be the encoding of $P$ as a labeled tree;
   let $T' := T_P$;
   while $T'$ contains pairs of different generalized perfectly similar nodes do
      let $m \neq n$ be nodes of $T'$ with $m \trianglerighteq_{\updownarrow} n$;
      let $T' :=$ Merge2$(T', m, n)$
   od;
   let $f$ be the canonical covering by $T_P$ of $T' = (V', Ed', \lambda')$;
               % f is the composition of all canonical coverings
               % by $T'$ of Merge2$(T', m, n)$ (before the assignment)
   let $T''$ be $T'$ in which each node label (which is an ordered pair) is replaced by its first component;
   let $P_{\mathrm{red}} = (T'', f(s), f(d))$; % $P_{\mathrm{red}}$ is the decoding of $T'$
   return $P_{\mathrm{red}}$.
```

FIGURE 17. Algorithm **Reduce**. (The module **Merge2** is described in Fig. 7.)

The first statement immediately follows from a repeated application of Lemma 7.3.

As to the second statement, let $P_{\mathrm{red}} = (T_{\mathrm{r}}, s_{\mathrm{r}}, d_{\mathrm{r}})$ with $T_{\mathrm{r}} = (V_{\mathrm{r}}, Ed_{\mathrm{r}}, \lambda_{\mathrm{r}})$. Let $P_{\mathrm{min}} = (T_{\mathrm{m}}, s_{\mathrm{m}}, d_{\mathrm{m}})$ with $T_{\mathrm{m}} = (V_{\mathrm{m}}, Ed_{\mathrm{m}}, \lambda_{\mathrm{m}})$ any minimal tree query equivalent to P. By the first statement, P_{red} and P_{min} are equivalent. By Proposition 7.1, there is a query containment mapping $h_{\mathrm{r}} : V_{\mathrm{r}} \to V_{\mathrm{m}}$ of P_{red} into P_{min} and a query containment mapping $h_{\mathrm{m}} : V_{\mathrm{m}} \to V_{\mathrm{r}}$ of P_{min} into P_{red}.

First notice that the image under h_{r} of T_{r} is a subtree of T_{m} containing both s_{m} and d_{m}, because h_{r} is a query containment mapping. Let $T_{h_{\mathrm{r}}}$ be this subtree. Then $P_{h_{\mathrm{r}}} = (T_{h_{\mathrm{r}}}, s_{\mathrm{m}}, d_{\mathrm{m}})$ is also a tree query. Clearly, h_{r} can also be seen as a query containment mapping of P_{red} into $P_{h_{\mathrm{r}}}$. Now, the image of $T_{h_{\mathrm{r}}}$ under h_{m} is a subtree of T_{r} containing both s_{r} and d_{r}, because h_{m} is a query containment mapping. Clearly, the restriction of h_{m} to the nodes in $T_{h_{\mathrm{r}}}$ is a query containment mapping of $P_{h_{\mathrm{r}}}$ into P_{red}. By Proposition 7.1, $P_{h_{\mathrm{r}}}$ is equivalent to P_{red}, and hence also to P_{min}. Since P_{min} is minimal by assumption, it follows that $P_{h_{\mathrm{r}}} = P_{\mathrm{min}}$ and that h_{r} is surjective.

Now, let $i = h_{\mathrm{m}} \circ h_{\mathrm{r}}$. As a composition of query containment mappings, $i : V_{\mathrm{r}} \to V_{\mathrm{r}}$ is a query containment mapping of P_{red} into itself. Let $n \in V_{\mathrm{r}}$. Then, by Lemma 7.2, $n \trianglerighteq_{\updownarrow} i(n)$ in $T_{P_{\mathrm{red}}}$. Since P_{red} is an output of Algorithm **Reduce**, it follows that $n = i(n)$. Hence, i is the identity on V_{r}. This is only possible if h_{r} is injective. Since it was already established that h_{r} is surjective, it follows that h_{r} is bijective. Hence, h_{m}, as its inverse, is also bijective. In particular, T_{r} has the same number of nodes as T_{min}, and, therefore, P_{red} is minimal.

We must conclude that h_{r} and h_{m} are bijective query containment mappings between P_{red} and P_{min} that are each other's inverses. As query containment mappings, h_{r} and h_{m} preserve edges and match sources, respectively, destinations. To establish that h_{r} and h_{m} are actually isomorphisms between P_{red} and P_{min}, it remains to show that corresponding nodes (through h_{r} and h_{m}) have the same label. So let $n \in V_{\mathrm{r}}$. Then, because h_{r} and h_{m} are containment mappings, $\lambda_{\mathrm{r}}(n) \geq \lambda_{\mathrm{m}}(h_{\mathrm{r}}(n)) \geq \lambda_{\mathrm{r}}(h_{\mathrm{m}}(h_{\mathrm{r}}(n))) = \lambda_{\mathrm{r}}(n)$. Hence, $\lambda_{\mathrm{r}}(n) = \lambda_{\mathrm{m}}(h_{\mathrm{r}}(n))$. □

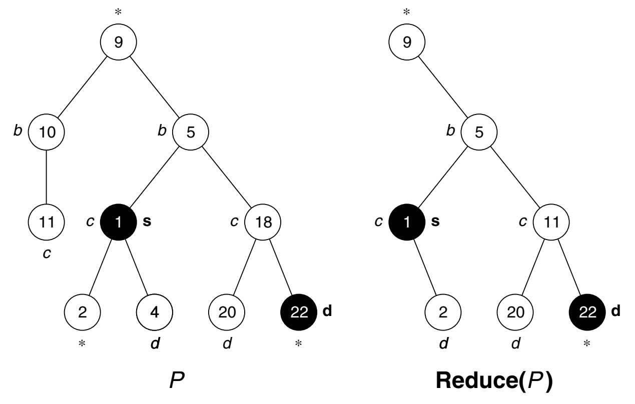


FIGURE 18. A tree query and its reduction.

We may thus conclude that, up to isomorphism, there is a unique tree query equivalent to a given one, and that we obtain it by applying Algorithm **Reduce** to it. For a given tree query P, the equivalent minimal tree query will henceforth be called the *reduction* of P and denoted as **Reduce**(P).

EXAMPLE 7.1. Consider again the final tree query in Fig. 11. Figure 18 exhibits this tree query, P, and its reduction, **Reduce**(P). The latter is the (up to isomorphism) unique minimal tree query equivalent to P. Observe that **Reduce**(P) is isomorphic to the tree query in Fig. 5.

7.2. Minimization of Path$^+$ expressions

While minimization works out fine in the declarative framework of tree queries, it is not obvious how to go about this strictly within the algebraic framework of Path$^+$ expressions. However, we can deal with the problem indirectly, as follows. First, we translate the Path$^+$ expressions under consideration into a tree query, using Algorithm **Path-T**. Then, we apply Algorithm **Reduce** to obtain the unique equivalent minimal tree query. Finally, we use Algorithm **T-Path** to re-translate this tree query into a Path$^+$ expression.[15] It is, of course, the hope that the minimization of the intermediate tree query will result in a significant reduction in the size of the original Path$^+$ expressions, and, in particular, in the amount of navigation required to evaluate it on a document.

EXAMPLE 7.2. By translating the Path$^+$ expression

$$\Pi_1(\downarrow);\ \Pi_2(\hat{d};\uparrow;\hat{c});\ \Pi_2(\hat{b};\downarrow;\hat{c});\uparrow;$$
$$\Pi_2(\Pi_1((\downarrow;\hat{b};\downarrow)\cap(\downarrow;\downarrow;\hat{c}));\downarrow);\downarrow;\Pi_1(\hat{c};\downarrow;\hat{d});\hat{c};\downarrow$$

of Example 3.2 into the final tree query of Fig. 11, and then re-translating this tree query, we obtained

$$\Pi_1(\Pi_1(\downarrow);\downarrow;\hat{d});\uparrow;\Pi_2(\Pi_1(\downarrow;\Pi_1(\downarrow;\hat{c});\hat{b});\downarrow);\hat{b};\downarrow;$$
$$\Pi_1(\downarrow;\hat{d});\hat{c};\downarrow$$

as an equivalent, normalized Path$^+$ expression in Example 5.1. In Example 7.1, we minimized the final tree query of Fig. 11 to the tree query shown in Fig. 18, (*right*), which is isomorphic to the tree query in Fig. 5. Re-translating the minimal query with Algorithm **T-Path** yields the normalized Path$^+$ expression

$$\Pi_1(\downarrow;\hat{d});\hat{c};\uparrow;\Pi_2(\downarrow);\hat{b};\downarrow;\Pi_1(\downarrow;\hat{d});\hat{c};\downarrow,$$

which was already claimed to be equivalent to the original one in Example 3.2.

We claim that normalized Path$^+$ expressions obtained in this way are *minimal* in the sense that any equivalent expression has at least as many '$\uparrow$' or '$\downarrow$' primitives combined.[16] However, our analysis will also show that there is no unique such minimal Path$^+$ expression, unless we further restrict our normal form.

The proof of this claim relies on the following lemma.

LEMMA 7.4. (1) *The combined number of '$\uparrow$' and '$\downarrow$' primitives in a Path$^+$ expression E is at least as large as the number of edges in* ***Path-T***(E).

(2) *The number of edges in a* ***T*** *expression P is precisely equal to the combined number of '$\uparrow$' and '$\downarrow$' primitives in* ***T-Path***(P).

Proof. For both statements, the proof is a straightforward structural induction argument. As to Statement 1, the possible inequality stems from the fact that the translation of composition and intersection may result in the merging of some edges of the partial translation of the subexpressions involved in the composition, respectively, intersection. □

We now have the following.

[15] This is effectively a variation on the normalization algorithm of Corollary 5.1.

[16] Of course, ∅ is also a minimal Path$^+$ expression.

THEOREM 7.2. (1) *If P is a minimal* ***T*** *expression, then* ***T-Path***(P) *is a minimal Path$^+$ expression.*

(2) *If E is a minimal Path$^+$ expression, then* ***Path-T***(E) *is a minimal* ***T*** *expression.*

Proof. (1) Obviously, the Path$^+$ expression ∅, as the translation of the minimal **T** expression ∅, is minimal. So, consider a minimal tree query P, and let E be any Path$^+$ expression equivalent to **T-Path**(P), whence also to P, of course. Then, by Proposition 4.2 and Theorem 7.1, **Reduce**(**Path-T**(E)) is isomorphic to P. By Lemma 7.4(1), the combined number of '$\uparrow$' and '$\downarrow$' primitives in E is at least as large as the number of edges in P. By Lemma 7.4(2), the number of edges in P is precisely equal to the combined number of '$\uparrow$' and '$\downarrow$' primitives in **T-Path**(P), from which the statement now follows.

(2) Obviously, the **T** expression ∅, as the translation of the minimal Path$^+$ expression ∅, is minimal. So, consider a Path$^+$ expression E which is not equivalent to ∅. The combined number of '$\uparrow$' and '$\downarrow$' primitives in E is at least as large as the number of edges in **Path-T**(E) (Lemma 7.4(1)), which is at least as large as the number of edges of **Reduce**(**Path-T**(E)), which is precisely equal to the combined number of '$\uparrow$' and '$\downarrow$' primitives in **T-Path**(**Reduce**(**Path-T**(E))) (Lemma 7.4(2)), which is at least the combined number of '$\uparrow$' and '$\downarrow$' primitives in the minimal Path$^+$ expression E. Hence, all the above inequalities are actually equalities. In particular, **Path-T**(E) and **Reduce**(**Path-T**(E)) have the same number of edges, implying that the former one is minimal, since the latter one is (Theorem 7.1). □

The following corollary is now immediate.

COROLLARY 7.1. *If E is a Path$^+$ expression, then* ***T-Path***(***Reduce***(***Path-T***(E))) *is a minimal Path$^+$ expression equivalent to E.*

EXAMPLE 7.3. From Example 7.2, it follows that

$$\Pi_1(\downarrow;\hat{d});\hat{c};\uparrow;\Pi_2(\downarrow);\hat{b};\downarrow;\Pi_1(\downarrow;\hat{d});\hat{c};\downarrow$$

is a minimal Path$^+$ expression equivalent to

$$\Pi_1(\downarrow);\ \Pi_2(\hat{d};\uparrow;\hat{c});\ \Pi_2(\hat{b};\downarrow;\hat{c});\uparrow;$$
$$\Pi_2(\Pi_1((\downarrow;\hat{b};\downarrow)\cap(\downarrow;\downarrow;\hat{c}));\downarrow);\downarrow;\Pi_1(\hat{c};\downarrow;\hat{d});\hat{c};\downarrow.$$

We conclude this subsection with some general remarks on the uniqueness of minimal Path$^+$ expressions. We discard expressions equivalent to ∅ from our discussion, as, in this case, ∅ is obviously the unique minimal expression equivalent to such an expression. First of all, intersections in minimal Path$^+$ expressions can only occur between subexpressions that return identical pairs of nodes, whatever the document under consideration. Otherwise, the merging involved in translating this intersection into the formalism of tree queries will eventually yield a tree query with fewer edges than the combined number of '↑' and '↓' primitives in the original expression, which, upon re-translation, would contradict the minimality of the original Path$^+$ expression. An intersection between subexpressions that always return identical pairs of nodes, however, is equivalent to the composition of these subexpressions. Thus, in a minimal Path$^+$ expression, intersection can always be replaced syntactically by composition, and, therefore, we only consider minimal Path$^+$ expressions without intersection. Also, inversion can be eliminated, and this elimination reduces the number of operations involved in the expression, while not affecting the combined number of '↑' and '↓' primitives. For this reason, it also makes sense not to consider minimal Path$^+$ expressions with explicit occurrences of the inversion operator. In summary, it makes good sense to consider only minimal expressions that are in Path$^+(\Pi_1, \Pi_2)$.

In a minimal Path$^+(\Pi_1, \Pi_2)$ expression, it is not possible that an '↑' primitive follows a '↓' primitive, for, otherwise, its translation into a tree query would require merging of edges, contradicting the minimality of the expression. So, in a minimal Path$^+(\Pi_1, \Pi_2)$ expression, all '↑' primitives precede all '↓' primitives. For the same reason, the second projection operation can occur only once, and only after the last '↑' primitive, if any, and before the first '↓' primitive, if any. Next, notice that subexpressions of minimal Path$^+$ expressions must be minimal, too. In particular, all '↑' primitives in the argument of a projection operation must precede all '↓' primitives. Since translating a minimal Path$^+$ into a tree query may not result in the merging of edges, we must conclude that '↑' primitives cannot occur inside projections.

In conclusion, equivalent minimal Path$^+(\Pi_1, \Pi_2)$ expressions are nearly isomorphic, except that

(1) multiple occurrences of 'Π_1' operations in between '↑' and/or '↓' primitives in one minimal expression may have been nested in another one; and
(2) there is some freedom in the precise location of label primitives.

Therefore, if we impose as additional restrictions onto minimal Path$^+(\Pi_1, \Pi_2)$ that

(1) the number of 'Π_1' operations at each level of nesting (with respect to the projections) is minimal, and
(2) the number of label primitives is minimal, and label conditions occur at the outermost level possible, and, there, as far to the right as possible,

then, minimal Path$^+(\Pi_1, \Pi_2)$ expressions are (i) unique and (ii) in normal form.

8. DECOMPOSITION AND EVALUATION

From the results in [2, 3], it follows that DPath$^+(\Pi_1)$ queries can be answered using an index-only plan with a $P(k)$-trie index for $k > 1$. We now discuss how to take advantage of this result and the normal form for Path$^+$ expressions established in Section 5 to come up with an efficient query evaluation plan for queries in Path$^+$.

Consider again a tree query $P = (T, s, d)$ in its most general form, as shown in Fig. 15. Let T'_r be T_r where t is replaced by the wildcard-labeled t', and T'_s be T_s where t is replaced by t'. Let $E := \textbf{T-Path}(P)$, which, by Theorem 5.1, is in normal form. We retain from the discussion in Section 5 that

$$\begin{aligned} E &= E_{\text{up}}; E_{\text{top}}; E_{\text{down}} \\ &= \textbf{T-Path}(T'_s, s, t'); \Pi_2(\textbf{T-Path}(T'_r, r, t')); \\ &\quad \textbf{T-Path}(T_d, t, d) \\ &= \textbf{T-Path}(T'_s, t', s)^{-1}; \Pi_2(\textbf{T-Path}(T'_r, r, t')); \\ &\quad \textbf{T-Path}(T_d, t, d). \end{aligned}$$

Here, $\textbf{T-Path}(T'_s, t', s)$, $\textbf{T-Path}(T'_r, r, t')$ and $\textbf{T-Path}(T_d, t, d)$ are all translations of tree queries in which the source is an ancestor of the destination, and, therefore, they are in DPath$^+(\Pi_1)$. Hence, for these subexpressions, an efficient query evaluation with an index-only plan is available.

In conclusion, every Path$^+$ query can be evaluated efficiently with an index-only plan provided a $P(k)$-trie index [3] with $k > 1$ is available, and this with no more than two natural join operations, as guaranteed by the normal form. Indeed, for every document D, we have that $E(D)$ equals

$$\begin{aligned} \textbf{T-Path}(T'_s, t', s)^{-1}(D) \bowtie \pi_2(\textbf{T-Path}(T'_r, r, t')(D)) \\ \bowtie \textbf{T-Path}(T_d, t, d)(D), \end{aligned}$$

where π_2 is defined on binary relations (as opposed to Π_2, which operates on Path$^+$ expressions), and, for a binary relation R, $\pi_2(R) = \{(n, n) \mid \exists m : (m, n) \in R\}$.

We now consider more general path queries by adding the set union and difference operations to Path$^+$. The set union operation alone does not fundamentally alter the query expressiveness results presented in this paper, since set union operations can be *pushed out* through algebraic transformation, resulting in a union of Path$^+$ expressions. On these subexpressions, the normalization and minimization results of this paper are still applicable. The set difference operation, however, significantly increases the query expressiveness of the language. In fact, it can be shown that adding set union, set difference and diversity[17] to Path$^+$ allows the formulation of all path queries that can be expressed with first-order formulas wherein at most three variables can occur [24]. Obviously, the presence of set difference allows the expression of non-monotonic path queries. Moreover, it also allows the expression

[17]The semantics of the *diversity* operator δ given a document $D = (V, Ed, \lambda)$ is the set $\delta(D) = \{(m, n) \mid m, n \in V \;\&\; m \neq n\}$.

of certain monotonic path queries that cannot be expressed in Path$^+$. To see this, consider again the monotonic queries that return all pairs of non-identical siblings in a document, respectively, all identical pairs of nodes having at least two children, of which it was shown in Example 6.8 that they cannot be expressed in Path$^+$. When the set difference operator is available, these queries can be expressed, however, as $\uparrow; \downarrow - \varepsilon$, respectively, $\Pi_2((\uparrow; \downarrow - \varepsilon); \uparrow)$. Consequently, path queries containing set difference can in general no longer be expressed as tree queries, whence our minimization and normalization algorithms are no longer applicable. However, notice that, at the core of these queries is a set difference of two Path$^+$ queries, each of which is in minimized normal form.

Ancestor–descendant relationships can be expressed in most of the semi-structured query languages, and have been included in the XPath languages in various studies (e.g. [5, 6, 8]). In this paper, we have regarded these relationships merely as the transitive closure operation of the primitive parent–child relationships, whose characteristics have been studied in the relational context. Furthermore, with proper encoding of the data—which represent the structural relationship of a semi-structured document—the ancestor–descendant relationship can be resolved via structural join [3, 19], which is a value join on the structural encoding.

In conclusion, the results developed for Path$^+$ can be used to process more general path queries. In this regard, one can view the Path$^+$ algebra to the Path algebra as one can view the project-select-join algebra to the full relational algebra.

9. COMPLEXITY-RELATED ISSUES

Although not the main concern of this paper, we briefly discuss some complexity-related issues with respect to the algorithms we presented.

A first concern we wish to address is the size of Path$^+(\cap)$ expressions. The focus of the paper has been on (normalized) Path$^+(\Pi_1, \Pi_2)$ expressions, and we have shown that it is possible to minimize such expressions in terms of the combined number of '$\uparrow$' and '$\downarrow$' primitives (Theorem 7.2). However, we also showed that Path$^+$, Path$^+(\Pi_1, \Pi_2)$ and Path$^+(\cap)$ are equivalent in expressive power (Proposition 4.1). Now, the naive translation algorithm from Path$^+$ expressions to Path$^+(\cap)$ expressions provided in the proof of Proposition 4.1 is exponential, because of the rules for translating projections. This may suggest that minimal Path$^+(\cap)$ expressions can in general be exponentially larger than their minimal Path$^+(\Pi_1, \Pi_2)$ counterparts. We will argue that this is *not* the case, however, by proposing an alternative translation algorithm. This translation algorithm first translates the given expression in a normalized, minimal Path$^+(\Pi_1, \Pi_2)$ expression as discussed in Subsection 7.2. The latter expression, say E, can be written as $E_{\text{up}}; E_{\text{top}}; E_{\text{down}}$. Let $\Pi_1(F)$ be a subexpression of E at the outer level, i.e. occurring at the outer level in E_{up} or E_{down}. Since E is in normal form, we know that F is of the form[18] $\Pi_1(F_1)^{l_1}\hat{\ell}^{l_2}(\downarrow\Pi_1(F_2)^{l_3}\hat{\ell}^{l_4})^k$, with $F_1, F_2 \in$ DPath$^+(\Pi_1)$, $l_1, l_2, l_3, l_4 \in \{0, 1\}$ and $k \in \{0, 1, 2, \ldots\}$. Clearly, $\Pi_1(F)$ is equivalent to $F; \uparrow^k$. By applying this rewriting rule top-down, we can entirely eliminate first projection from E_{up} and E_{down}. If $E_{\text{top}} \neq \emptyset$, we know it is of the form $\Pi_2(F)$, with F as above. Clearly, $\Pi_2(F)$ is equivalent to $\uparrow^k; F \cap \varepsilon$. Occurrences of the first projection in F can then be removed as above. In conclusion, we find that the resulting Path$^+(\cap)$ expression contains at most one intersection operator and the combined number of '$\uparrow$' and '$\downarrow$' operators occurring in it is at most twice the minimal number possible.[19]

We illustrate this on our running example.

EXAMPLE 9.1. Consider again the Path$^+$ expression given in Example 3.2:

$$\Pi_1(\downarrow); \Pi_2(\hat{d}; \uparrow; \hat{c}); \Pi_2(\hat{b}; \downarrow; \hat{c}); \uparrow;$$
$$\Pi_2(\Pi_1((\downarrow; \hat{b}; \downarrow) \cap (\downarrow; \downarrow; \hat{c})); \downarrow); \downarrow; \Pi_1(\hat{c}; \downarrow; \hat{d}); \hat{c}; \downarrow.$$

In Example 7.3, we established that

$$\Pi_1(\downarrow; \hat{d}); \hat{c}; \uparrow; \Pi_2(\downarrow); \hat{b}; \downarrow; \Pi_1(\downarrow; \hat{d}); \hat{c}; \downarrow$$

is an equivalent minimal Path$^+(\Pi_1, \Pi_2)$ expression in normal form. Eliminating the projection as explained above yields the Path$^+(\cap)$ expression

$$\downarrow; \hat{d}; \uparrow; \hat{c}; \uparrow; (\uparrow; \downarrow \cap \varepsilon); \hat{b}; \downarrow; \downarrow; \hat{d}; \uparrow; \hat{c}; \downarrow.$$

The number of combined occurrences of the '$\uparrow$' and '$\downarrow$' primitives in the latter expression is 9, compared with 6 in the minimal Path$^+(\Pi_1, \Pi_2)$ expression. Also notice that the latter expression contains only one intersection operator.

The algorithms **Merge1** (Fig. 6) and **Merge2** (Fig. 7) are auxiliary algorithms used in some of the main algorithms in this work. Obviously, **Merge1**(T_1, T_2, m_1, m_2) is linear in the smaller of the depth of m_1 in T_1 and the depth of m_2 in T_2. Similarly, **Merge2**(T, m_1, m_2) is linear in the smaller of the distance between m_1 and n and the distance between m_2 and n, where n is the least common ancestor of m_1 and m_2.

We next turn to the two central translation algorithms of this paper, **Path-T**, for translating Path$^+$ expressions into tree queries, and **T-Path**, for translating tree queries into Path$^+(\Pi_1, \Pi_2)$ expressions. We shall argue that both algorithms run in linear time.

We start with Algorithm **Path-T** (Fig. 9). Suppose that E is a Path$^+$ expression, and that the result of the translation is non-empty, i.e. a tree query. One can easily see that, at each time during the execution of the algorithm, the combined number

[18]Ignoring composition signs for simplicity.

[19]Notice that if we omit the minimization step in the translation algorithm described above, the combined number of '$\uparrow$' and '$\downarrow$' operators occurring in the Path$^+(\cap)$ expression is of course still at most twice the combined number of '$\uparrow$' and '$\downarrow$' operators in the Path$^+(\Pi_1, \Pi_2)$ expression.

of '$\uparrow$' and '$\downarrow$' operators occurring in the parts of E that have already been translated is equal to the total number of edges in the tree queries generated as partial translations plus the total number of node pairs that were compared in all applications of **Merge1** and **Merge2** combined. In other words, this is an invariant of the algorithm. If we also consider the case where the result is $\emptyset$, the invariant still holds, provided 'equal to' is replaced by 'greater than'. Now, the total number of cases encountered when Algorithm **Path-T** is applied to E is equal to the number of symbols in E. The total contribution to the time complexity of **Path-T** of the execution of steps that require constant time is therefore linear in the number of symbols in E. The steps which require time dependent on the size of the input for the case in which they occur are precisely the applications of **Merge1** and **Merge2**. Because of the linearity of these auxiliary algorithms and the invariant, we may conclude that their total contribution to the time complexity of **Path-T** is linear in the combined number of '$\uparrow$' and '$\downarrow$' operators occurring in E. Adding these up, we see that the time complexity of **Path-T**(E) is linear in the total number of symbols in E.

We now turn to Algorithm **T-Path** (Fig. 12). Suppose that P is a **T** expression. To compute the time complexity, we trace **T-Path**(P). Since the amount of work required in between consecutive nested calls of any pair of the six cases is input-independent, we may suffice with estimating the total number of recursive calls to these six cases. In doing so, we may of course ignore Case 0, which is only encountered if $P = \emptyset$, and therefore does not contribute to the overall time complexity of the algorithm. Therefore, we assume from now on that $P = (T, s, d)$ is a tree query. By Theorem 5.1, the normalized Path$^+(\Pi_1, \Pi_2)$ expression that results from **T-Path**(P) contains at most one occurrence of the second projection. As Case 1 is the only place in the algorithm where the second projection is generated, we may conclude that Case 1 is encountered at most once during the execution of **T-Path**(P). We also know that the subexpression contained within a first projection in the resulting expression contains at least one occurrence of the '$\downarrow$' primitive at the outer level. It is easily seen that each '$\downarrow$' primitive generated during the translation corresponds in a one-to-one fashion with an edge of P. Therefore, Case 4, the only place in the algorithm where the first projection is generated, is encountered at most $|T| - 1$ times during the execution of **T-Path**(P), where $|T|$ represents the number of nodes of T. Hence, Cases 1 and 4 combined are encountered at most $|T|$ times. Next, notice that, in each of the cases, the number of nodes in the input tree equals the total number of nodes in the output trees, except for Cases 1 and 4, in which one node is duplicated. Hence, Cases 2 and 3 combined are encountered at most $2|T|$ times. Finally, Cases 5 and 6 are the only cases in which '$\uparrow$' and '$\downarrow$' primitives are generated. As a consequence, these cases combined are encountered exactly $|T|-1$ times. Hence, at most $4|T|-1$ recursive calls to any of the six cases are encountered during the execution of **T-Path**(P). We may thus conclude that the time complexity of **T-Path**(P), with $P = (T, s, d)$, is also linear, this time in the number of nodes of T.

EXAMPLE 9.2. Consider the translation effectuated in Example 4.4, and visualized in Fig. 13. In this example, the tree query to be translated has 10 nodes. We count the number of recursive calls to each of the six cases in Algorithm **T-Path** and compare them with the estimates obtained above. There is 1 call to Case 1, and there are 5 calls to Case 4, or 6 to Cases 1 and 4 combined (or not more than 10, the maximal number estimated). There are 8 calls to Case 2, and there are 8 calls to Case 3, or 16 to Cases 2 and 3 combined (or not more than 20, the maximal number estimated). Finally, there are 8 calls to Case 5, and there is 1 call to Case 6, or 9 to Cases 5 and 6 combined (precisely the number estimated). In total, there are 31 recursive calls to one of the six cases (or not more than 39, the maximal number estimated).

As a consequence, we can translate an arbitrary Path$^+$ expression into an equivalent Path$^+(\Pi_1, \Pi_2)$ or Path$^+(\cap)$ expression in a time linear in the total number of symbols of the original expression.

Finally, we discuss the minimization algorithm **Reduce** (Fig. 17). We will argue that this algorithm is quadratic in the number of nodes of the input. Ignoring the irrelevant case that the input is empty, let $P = (T, s, d)$ be the tree query to be reduced. We relabel T to obtain T_P, the encoding of P as a labeled tree. Let $|T| = |T_P|$ be the number of nodes in T or T_P. We assume that, for every node of T_P, we have direct access to its leftmost child, next sibling and parent.[20] We also assign to each node its depth in the tree, and number these nodes from top to bottom, level per level, from left to right, i.e. such that the following conditions are satisfied:

(1) the number of a node is always larger than the number of any of its children,
(2) the children of a node are consecutively numbered and
(3) if the number of a first node is smaller than the number of a second node, then the number of any child of the first node is smaller than the number of any child of the second node.

Below, n_i denotes the node numbered i. All this preparatory work can be achieved in linear time. We create a $|T| \times |T|$ matrix in which cell (i, j) refers to the pair (n_i, n_j). Clearly, this matrix can be created in quadratic time. It will be used to compute all pairs of nodes (n_i, n_j) such that $n_i \trianglerighteq_{\updownarrow} n_j$. First, we will mark as deleted all cells that correspond to nodes that are not at the same depth in the tree, as a node can never be generalized perfectly similar to a node at another depth. Obviously, this can be done in quadratic time. Next, we compute generalized similarity ('$\trianglerighteq_{\downarrow}$') and mark all remaining pairs (n_i, n_j) such that $n_i \not\trianglerighteq_{\downarrow} n_j$ as deleted. We do that by going through the pairs of nodes in *reverse*

[20]In any implementation, we may assume the children of a node to be ordered.

lexicographic order, with two pointers. The first one is used to examine whether a pair (n_i, n_j) satisfies the required condition on the labels. If this is the case, we check if, for each child n_k of n_i, there exists a child n_ℓ of n_j such that $n_k \trianglerighteq_{\downarrow} n_\ell$. Since $i < k$, all such pairs (n_k, n_ℓ) have already been dealt with, and, therefore, it suffices to check whether there is such a pair that has not yet been marked as deleted. For that purpose, we use the second pointer. Since the numbering of parent nodes is always reflected in the ordering of their children, we will never have to backtrack with the second pointer. Therefore, we can compute generalized similarity on the node pairs that are at the same depth in the tree in quadratic time as well. In a very similar way, we can compute generalized perfect similarity from generalized similarity. This time, we traverse the node pairs in the regular lexicographic order. Each time we encounter a pair of different nodes, we check whether their parents are generalized similar, and mark the pair as deleted if the condition is not satisfied. For that purpose, we use the second pointer, and, because of the way we numbered nodes, we shall never have to backtrack with this second pointer. Hence, we may conclude that we can compute generalized perfect similarity in quadratic time. Finally, we can reduce the tree query, again by traversing the node pairs in the regular lexicographic order. In this way, we ensure that each time a pair (n_i, n_j) is encountered for which $n_i \trianglerighteq_{\updownarrow} n_j$, n_i and n_j share the same parent. Indeed, parent nodes are encountered before their children, and are generalized perfectly similar as soon as a child of one parent is generalized perfectly similar to a child of the other parent, and therefore would already have been merged to a single node. To effectuate the merging, mark the node n_i as deleted in the tree query, and make the children of n_i children of n_j. This operation does not affect the generalized perfect similarity relationship on the remaining nodes. In the course of traversing the node pairs, any pair in which one or both of the nodes has been marked as deleted in the tree should be ignored, of course. Since a node can change parent at most $|T|$ times, this final step in the algorithm is quadratic, too. Finally, we need to effectively remove the deleted nodes in the labeled tree and decode the result into the tree query that is the result of applying Algorithm **Reduce** to P. Clearly, this postprocessing step requires linear time. Hence, we may indeed conclude that the time complexity of **Reduce**(P), with $P = (T, s, d)$, is quadratic in the number of nodes of T.

As mentioned in Section 2, Ramanan [13] only considered tree queries in which the source equals the root. It is noteworthy, however, that the techniques he used in his minimization algorithm for the case he considered are not quite unlike ours. He also obtained quadratic time complexity, and suggests this is optimal.

10. FUTURE WORK

The work presented in this paper on Path$^+$ and the languages shown to be equivalent to it begs several other questions. Among the most interesting ones are those that have to do with forms of negation. One possible direction is investigating under which conditions tree queries are closed under set union, set difference and/or diversity.

In particular, we note that adding all three operators to Path$^+$ yields precisely the Relation Algebra of Tarski [25]. Another possible direction is studying extensions of Path$^+$ which allow weak forms of negation in the predication operations, but which are not necessarily as powerful as Core XPath [8]. Another direction to go is applying these languages to richer data structures such as various types of graphs, which would make the results particularly relevant for querying the semantic web. The present authors are currently investigating these issues.

ACKNOWLEDGEMENTS

The authors wish to thank the anonymous reviewers for their helpful comments on a previous version of this paper.

REFERENCES

[1] Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J. and Siméon, J. (2007) *XQuery 1.0: An XML query language. W3C Recommendation 23 January 2007*, http://www.w3.org/TR/xquery.

[2] Fletcher, G.H.L., Van Gucht, D., Wu, Y., Gyssens, M., Brenes, S. and Paredaens, J. (2008) A Methodology for Coupling Fragments of XPath with Structural Indexes for XML Documents. In Arenas, M., Schwartzbach, M.I. (eds) *Database Programming Languages, 11th Int. Symp., DBPL*, September 23–24, 2007, Vienna, Austria, Revised Selected Papers. Lecture Notes in Computer Science 4797, pp. 48–65. Springer, Berlin.

[3] Brenes, S., Wu, Y., Van Gucht, D. and Cruz, P.S. (2008) Trie Indexes for Efficient XML Query Evaluation. *Proc. 11th Int. Workshop on the Web and Databases, WebDB*, Vancouver, BC, June 13.

[4] Clark, J. and DeRose, S. (1999) *XML Path Language (XPath) version 1.0, W3C Recommendation 16 November 1999*, http://www.w3.org/TR/xpath.

[5] Gottlob, G., Koch, C. and Pichler, R. (2005) Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, **30**, 444–491.

[6] Benedikt, M., Fan, W. and Kuper, G.M. (2005) Structural properties of XPath fragments. *Theor. Comput. Sci.*, **336**, 3–31.

[7] Miklau, G. and Suciu, D. (2004) Containment and equivalence for a fragment of XPath. *J. ACM*, **51**, 2–45.

[8] Marx, M. and de Rijke, M. (2005) Semantic characterizations of navigational XPath. *SIGMOD Rec.*, **34**, 41–46.

[9] Gyssens, M., Paredaens, J., Van Gucht, D. and Fletcher, G.H.L. (2006) Structural Characterizations of the Semantics of XPath as Navigation Tool on a Document. *Proc. 25th ACM SIGACT-SIGART-SIGMOD Symp. Principles of Database Systems, PODS 2006* Chicago, IL, June 26–28, 2006, pp. 318–327. ACM Press, New York.

[10] ten Cate, B. (2006) The Expressivity of XPath with Transitive Closure. *Proc. 25th ACM SIGACT-SIGART-SIGMOD Symp.*

Principles of Database Systems, PODS, Chicago, IL, June 26–28, pp. 328–337. ACM Press, New York.

[11] Benedikt, M. and Koch, C. (2008) XPath leashed. *ACM Comput. Surv.*, **41**, 1–54.

[12] Wood, P.T. (2001) Minimising Simple XPath Expressions. *Proc. 4th Int. Workshop on the Web and Databases, WebDB*, Santa Barbara, CA, May 24–25, pp. 13–18.

[13] Ramanan, P. (2002) Efficient Algorithms for Minimizing Tree Pattern Queries. *Proc. 2002 ACM SIGMOD Int. Conf. Management of Data*, Madison, WI, June, 3–6, pp. 299–309. ACM Press, New York.

[14] Amer-Yahia, S., Cho, S., Lakshmanan, L.V.S. and Srivastava, D. (2002) Tree pattern query minimization. *VLDB J.*, **11**, 315–331.

[15] Paparizos, S., Patel, J.M. and Jagadish, H.V. (2007) SIGOPT: Using Schema to Optimize XML Query Processing. *Proc. 23rd Int. Conf. Data Engineering, ICDE*, Istanbul, Turkey, April 15–20, pp. 1456–1460. IEEE.

[16] Kimelfeld, B. and Sagiv, Y. (2008) Revisiting Redundancy and Minimization in an XPath Fragment. In Kemper, A., Valduriez, P., Mouaddib, N., Teubner, J., Bouzeghoub, M., Markl, V., Amsaleg, L. and Manolescu I., (eds.), *EDBT 2008, 11th Int. Conf. Extending Database Technology*, Nantes, France, March 25–29, Proceedings, ACM International Conference Proceedings Series 261, pp. 61–72. ACM Press, New York.

[17] Flesca, S., Furfaro F. and Masciari E. (2008) On the minimization of XPath queries. *J. ACM*, **55**, article no. 2.

[18] Fletcher, G.H.L., Van Gucht, D., Wu, Y., Gyssens, M., Brenes, S. and Paredaens, J. (2009) A methodology for coupling fragments of XPath with structural indexes for XML documents. *Inf. Syst.*, **34**, 657–670.

[19] Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., and Srivastava, D. (2002) Structural Joins: a Primitive for Efficient XML Query Pattern Matching. *Proc. 18th Int. Conf. Data Engineering, ICDE*, San Jose, CA, February 26–March 1, pp. 141–152. IEEE.

[20] Kaushik, R., Shenoy, P., Bohannon, P. and Gudes, E. (2002) Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. *Proc. 18th Int. Conf. Data Engineering, ICDE*, San Jose, CA, February 26–March 1, pp. 129–140. IEEE.

[21] Marx, M. (2005) Conditional XPath. *ACM Trans. Database Syst.*, **30**, 929–959.

[22] Wu, Y., Van Gucht, D., Gyssens, M. and Paredaens, J. (2008) A Study of Positive XPath with Parent/Child Navigation. IUCS Technical Reports 660, Indiana University, Bloomington, IN.

[23] Chandra, A.K. and Merlin, P.M. (1977) Optimal Implementation of Conjunctive Queries in Relational Data Bases. *Conf. Record of the 9th Annual ACM Symp. Theory of Computing, STOC*, Boulder, CO, May 2–4, pp. 77–90. ACM Press, New York.

[24] Tarski, A. and Givant, S. (1987) *A Formalization of Set Theory Without Variables*. American Mathematical Society, Providence, RI.

[25] Tarski, A. (1941) On the calculus of relations. *J. Symb. Log.*, **6**, 73–89.