The Expressivity of XPath with Transitive Closure*

Balder ten Cate
ISLA, Informatics Institute
Universiteit van Amsterdam
balder.tencate@uva.nl

ABSTRACT

We extend Core XPath, the navigational fragment of XPath 1.0, with transitive closure and path equalities. The resulting language, Regular XPath$^{\approx}$, is expressively complete for FO* (first-order logic extended with a transitive closure operator that can be applied to formulas with exactly two free variables). As a corollary, we obtain that Regular XPath$^{\approx}$ is closed under path intersection and complementation. We also provide characterizations for the $*$-positive fragment of Regular XPath$^{\approx}$, and for μRegular XPath (the extension of Regular XPath$^{\approx}$ with least fixed points).

Categories and Subject Descriptors
H.2.3 [Database Management]: Languages

General Terms
Languages, Theory

Keywords
Semi-structured data, XML, XPath, Transitive closure

1. INTRODUCTION

In this paper, we study the extension of XPath 1.0 with a transitive closure operator on path expressions. The reflexive transitive closure of a path expression A, which is denoted by A^*, is the infinite union

$$. \cup A \cup (A/A) \cup (A/A/A) \cup \cdots$$

Transitive closure is a natural and useful operation on path expressions [1], and many query languages for semistructured data support it (e.g., WebSQL [33], Lorel [3], UnQL [14]). XPath 1.0 and XPath 2.0 form a notable exception: they do not support transitive closure of arbitrary path expressions. Practical XML applications requiring the use of transitive closure, such as reported in [35], have therefore led researchers to propose and implement extensions of XPath that support it. As a result, the transitive closure operator is now part of the community standard EXSLT [20, 42] and is supported by the XML engines Saxon [41] and Xalan [44]. With it, all *regular path expressions* [1] are expressible [23], as well as all *caterpillar expressions* [13]. Moreover, the validity of a DTD can be expressed in XPath with transitive closure [30], thus opening the road to DTD-sensitive query optimization [21].

In spite of its clear practical benefits, not much is known about the theoretical properties of XPath extended with transitive closure. In particular, finding a characterization of the exact expressive power of this language is considered an important open problem [23]. Given the recent results for Core XPath and Conditional XPath [32, 31], a natural conjecture would be that XPath can express all of FO* (first-order logic extended with a transitive closure operator that operates on formulas with exactly two free variables). Our main result is a confirmation of this hypothesis. More specifically, we extend Core XPath, the navigational fragment of XPath 1.0 [24, 25], with transitive closure and path equalities. The resulting language, Regular XPath$^{\approx}$, is then characterized as follows:

THEOREM 1. *Regular XPath$^{\approx}$ and FO* have the same expressive power (both in terms of definable sets of nodes and in terms of definable binary relations).*

COROLLARY 2. *The class of binary relations definable in Regular XPath$^{\approx}$ is closed under intersection and complementation.*

Corollary 2 states that extending Regular XPath$^{\approx}$ with intersection and complementation operators on path expressions (as is done in XPath 2.0 with the `intersect` and `except` operators) would not lead to an increase in expressive power. In other words, Regular XPath$^{\approx}$ has the same expressive power as Tarski's relation algebra with transitive closure [39, 38].

Besides Theorem 1, we also give characterizations for the $*$-positive fragment of Regular XPath$^{\approx}$, and for μRegular XPath (the extension of Regular XPath$^{\approx}$ with least fixed points).

Theorem 1 and Corollary 2 fit in the line of research that was initiated by Benedikt, Fan and Kuper [9], and that investigates the expressive power and structural properties of variants of XPath. It complements the results of Marx and De Rijke [32] and Marx [31], which characterize Core XPath and Conditional XPath in terms of the two-variable fragment of first-order logic and full first-order logic, respectively.

Characterizations such as these are important because they provide landmarks in the space of possible query languages, much like Codd's expressive completeness result for relational algebras [15] did in the context of relational databases (cf. the discussion of

*A full version of this article containing additional appendices is available from the author's website: `http://staff.science.uva.nl/~bcate`.

I would like to thank Maarten Marx for many fruitful discussions and for his comments on earlier versions of this article, as well as Maarten de Rijke for useful comments. This research was supported by NWO under grant 612.069.006 and 639.021.508.

PODS'06, June 26–28, 2006, Chicago, Illinois, USA.

Codd completeness as a guiding principle in design of XPath 2.0 in Michael Kay's recent textbook [28]).

In light of the results of Benedikt et al. [9], we may conclude from Corollary 2 that Regular XPath$^{\approx}$, like Conditional XPath [31], is exceptionally well behaved: none of the languages studied in [9] are closed under complementation.

Outline of the paper: In section 2, we define the languages Regular XPath$^{\approx}$ and FO* and their semantics. Section 3 contains the proof of Theorem 1. In section 4 we consider some variations of the result. Finally, we conclude in section 5.

2. PRELIMINARIES

Trees. In this paper, we work with node-labelled finite sibling-ordered trees, where the node labels are elements of a fixed set of XML tag names. From now on, we will simply call such structures "trees". We will represent these trees as tuples $(N, R_\downarrow, R_\rightarrow, V)$, where N is the set of nodes of the tree, $R_\downarrow$ and $R_\rightarrow$ are the *child* and *immediate-next-sibling* relations, and V assigns to each label p (of some given set of XML tag labels), a subset $V(p) \subseteq N$.

Formally, a tree in our sense can be defined as any structure that can be represented (modulo isomorphism) as $(N, R_\downarrow, R_\rightarrow, V)$, where N is a finite set of finite sequences of natural numbers, closed under taking initial subsequences and such that for all sequences $\langle n_1, \ldots, n_k\rangle \in N$, also $\langle n_1, \ldots, n_k - 1\rangle \in N$ (unless $n_k = 0$), and the relations $R_\downarrow$ and $R_\rightarrow$ are given by $R_\downarrow = \{(\langle n_1, \ldots, n_k\rangle, \langle n_1, \ldots, n_k, n_{k+1}\rangle) \mid \langle n_1, \ldots, n_k, n_{k+1}\rangle \in N\}$ and $R_\rightarrow = \{(\langle n_1, \ldots, n_k\rangle, \langle n_1, \ldots, n_k+1\rangle \mid \langle n_1, \ldots, n_k+1\rangle \in N\}$. V is a function that assigns to each label p a subset $V(p) \subseteq N$.

We will use $R_\uparrow$ and $R_\leftarrow$ as abbrevations for $R_\downarrow^\smile$ and $R_\rightarrow^\smile$ (i.e., the converse relations of $R_\downarrow$ and $R_\rightarrow$). It may be assumed that every node $n \in N$ satisfies exactly one label, but all results in this paper are independent of such an assumption.

By a *binary* tree, we will mean a tree of which every node has at most two children.

Regular XPath$^{\approx}$. The language Regular XPath$^{\approx}$ that we will define next extends Core XPath (the navigational core of XPath 1.0 [24, 25]) with transitive closure and path equalities.

DEFINITION 3. *There are two sorts of expressions in* Regular XPath$^{\approx}$*:* path expressions *and* node expressions. *The variables $\alpha, \beta, \ldots$ will range over path expressions, and $\phi, \psi, \ldots$ over node expressions. The expressions of* Regular XPath$^{\approx}$ *are given by simultaneous induction:*

path expressions:
$\alpha ::= \uparrow \mid \downarrow \mid \leftarrow \mid \rightarrow \mid . \mid \alpha[\phi] \mid \alpha/\beta \mid \alpha \cup \beta \mid \alpha^*$

node expressions:
$\phi ::= p \mid \top \mid \neg\phi \mid \phi \wedge \psi \mid \langle\alpha\rangle \mid \mathit{loop}(\alpha)$

where p is an element of the set of XML tags. Each node expression defines a set of nodes in the tree, and each path expression defines a binary relation. The semantics of Regular XPath$^{\approx}$ *is given in Figure 1.*

We will use α^+ as shorthand for α/α^*, $\bigcup_{i\le n}\alpha_i$ for $\alpha_1 \cup \cdots \cup \alpha_n$, and $\alpha^\smile$ for the converse of α, which can be defined inductively. In path expressions of the form $\alpha[\phi]$, the subexpression $[\phi]$ will be referred to as a *test*.

The *loop* operator provides a convenient notation for the *path equalities* studied in [17, 9, 40]. Path equalities are node expressions of the form $\alpha \approx \beta$, where α and β are path expressions. They

$$
\begin{array}{ll}
[\![\alpha]\!]^M & = R_\alpha \quad \text{for } \alpha \in \{\downarrow, \uparrow, \leftarrow, \rightarrow\} \\
[\![.]\!]^M & = \{(n,n) \mid n \in N\} \\
[\![\alpha[\phi]]\!]^M & = \{(n,m) \in [\![\alpha]\!]^M \mid m \in [\![\phi]\!]^M\} \\
[\![\alpha/\beta]\!]^M & = \{(n,m) \mid (n,k) \in [\![\alpha]\!]^M \text{ and } (k,m) \in [\![\beta]\!]^M \\
 & \quad \text{for some } k \in N\} \\
[\![\alpha \cup \beta]\!]^M & = [\![\alpha]\!]^M \cup [\![\beta]\!]^M \\
[\![\alpha^*]\!]^M & = \{(n,m) \mid \text{there are } n_1, \ldots n_k \text{ such that } n = n_1, \\
 & \quad m = n_k, \text{ and } (n_\ell, n_{\ell+1}) \in [\![\alpha]\!]^M \text{ for } \ell < k\} \\
[\![p]\!]^M & = V(p) \\
[\![\top]\!]^M & = N \\
[\![\phi \wedge \psi]\!]^M & = [\![\phi]\!]^M \cap [\![\psi]\!]^M \\
[\![\neg\phi]\!]^M & = N \setminus [\![\phi]\!]^M \\
[\![\langle\alpha\rangle]\!]^M & = \{n \in N \mid (n,m) \in [\![\alpha]\!]^M \text{ for some } m \in N\} \\
[\![\mathit{loop}(\alpha)]\!]^M & = \{n \in N \mid (n,n) \in [\![\alpha]\!]^M\}
\end{array}
$$

Figure 1: The semantics of Regular XPath$^{\approx}$

$$
\begin{array}{ll}
\texttt{self} & = . \\
\texttt{child} & = \downarrow \\
\texttt{parent} & = \uparrow \\
\texttt{descendant} & = \downarrow^+ \\
\texttt{ancestor} & = \uparrow^+ \\
\texttt{descendant\_or\_self} & = \downarrow^* \\
\texttt{ancestor\_or\_self} & = \uparrow^* \\
\texttt{following\_sibling} & = \rightarrow^+ \\
\texttt{preceding\_sibling} & = \leftarrow^+ \\
\texttt{following} & = \uparrow^* / \rightarrow^+ / \downarrow^* \\
\texttt{preceding} & = \uparrow^* / \leftarrow^+ / \downarrow^*
\end{array}
$$

Figure 2: XPath axes in Regular XPath$^{\approx}$

are interpreted existentially: $\alpha \approx \beta$ holds at a node n iff *some* node m can be reached from n both via α and via β. In one direction, $\mathit{loop}(\alpha)$ is equivalent to path equality $\alpha \approx .$, and, in the other direction, $\alpha \approx \beta$ is equivalent to $\mathit{loop}(\alpha/\beta^\smile)$. Section 4.1 discusses *loop* and its influence on the expressivity of Regular XPath$^{\approx}$ in more detail.

Comparison with standard XPath notation. Our notation differs slightly from the official XPath syntax. We use only five primitive path expressions, $\uparrow$, $\downarrow$, $\leftarrow$, and $\rightarrow$ and ., corresponding to the XPath axes `parent::*`, `child::*`, `preceding_sibling::*[1]`, `following_sibling::*[1]` and `self::*`. Using transitive closure, all other navigational XPath axes can be expressed in terms of these (cf. Figure 2).

Also, in Regular XPath$^{\approx}$, the basic navigational steps are strictly separated from the node tests. For instance, we write $\uparrow^+[p]$ instead of XPath notation `ancestor::p`. Angled brackets are used to distinguish a path expression α from the node expression that checks for the *existence* of an α-successor.

Path equalities $\alpha \approx \beta$ can be expressed in XPath 1.0 as `count(`$\alpha|\beta$`) < count(`α`) + count(`β`)`, and in XPath 2.0 simply as the node test `(`α `intersect` β`)`. Note that the XPath 1.0 expression $\alpha = \beta$ expresses a different condition: it holds if there are elements that can be reached via α and β, respectively, with the same *string value*.

FO*. The main contribution of this paper is a characterization of the expressive power of Regular XPath$^{\approx}$, in terms of the follow-

ing language:

DEFINITION 4. *The language* FO* *extends* FO *(first-order logic with equality) with a transitive closure operator that operates on formulas with exactly two free variables: for any formula $\phi(x,y)$, $\phi^*(x,y)$ is again an* FO* *formula, and it expresses that x and y stand in the reflexive transitive closure of the relation defined by $\phi(x,y)$.*[1] *More precisely, $M \models \phi^*(x,y)\ [d,e]$ iff there are elements $d_1,\ldots,d_n$ of the domain of M such that $d = d_1$, $e = d_n$, and $M \models \phi(x,y)\ [d_i, d_{i+1}]$ for all $i < n$.*

In what follows, whenever we refer to FO*, we always have in mind the specific instance where the non-logical vocabulary consists of two binary relations, $R_\downarrow$ and $R_\rightarrow$, and countably many unary predicates corresponding to the XML tags.

Relations with other languages. Regular XPath$^\approx$ extends several other languages that have been studied in recent literature. The *loop*-free fragment of Regular XPath$^\approx$ is known as Regular XPath [30]. The fragment of Regular XPath in which $*$ can only be applied to the basic path expressions $\uparrow$, $\downarrow$, $\leftarrow$, $\rightarrow$, and in which $\leftarrow$ and $\rightarrow$ can only be used in the form $\leftarrow^+$ and $\rightarrow^+$, is called Core XPath [24, 25]. In between Core XPath and Regular XPath$^\approx$ lies Conditional XPath [31], the fragment of Regular XPath where $*$ is only allowed on path expressions of the form $\alpha[\phi]$ with $\alpha \in \{\uparrow,\downarrow,\leftarrow,\rightarrow\}$.

FO* is closely related to the languages FOREG [37] and $FO+TC_1$ [19]. In terms of expressive power, FO* lies in-between these languages. It extends FOREG by allowing transitive closure of arbitrary definable binary relations, not only vertical or horizontal ones. Indeed, FO* is a strict extension of FOREG, as the Boolean circuit problem can be expressed in FO* [4] but not in FOREG [37]. On the other hand, FO* is contained in $FO+TC_1$, which supports transitive closure of formulas with more than two free variables (where the extra free variables act as parameters). It is not known whether the latter inclusion is strict.

3. THE MAIN RESULT

This section is dedicated to the proof of Theorem 1. The translation given in Figure 3 shows that Regular XPath$^\approx$ is contained in FO*, in the sense that every Regular XPath$^\approx$ path expression is equivalent to an FO* formula in two free variables, and every Regular XPath$^\approx$ node expression is equivalent to an FO* formula in one free variable. In the remainder of this section, we prove the difficult direction of Theorem 1, by describing a translation from FO* formulas with two variables to Regular XPath$^\approx$ path expressions (the result for node expressions then follows).

First, we introduce *separated path expressions*, which constitute a normal form for Regular XPath$^\approx$ path expressions on binary trees. Then, we introduce the notion of *tree queries*, which, roughly speaking, generalize (separated) path expressions to a setting with more than two free variables. Finally, we inductively translate FO*-formulas to tree queries. The heart of the proof lies in Lemma 10, which shows that the separated path expressions are closed under taking transitive closure.

3.1 Upward and downward path expressions

We start by considering a simple type of path expressions, that traverse the tree in one direction only.

[1]Under the more common but less readable notation commonly used in finite model theory, $\phi^*(x,y)$ would be written as $(x = y) \vee [TC_{uv}\ \phi(u,v)](x,y)$.

$$
\begin{array}{lcl}
\mathrm{TR}_{xy}(\alpha) & = & R_\alpha(x,y) \text{ for } \alpha \in \{\downarrow,\uparrow,\leftarrow,\rightarrow\} \\
\mathrm{TR}_{xy}(.) & = & (x = y) \\
\mathrm{TR}_{xy}(\alpha[\phi]) & = & \mathrm{TR}_{xy}(\alpha) \wedge \mathrm{TR}_y(\phi) \\
\mathrm{TR}_{xy}(\alpha/\beta) & = & \exists z.(\mathrm{TR}_{xz}(\alpha) \wedge \mathrm{TR}_{zy}(\beta)) \\
\mathrm{TR}_{xy}(\alpha \cup \beta) & = & \mathrm{TR}_{xy}(\alpha) \vee \mathrm{TR}_{xy}(\beta) \\
\mathrm{TR}_{xy}(\alpha^*) & = & (\mathrm{TR}_{xy}(\alpha))^*(x,y) \\
 & & \\
\mathrm{TR}_x(p) & = & Px \\
\mathrm{TR}_x(\top) & = & \top \\
\mathrm{TR}_x(\phi \wedge \psi) & = & \mathrm{TR}_x(\phi) \wedge \mathrm{TR}_x(\psi) \\
\mathrm{TR}_x(\neg\phi) & = & \neg\mathrm{TR}_x(\phi) \\
\mathrm{TR}_x(\langle\alpha\rangle) & = & \exists y.\mathrm{TR}_{xy}(\alpha) \\
\mathrm{TR}_x(loop(\alpha)) & = & \mathrm{TR}_{xx}(\alpha)
\end{array}
$$

Figure 3: Translation from Regular XPath$^\approx$ to FO*

DEFINITION 5 (DOWNWARD AND UPWARD EXPRESSIONS). *A path expression is called* downward *(respectively* upward*) if it is built up from $\downarrow$ (respectively $\uparrow$) and . using the regular operations ($/$, $\cup$ and $*$) and arbitrary tests.*

Note that downward path expressions may contain occurences of $\uparrow$, $\leftarrow$ and $\rightarrow$, but only within tests. Likewise for upward path expressions.

LEMMA 6 (INTERSECTION AND COMPLEMENTATION). *For all downward path expressions α and β, the intersection $\alpha \cap \beta$ and the relative complement $\downarrow^* \cap (-\alpha)$ are equivalent to downward path expressions. Likewise for upward path expressions.*

PROOF. We will only give a short sketch of the proof here. A more elaborate argument can be found in the full version of this paper.

One may think of a downward path expression as defining a string language. The strings correspond to downward paths in trees, and the letters of the alphabet are truth assignments over the (finitely many) tests expressions occuring in the relevant path expression. In fact, downward path expressions define exactly all *regular* such string languages. By a classical result from automata theory, the latter class is closed under intersection and complementation. It follows that, for downward path expressions α and β, both $\alpha \cap \beta$ and $\downarrow^* \cap (-\alpha)$ can be written as downward path expressions. □

DEFINITION 7 (UNIFORM SPLITTINGS). *A* uniformly splitting downward path expression *is a downward path expression α of the form $\bigcup_{i \le n}(\alpha^{i,\mathrm{I}}/\alpha^{i,\mathrm{II}})$ (with $n \ge 1$), such that, whenever $x\ R^*_\downarrow\ y\ R^*_\downarrow\ z$ in a tree, then $x[\![\alpha]\!]z$ holds iff there is an $i \le n$ such that $x[\![\alpha^{i,\mathrm{I}}]\!]y$ and $y[\![\alpha^{i,\mathrm{II}}]\!]z$. Uniformly splitting upward path expressions are defined analogously.*

LEMMA 8 (EXISTENCE OF UNIFORM SPLITTINGS). *Each downward (upward) path expression α is equivalent to a uniformly splitting downward (upward) path expression.*

PROOF. We will prove the lemma for downward path expressions. The case for upward path expressions then follows (note that, for α a downward path expression, $\alpha^\smile$ is upward, and vice versa).

The proof proceeds by induction. Let α be any downward path expression. If α is of the form $\downarrow$ or ., then it has the uniform splitting $(\downarrow\ /.) \cup (./\ \downarrow)$ or $./.$, respectively. If α of the form $\beta \cup \gamma$, then the uniform splitting of α is simply the union of the uniform splittings of β and γ.

Suppose α is of the form $\beta[\phi]$. By induction hypothesis, β has a uniform splitting $\bigcup_{i\leq n}(\beta^{i,\mathrm{I}}/\beta^{i,\mathrm{II}})$, and hence $\bigcup_{i\leq n}(\beta^{i,\mathrm{I}}/\beta^{i,\mathrm{II}}[\phi])$ is a uniform splitting of α.

Next, suppose α is of the form β/γ. By induction hypothesis, β and γ have uniform splittings $\bigcup_{i\leq n}(\beta^{i,\mathrm{I}}/\beta^{i,\mathrm{II}})$ and $\bigcup_{i\leq m}(\gamma^{i,\mathrm{I}}/\gamma^{i,\mathrm{II}})$, respectively. Suppose that $x\ R^*_\downarrow\ y\ R^*_\downarrow\ z$ and $x[\![\alpha]\!]z$. Then, by definition, there is a node u such that $x[\![\beta]\!]u$ and $u[\![\gamma]\!]z$. Clearly, either $x\ R^*_\downarrow\ y\ R^*_\downarrow\ u\ R^*_\downarrow\ z$ or $x\ R^*_\downarrow\ u\ R^*_\downarrow\ y\ R^*_\downarrow\ z$. It follows that α is equivalent to $\bigcup_{i\leq n}(\beta^{i,\mathrm{I}}/(\beta^{i,\mathrm{II}}/\gamma)) \cup \bigcup_{i\leq m}((\beta/\gamma^{i,\mathrm{I}})/\gamma^{i,\mathrm{II}})$, and that the latter is in fact a uniform splitting.

Finally, let α be of the form β^*. By the induction hypothesis, β has a uniform splitting $\bigcup_{i\leq n}(\beta^{i,\mathrm{I}}/\beta^{i,\mathrm{II}})$. It follows that α is equivalent to $. \cup (\beta^*/\bigcup_{i\leq n}(\beta^{i,\mathrm{I}}/\beta^{i,\mathrm{II}})/\beta^*)$ and hence to $(./.) \cup \bigcup_{i\leq n}((\beta^*/\beta^{i,\mathrm{I}})/(\beta^{i,\mathrm{II}}/\beta^*))$, which is a uniform splitting. □

3.2 Separated path expressions

Next, we introduce separated path expressions. They form a natural common generalization of upward and downward path expressions. In fact, on binary trees, *every* path expression is equivalent to a separated path expression, as will follow from our later results.

DEFINITION 9 (SEPARATED PATH EXPRESSIONS). *A path expression is called* separated *if it is of the form $\bigcup_i(\alpha_i/\beta_i)$, where each α_i is an upward path expression and each β_i is a downward path expression.*

Note that every downward path expression α is equivalent to the separated path expression $\alpha/.$, and likewise for upward path expressions. Also note that, on binary trees, $\rightarrow$ is equivalent to $.[\langle\rightarrow\rangle]/\uparrow/\downarrow[\langle\leftarrow\rangle]$, and similarly for $\leftarrow$.

LEMMA 10 (CLOSURE UNDER TRANSITIVE CLOSURE). *For each separated path expression γ there is a separated path expression equivalent to γ^*.*

PROOF. Let $\gamma = (\alpha_1/\beta_1)\cup\cdots\cup(\alpha_n/\beta_n)$, where each α_i is an upward path expression and each β_i is a downward path expression. Clearly, γ^* is equivalent to

$$(\gamma^* \cap .) \ \cup\ (\gamma^* \cap \uparrow^+) \ \cup\ (\gamma^* \cap \downarrow^+) \ \cup\ (\gamma^* \cap \curvearrowright),$$

where $\curvearrowright$ is shorthand for $\uparrow^*/(\leftarrow^+ \cup \rightarrow^+)/\downarrow^*$. We claim that each of these four path expressions is equivalent to a separated path expression. It immediately follows that γ^* as a whole is equivalent to a separated path expression.

► First, let us consider $(\gamma^* \cap .)$. This is equivalent to the separated $./.[loop(\gamma^*)]$.

► Next, let us consider $(\gamma^* \cap \uparrow^+)$. We will prove that this path expression is in fact equivalent to an upward path expression. Before going into the formal details, let us provide some intuition. Suppose that two nodes, u and v, in some tree, stand in the relation $(\gamma^* \cap \uparrow^+)$. This situation is depicted in Figure 4(a). It is not hard to see that any witnessing "trace" of γ^* from u to v must be of the form depicted in Figure 4(b). Observe that each path in Figure 4(b) labelled α_{i_m} $(m \leq n)$ must visit the respective black "re-entry point" on the line from u to v. Hence, we can apply Lemma 8 to each α_{k_m}, obtaining a situation as depicted in Figure 4(c). Applying Lemma 8 once more, but now splitting on v, we obtain Figure 4(d). Finally, each reflexive arrow may be replaced by an instance of the *loop* predicate, leading us to the final analysis depicted in Figure 4(e). It follows that the path from u to v may be described by a single upward path expression, namely the composition of the tests and upward path expressions depicted in Figure 4(e).

Now, let us now make these observations precise, and define an upward path expression that is equivalent to $(\gamma^* \cap \uparrow^+)$ on all trees. Rather than writing down this expression explicitly, it is more convenient to present it in the form of a *generalized NFA*, i.e., a nondeterministic finite state automaton whose transitions are labelled by upward path expression. By a fundamental result in automata theory, any such generalized NFA is equivalent to a single upward path expression.

Recall that our path expression γ is of the form $(\alpha_1/\beta_1) \cup \cdots \cup (\alpha_n/\beta_n)$. As the states of our generalized NFA, we will use $\{1,\ldots,n,\mathsf{s},\mathsf{f}\}$, where s and f are designated starting and final states, respectively. The transitions, which follow the path described in Figure 4(e), are given in Figure 5 (for $i,j \in \{1,\ldots,n\}$). (Note the intersections with $\uparrow^+$, which ensure that corresponding actions move at least one step up in the tree. Keep in mind that, by Lemma 6, the intersection of two upward path expressions is equivalent to an upward path expression.)

It is not hard to see that the generalized NFA described above generates exactly the paths in the tree that are of the form depicted in Figure 4(e). Hence, the upward path expression corresponding to this automaton is equivalent to $(\gamma^* \cap \uparrow^+)$.

► Next, let us consider $(\gamma^* \cap \downarrow^+)$. As this path expression is equivalent to $((\gamma^\smile)^* \cap \uparrow^+)^\smile$, we can use exactly the same technique as in the previous case to find an equivalent downward path expression (note that the converse of a separated path expression is still a separated path expression, and the converse of an upward path expression is a downward path expression).

► Finally, consider $(\gamma^* \cap \curvearrowright)$. If the relation denoted by this path expression holds between two nodes, u and v, in a tree, this means that u and v are incomparable with respect to the descendant relation of the tree, and $u[\![\gamma^*]\!]v$. It is not hard to see that, in this case, there is a common ancestor w of u and v, and an $i \leq n$, such that $u[\![\gamma^*/\alpha_i)]\!]w$ and $w[\![\beta_i/\gamma^*]\!]v$ (it suffices to consider a "trace" of γ^* from u to v, and to pick as w the highest point in the tree that is visited on the trace). In other words, $(\gamma^* \cap \curvearrowright)$ is equivalent to $\bigcup_{i\leq n}\big(((\gamma^*/\alpha_i) \cap \uparrow^+)/((\beta_i/\gamma^*) \cap \downarrow^+)\big)$. By similar reasoning as in the previous two cases, we have that, for each $i \leq n$, $((\gamma^*/\alpha_i) \cap \uparrow^+)$ is equivalent to an upward path expression and $((\beta_i/\gamma^*) \cap \downarrow^+)$ is equivalent to a downward path expression. Hence, $(\gamma^* \cap \curvearrowright)$ is equivalent to a separated path expression. □

3.3 Tree queries

Our aim is to translate formulas $\phi(x,y) \in \mathsf{FO}^*$ to path expressions of Regular XPath$^\approx$. This is, of course, only possible for FO^*-formulas with two free variables. Indeed, it is not even clear what it would *mean* for a formula with more than two free variables to be equivalent to a Regular XPath$^\approx$ path expression. Still, we need to handle formulas with more free variables, in order to define our translation inductively. In this section, we introduce the notion of a *conjunctive tree query* [2, 31], which will help us overcome this problem. Intuitively, tree queries generalize Regular XPath$^\approx$ path expressions to a setting with more than two free variables.

DEFINITION 11 (TREE CONFIGURATION). *A* tree configuration *for a finite set of variables V is a finite tree (D,E) without sibling ordering, where D is the set of nodes of the tree and E the child relation, plus a function $f : V \to D$ mapping the variables in V to nodes of the tree. A tree configuration is* tight *if each node $d \in D$ either is in the range of f (i.e., is labelled by a variable), or has at least two children.*

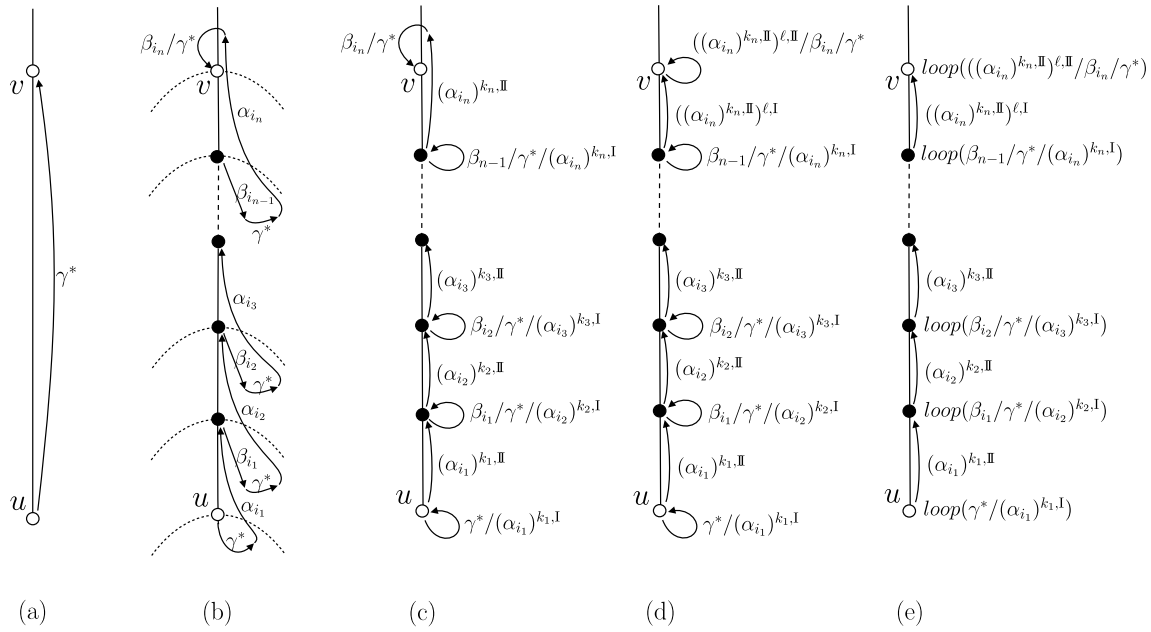


Figure 4: Decomposing $\gamma^* \cap \uparrow^+$ into tests and upward path expressions

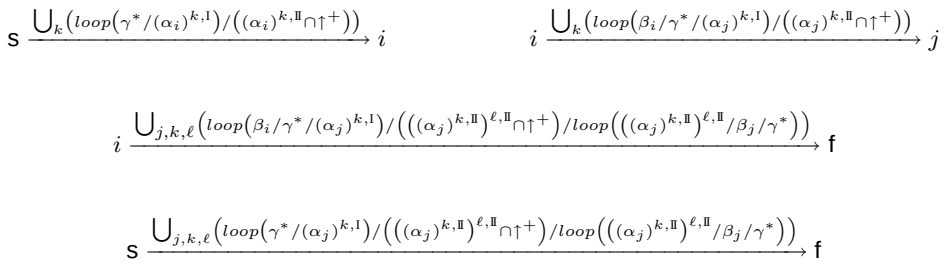


Figure 5: Transitions of the generalized NFA used in the proof of Lemma 10.

A tree configuration can be seen as a partial description of a model. The edges of the tree configuration represent strict descendent relations in the intended model, and nodes in the tree configuration with more than one child are required to be the least common ancestor of any pair of these children. Thus, a tree configuration completely describes the relative locations of the nodes named by the variables in V (and their least common ancestors). Tightness, then, requires that the tree configuration describes nothing more than that. Indeed, it is not hard to see that in a tight tree, every node is the least common ancestor of a set of nodes named by variables. In particular, this implies that there are only finitely many tight tree configurations over a finite set of variables.

DEFINITION 12 (TREE QUERY). *A* tree query *over a set of variables V consists of a tight tree configuration (D, E, f) for V plus a labeling function ℓ assigning to each node $d \in D$ a* Regular XPath$^{\approx}$ *node expression, and to each edge $e \in E$ a* Regular XPath$^{\approx}$ *downward path expression.*

The labeling function ℓ puts further restrictions on the intended model, by requiring that node expressions hold at particular nodes, and that downward path expressions hold between nodes and their descendants.

All the information expressed by a tree query (both the relative location of the variables in the tree and the additional requirements specified by the labeling function ℓ) can be written down by means of a formula of FO*. Indeed, tree queries can be seen as conjunctive queries over Regular XPath$^{\approx}$ expressions:

FACT 13. *Every tree query over a set of variables $\{x_1, \ldots, x_n\}$ can be written as an* FO**-formula of the form $\exists d_1 \ldots d_m . \bigwedge_k \phi_k$ where each ϕ_k has one of the following form:*

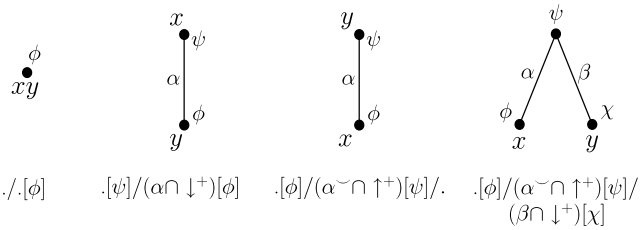
- $d_i R_\downarrow^+ d_j$,
- LCA(d_i, d_j, d_ℓ),
  where LCA(x, y, z) *is shorthand for $xR_\downarrow^* y \wedge xR_\downarrow^* z \wedge \forall u.(uR_\downarrow^* y \wedge uR_\downarrow^* z \rightarrow uR_\downarrow^* x)$, which expresses that x is the least common ancestor of y and z.*
- $d_i = x_j$ *(representing the variable labeling f),*
- TR$_{d_i}(\psi)$ *for some* Regular XPath$^{\approx}$ *node expression ψ (representing a node label),*
- TR$_{d_i,d_j}(\alpha)$ *for some* Regular XPath$^{\approx}$ *path expression α (representing an edge label).*

We will treat tree queries on a par with the FO*-formulas they represent. In particular, we will be taking disjunctions of tree queries, and we will speak of equivalence between FO*-formulas and (disjunctions of) tree queries.

The following proposition confirms our earlier statement that tree queries generalize Regular XPath$^{\approx}$ path expressions to a setting with more than two free variables.

PROPOSITION 14. *For every query tree over $\{x, y\}$ there is an equivalent separated path expression α. Conversely, every separated path expression is equivalent to a disjunction of tree queries over $\{x, y\}$.*

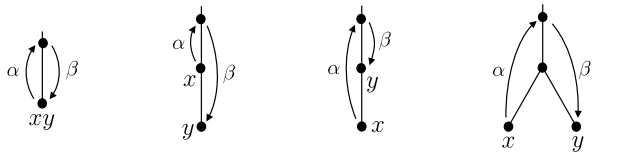
PROOF. [$\Rightarrow$] Every tree query over $\{x, y\}$ is of one of the following four kinds, and is equivalent to the corresponding separated path expression:



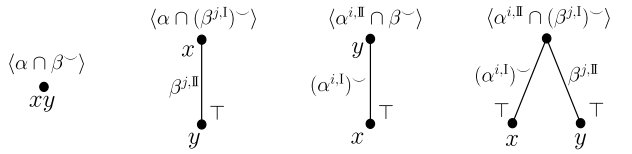
$./.[\phi]$ $\quad .[\psi]/(\alpha \cap \downarrow^+)[\phi]$ $\quad .[\phi]/(\alpha^\smile \cap \uparrow^+)[\psi]/.$ $\quad .[\phi]/(\alpha^\smile \cap \uparrow^+)[\psi]/(\beta \cap \downarrow^+)[\chi]$

Note that only path intersections as specified in Lemma 6 are used.

[$\Leftarrow$] It suffices to show that all path expressions of the form α/β, with α upward and β downward, are equivalent to disjunctions of tree queries over $\{x, y\}$ (note that separated path expressions are unions of such path expressions). Consider therefore any upward path expression α and downward path expression β, and let x and y are nodes in a tree that stand in the α/β relation. Then, one of the following four cases applies.



(where some of the nodes may coincide.) By Lemma 8, α and β have uniform splittings $\bigcup_{i \le n}(\alpha^{i,\mathrm{I}}/\alpha^{i,\mathrm{II}})$ and $\bigcup_{j \le m}(\beta^{j,\mathrm{I}}/\beta^{j,\mathrm{II}})$, respectively. It follows that α/β is equivalent to the disjunction of all tree queries of the following corresponding forms (with $i \le n$ and $j \le m$):



Again, only path intersections as specified in Lemma 6 are used. □

3.4 The main argument

Having introduced all the necessary concepts, we are now ready to proceed with the translation of formulas $\phi(x, y) \in \mathsf{FO}^*$ to path expressions of Regular XPath$^{\approx}$. We will first restrict attention to binary trees, i.e., trees in which every node has at most two children.

THEOREM 15. *Let V be a finite, non-empty set of variables. On binary trees, every FO^*-formula ϕ whose free variables are included in V is equivalent to a finite disjunction of binary tree queries over V.*

PROOF. We proceed by induction on ϕ. The base case (where ϕ is an atomic formula) is easy:

► The FO^* formula $\top$ is equivalent to the disjunction of all (finitely many) binary tree queries (D, E, f, ℓ) over V, where $\ell(d) = \top$ for all nodes $d \in D$, and $\ell(e) = \downarrow^+$ for all edges $e \in E$.

► The FO^* formula $x = y$ is equivalent to the disjunction of all (finitely many) binary tree queries (D, E, f, ℓ) over V, in which $f(x) = f(y)$, where $\ell(d) = \top$ for all nodes $d \in D$, and $\ell(e) = \downarrow^+$ for all edges $e \in E$.

► The FO^* formula Px is equivalent to the disjunction of all (finitely many) binary tree queries (D, E, f, ℓ) over V, where $\ell(f(x)) = p$, $\ell(d) = \top$ for all nodes $d \in D \setminus \{f(x)\}$, and $\ell(e) = \downarrow^+$ for all edges $e \in E$.

► The FO^* formula $xR_{\downarrow}y$ is equivalent to the disjunction of all (finitely many) binary tree queries (D, E, f, ℓ) over V, where $\langle f(x), f(y)\rangle \in E$, $\ell(\langle f(x), f(y)\rangle) = \downarrow$, $\ell(d) = \top$ for all nodes $d \in D$, and $\ell(e) = \downarrow^+$ for each edge $e \in E \setminus \{\langle f(x), f(y)\rangle\}$.

► The FO^* formula $xR_{\rightarrow}y$ is equivalent to the disjunction of all (finitely many) binary tree queries (D, E, f, ℓ) over V, where $f(x)$ and $f(y)$ are distinct children of a node $u \in D$, and where $\ell(f(x)) = \langle\rightarrow\rangle$, $\ell(f(y)) = \langle\leftarrow\rangle$, $\ell(\langle u, f(x)\rangle) = \ell(\langle u, f(y)\rangle) = \downarrow$, and $\ell(d) = \top$ for all other $d \in D$ and $\ell(e) = \downarrow^+$ for all other $e \in E$.

Now for the inductive step:

► Let $\phi \equiv \phi_1 \wedge \phi_2$. By induction hypothesis, ϕ_1 and ϕ_2 are both equivalent to a disjunctions of binary tree queries $(s_1 \vee \cdots \vee s_n)$ and $(t_1 \vee \cdots \vee t_m)$. It follows by the Boolean distribution law that ϕ is equivalent to $\bigvee_{i \le n, j \le m}(s_i \wedge t_j)$. It therefore suffices to show that the conjunction of two binary tree queries can be written as a disjunction of binary tree queries.

Let s and t be binary queries over the same set of variables V. If s and t are based on different (more precisely, non-isomorphic) tree configurations, then they cannot be satisfied at the same time, and hence $s \wedge t$ is equivalent to the empty disjunction. If, on the other hand, s and t share the same underlying tree configuration, then a new tree query may be obtained by taking conjunctions of corresponding node labels and intersections of corresponding edge labels (using Lemma 6). The resulting binary tree query is clearly equivalent to $s \wedge t$.

► Let $\phi \equiv \neg\psi$. By the induction hypothesis, ψ is equivalent to a disjunction of binary tree queries $(t_1 \vee \ldots \vee t_n)$. It follows that ϕ is equivalent to $(\neg t_1 \wedge \cdots \wedge \neg t_n)$, and hence, it suffices to show that the negation of a single tree query is equivalent to a disjunction of tree queries (we already explained above how to handle conjunctions).

Let $t = (D, E, f, \ell)$ be any binary tree query over a set of variables V. If t is not satisfied in a model, then that means that one of the following two holds:

- The tree configuration underlying t does not correctly represent the relative location of the nodes named by the variables and their least common ancestors. But this is equivalent to saying that one of the finitely many other, non-isomorphic, tight binary tree configurations holds.

- The tree configuration underlying t correctly represents the relative location of the nodes named by the variables and their least common ancestors, but the path expression α assigned to some edge, or the node expression ϕ associated to some node, does not hold in the model. But this can also be expressed by means of a binary tree query, namely the tree query based on the same tree configuration as t, in which the

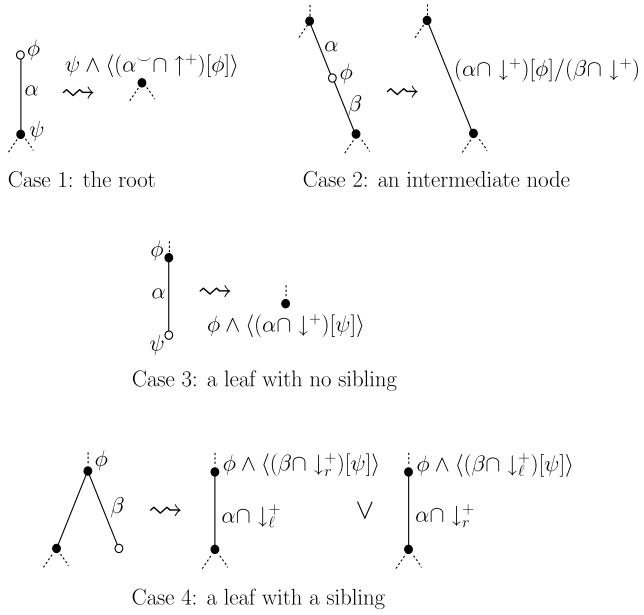


Figure 6: Removal of anonymous nodes from a tree query

labeling function assigns $\downarrow^+ \setminus \alpha$, respectively $\neg\phi$, to the relevant node or edge, and $\top$ or $\downarrow^+$ to all other nodes and edges (again, using Lemma 6).

Thus, $\neg t$ is equivalent to a disjunction of binary tree queries (over the same set of variables V).

► Let $\phi \equiv \exists y.\psi$. Without loss of generality, we may assume that $y \notin V$. By the induction hypothesis, ψ is equivalent to a disjunction of binary tree queries over $V \cup \{y\}$. We start by removing the label y from each of these tree queries. The resulting tree queries might not be tight: they might contain "*anonymous nodes*": nodes that are not labelled with a variable and with less than two children. However, these anonymous nodes can be successively eliminated by means of the rewrite rules depicted in Figure 6 (where $\downarrow_\ell^+$ and $\downarrow_r^+$ are shorthand for $\downarrow [\langle\rightarrow\rangle]/\downarrow^*$ and $\downarrow [\langle\leftarrow\rangle]/\downarrow^*$, respectively). Note that these rewrite rules only use intersection of upward path expressions and intersection of downward path expressions (cf. 6). Thus, $\exists y.\phi$ is equivalent to a disjunction of (tight) binary tree queries over V.

► Let $\phi \equiv TC_{xy}(\psi(x,y))$. By induction hypothesis, ψ is equivalent to a disjunction of binary tree queries over x, y, and hence, by Proposition 14, to a separated path expression α. It follows by Lemma 10 that α^* is equivalent to a separated path expression, and hence, by Proposition 14, to a disjunction of (binary) tree queries over $\{x, y\}$. □

Theorem 1 now follows:

PROOF OF THEOREM 1. Follows from Theorem 15 and Proposition 14, for the case of binary trees. The general case follows using the technique explained in the full version of this paper. □

4. VARIATIONS ON THEOREM 1

4.1 The role of the *loop* predicate

The proof of Theorem 1, and, more specifically, that of Lemma 10, relies on the presence of *loop* in the language. This stands in contrast with the known characterizations of Core XPath and Conditional XPath [32, 31], which do not rely on the use of *loop*. Indeed, because both Core XPath and Conditional XPath are closed under intersection of path expressions [26, 31], adding *loop* does not increase the expressivity of these languages (recall that $loop(\alpha)$ is equivalent to $\langle\alpha \cap .\rangle$).

The computational effects of adding *loop* to the language are not yet well understood (although some results on weaker fragments of XPath containing path equalities can be found in [17, 9]). In particular, while the query containment problem for Core XPath and for Regular XPath$^{\approx}$ without *loop* is ExpTime-complete (cf. Section 4.2), we were not able to find an upperbound on the complexity of the query containment problem for full Regular XPath$^{\approx}$ or for Core XPath extended with *loop*. Similarly, while the query evaluation problem for Regular XPath is solvable in linear time [30], standard bottom-up techniques only give us a quadratic upper bound for the full Regular XPath$^{\approx}$ (and, indeed, for Core XPath extended with *loop*) [29].

This raises the question whether the expressive completeness of Regular XPath$^{\approx}$ really depends on the presence of *loop*. Although we have not been able to prove it, we believe that *loop* does indeed contribute to the expressive power of Regular XPath$^{\approx}$. In particular, we conjecture that the property

(†) *The current node has an even number of descendants*

cannot be expressed in Regular XPath (i.e., without the use of *loop*). This would imply that Regular XPath is not closed under intersection of path expressions (recall that $loop(\alpha)$ is equivalent to $\langle\alpha \cap .\rangle$), and hence is unlikely to admit a natural characterization in terms of an extension of first-order logic.

Note that (†) *is* expressible in Regular XPath$^{\approx}$ using *loop*. Indeed, let next be the path expression

$$\downarrow [\neg\langle\leftarrow\rangle] \;\cup\; .[\neg\langle\downarrow\rangle]/(\uparrow \text{ while } \neg\langle\rightarrow\rangle)/ \rightarrow$$

where $(\alpha \text{ while } \phi)$ is shorthand for $(.[\phi]/\alpha)^*$. It is not hard to see that next defines the successor relation in the depth-first left-to-right tree ordering (the "document order"), and hence the node expression $loop\big((\text{next}/\text{next})^*/(\uparrow \text{ while } \neg\langle\rightarrow\rangle)\big)$ holds at a node iff it has an even number of descendants.

Note also that (†) is expressive without *loop* in the special case where the starting node is the root: in that case $\langle(\text{next}/\text{next})^*/(\uparrow \text{ while } \neg\langle\rightarrow\rangle)[\neg\langle\uparrow\rangle]\rangle$ does the job.

4.2 Adding least fixed points

Transitive closure only provides a limited form of recursion. In this section, we consider a further extension of Regular XPath$^{\approx}$ with *least fixed points*. We call this language μRegular XPath. Formally, we introduce countably many new symbols, called *set variables* (notation: $X, Y, \ldots$) and extending the syntax of Regular XPath as follows:

path expressions:
$\alpha ::= \uparrow \mid \downarrow \mid \leftarrow \mid \rightarrow \mid . \mid \alpha[\phi] \mid \alpha/\beta \mid \alpha \cup \beta \mid \alpha^*$

node expressions:
$\phi ::= p \mid \top \mid \neg\phi \mid \phi \wedge \psi \mid \langle\alpha\rangle \mid X \mid \mu X.\phi$

where X is a state variable that occurs only positively in ϕ (i.e., under an even number of negations). Note the addition of the last two clauses in the definition of the node expressions. We could have included the *loop* operator as well, but, as we will see, doing so would not increase the expressive power of the language.

The semantics of μRegular XPath-expressions is given relative to an assignment g mapping the set variables to sets of nodes in the

tree. All clauses in Figure 1 are relativized to such assignments, and node expressions of the form X or $\mu X.\phi$ are given the following semantics:

$$\begin{array}{lcl} [\![X]\!]^{M,g} & = & g(X) \\ [\![\mu X.\phi]\!]^{M,g} & = & \bigcap\{S \subseteq N \mid [\![\phi]\!]^{M,g[X \mapsto S]} \subseteq S\} \end{array}$$

A μRegular XPath expression is *closed* if all set variables occuring in it are bound by μ-operators. We restrict attention to closed expressions. Consult [12] for more details on the μ-operator.

In spite of its complicated looking definition, μRegular XPath is a very well behaved language. In fact, it is expressively complete for MSO.

THEOREM 16. *μRegular XPath and MSO have the same expressive power, both in terms of definable sets and in terms of definable binary relations.*

PROOF. The expressive completeness with respect to sets (i.e, the every $\phi(x) \in$ MSO is equivalent to a μRegular XPath node expression) is proved by Barceló and Libkin [8], and, in fact, already holds for the $*$-free fragment. Engelfriet and Bloem [10] showed that all MSO formulas with two free variables can be translated to a Regular XPath path expression with MSO tests (i.e., allowing arbitrary $\phi(x) \in$ MSO to be used as a tests). Together, these two results imply that every MSO formula with two free variables is equivalent to a μRegular XPath path expression. □

COROLLARY 17. *μRegular XPath is closed under path intersection and complementation.*

Note that these results depend on the presence of the Kleene star. Although every MSO formula in one free variable is equivalent to a $*$-free node expression of μRegular XPath, even the simple path expression $(\downarrow / \downarrow)^*$ cannot be written in μRegular XPath without using $*$ (the fixed point operator can only occur *within tests*).

The high expressive power of μRegular XPath comes at a price. Expressions involving the fixed point operator μ are notoriously difficult to grasp. Furthermore, evaluating μRegular XPath queries is likely to be much harder than evaluating Regular XPath queries. More precisely, while Regular XPath$^{\approx}$ queries can be evaluated in polynomial time (as discussed in the previous section), the best known query evaluation algorithms for μRegular XPath are in NP$\cap co$-NP [18]. Concerning the *query containment* problem for μRegular XPath, we have the following:

THEOREM 18. *The query containment problem for μRegular XPath is ExpTime-complete.*

PROOF. The lower bound follows from the ExpTime-hardness of Core XPath query containment [30]. As for the upper bound, note that a μRegular XPath path expression α is contained in a μRegular XPath path expression β iff the node expression $\langle\alpha[p]\rangle \wedge \neg\langle\beta[p]\rangle$ is unsatisfiable, where p is a proposition letter not occuring in α or β [34]. Hence, it suffices to prove ExpTime-membership of the *satisfiability* problem for μRegular XPath node expressions. The latter follows from results of Vardi [43], in the case of binary trees. The general case follows using the encoding discussed in the full version of this paper. □

This result is quite robust. On the one hand, since the validity of a DTD can be expressed inside Regular XPath [30], the containment problem for μRegular XPath expressions *relative to a DTD* is still decidable in ExpTime. On the other hand, the ExpTime-*hardness* holds already for Core XPath path expressions using only the `child` and `descendant` axes [22].

4.3 Positive transitive closure

Engelfriet and Hoogeboom [19] recently showed that, if a formula $\phi(x,y) \in$ FO* has only positive occurences of $*$ (i.e., under an even number of negation signs), there is a a non-deterministic single head tree walking automaton with pebbles (*pebble tree walking automaton* for short) that computes the same relation.[2] Tree walking automata offer an attractive model of computation, and have been applied in a variety of ways in the context of XML [36].

This raises the question which Regular XPath$^{\approx}$ expressions have an FO* translation satisfying this property. The natural conjecture, here, turns out to be the correct one: let us call an FO* formula *$*$-positive* if every occurence of the transitive closure operator is under the scope of an even number of negation signs, and likewise for Regular XPath$^{\approx}$ expressions.

THEOREM 19. *The $*$-positive fragment of* Regular XPath$^{\approx}$ *has the same expressive power as the $*$-positive fragment of* FO**, both in terms of definable sets and in terms of definable binary relations.*

PROOF. The translation TR given in Figure 3 shows that every $*$-positive Regular XPath$^{\approx}$-expression has a $*$-positive FO*-translation. As for the other direction, the reader may verify that, in the proof of Theorem 1, the polarity of occurences of $*$ is preserved. The only place where this is not immediately clear is Lemma 6, and in the full version of this paper, we provide a polarity preserving version of this lemma. □

As immediate corollaries, we obtain

COROLLARY 20. *Every $*$-positive* Regular XPath$^{\approx}$ *path expression is computed by a pebble tree walking automaton.*

COROLLARY 21. *The $*$-positive fragment of* Regular XPath$^{\approx}$ *is closed under path intersection.*

Also, using Theorem 19 we can show that the $*$-positive fragment of Regular XPath$^{\approx}$ can express all of Conditional XPath.

THEOREM 22. *Every* Conditional XPath *path expression is equivalent to a $*$-positive* Regular XPath$^{\approx}$ *path expression. Likewise for node expressions. These inclusions are strict.*

PROOF. Conditional XPath is known to be expressively complete for FO over a vocabulary consisting of $R_{\downarrow^*}$, $R_{\rightarrow^*}$, and a unary predicate for each XML tag [31]. Hence, by Theorem 19 it suffices to show that every formula $\phi(x,y)$ of this first-order language is equivalent to a $*$-positive FO* formula $\phi'(x,y)$. This is quite easy to see: replace all positive occurences of $R_{\downarrow^*}(u,v)$ in ϕ by $R^*_{\downarrow}(u,v)$, and replace occurences of $R_{\downarrow^*}(u,v)$ in negative position by

$$\neg\Big(R^+_{\downarrow}(v,u) \vee \exists ab\Big(R^*_{\downarrow}(a,u) \wedge R^*_{\downarrow}(b,v) \wedge (R^+_{\rightarrow}(a,b) \vee R^+_{\rightarrow}(b,a))\Big)\Big)$$

(where $R^+_{\downarrow}$ and $R^+_{\rightarrow}$ are the usual shorthands). Observe how the extra negation sign ensures that the transitive closure operator occurs under an even number of negations. Occurences of $R_{\rightarrow^*}$ can be replaced in a similar fashion. The resulting FO* is equivalent to ϕ and $*$-positive.

The inclusion is strict: even the simple $*$-positive path expression $(\downarrow / \downarrow)^*$ cannot be expressed in Conditional XPath [30], nor can the node expression $\langle(\downarrow / \downarrow)^*\rangle$. □

[2] Engelfriet and Hoogeboom [19] in fact prove something stronger: they show that pebble tree walking automata can compute exactly those relations that are definable in FO$+posTC_1$, first-order logic extended with the positive unary transitive closure operator.

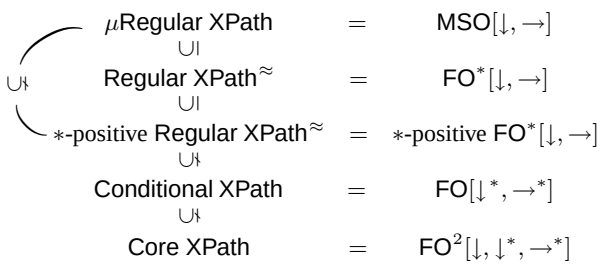


Figure 7: A hierarchy of XPath fragments

5. CONCLUSION

Given the prominent role of regular path expressions in the literature on semi-structured data [1], it is very natural to consider extending XPath with a transitive closure operator. As we discussed in the introduction, such extensions have indeed been proposed and implemented. In this paper, we have investigated the effect this has on the *expressive power* of the language. We studied the navigational fragment of XPath 1.0 extended with transitive closure, Regular XPath$^{\approx}$, and found out that it is as expressive as FO*. We also considered two other languages, the $*$-positive fragment of Regular XPath$^{\approx}$, and μRegular XPath, and we characterized them in terms of the $*$-positive fragment of FO*, and MSO, respectively. Our results, together with the previously known results on Core XPath and Conditional XPath, are summarized in Figure 7.[3] As indicated in the figure, it has recently been shown that the $*$-positive fragment of FO* is strictly contained in MSO [11]. However, we still do not know whether FO* is strictly contained in MSO, or the $*$-positive fragment of FO* is strictly contained in FO* (or both).

We conclude by listing some directions for future research.

CHALLENGE 23. *Show that '$loop$' (or, equivalently, path equality) contributes to the expressive power of* Regular XPath$^{\approx}$.

Showing this is equivalent to showing that Regular XPath$^{\approx}$ without $loop$ is not closed under path intersection (recall that $loop(\alpha)$ is equivalent to $\langle \alpha \cap . \rangle$). In Section 4.1, we gave a concrete example of a Regular XPath$^{\approx}$ node expression that we believe cannot be expressed without the use of $loop$.

CHALLENGE 24. *Determine the complexity of the query containment problem for* Regular XPath$^{\approx}$.

We know that the query containment problem for Regular XPath$^{\approx}$ without $loop$ is ExpTime-complete (cf. Theorem 18). However, we do not have any upper bound on the complexity of the query containment problem even for Core XPath extended with $loop$.

CHALLENGE 25. *Find effient algorithms for computing the transitive closure of XPath path expressions.*

Saxon and Xalan provide naive implementations of the transitive closure operator. Within the relational database literature, transitive closure has been studied quite intensively [5, 16], and an impressive number of algorithms have been developed for computing the transitive closure of a relation (cf. [6, 27] and references therein). The fact that an algorithm for computing transitive closure was patented [7] is illustrative. The relative performance of these algorithms, when applied to XPath expressions, remains to be investigated.

[3]The characterization of Core XPath in terms of FO^2 only holds with respect to *definable set of nodes*, not *definable binary relations*. Even the simple path expression $\downarrow / \downarrow$ is not definable by means of a formula $\phi(x, y) \in$ FO^2.

6. REFERENCES

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 2000.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison-Wesley, 1995.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The LOREL query language for semistructured data. *Journal on Digital Libraries*, 1(1):68–88, 1997.

[4] L. Afanasiev, P. Blackburn, I. Dimitriou, B. Gaiffe, E. Goris, M. Marx, and M. de Rijke. PDL on ordered trees. *Journal of Applied Non-Classical Logics*, 15(2):115–135, 2005.

[5] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, 1988.

[6] R. Agrawal, S. Dar, and H. V. Jagadish. Direct transitive closure algorithms: design and performance evaluation. *ACM Transactions on Database Systems*, 15(3):427–458, 1990.

[7] R. Agrawal and H. V. Jagadish. Method for computing transitive closure. United States Patent 4930072. Filed August 31, 1987; Published May 29, 1990. URL: `http://www.freepatentsonline.com/4930072.html`.

[8] P. Barceló and L. Libkin. Temporal logics over unranked trees. In P. Panangaden, editor, *Proceedings of LICS 2005*, pages 31–40. IEEE Computer Society Press, 2005.

[9] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. *Theoretical Computer Science*, 336(1):3–31, 2005.

[10] R. Bloem and J. Engelfriet. Characterization of properties and relations defined in monadic second order logic on the nodes of trees. Technical Report 97–03, Leiden University, August 1997.

[11] M. Bojańczyk, M. Samuelides, T. Schwentick, and L. Segoufin. On the expressive power of pebble automata. Manuscript.

[12] J. Bradfield and C. Stirling. PDL and the modal μ-calculus. In *Handbook of Modal Logic*. Elsevier North-Holland, 2006. To appear.

[13] A. Brüggemann-Klein and D. Wood. Caterpillars, context, tree automata and tree pattern matching. In *Proceedings of DALT 1999*, pages 270–285. World Scientific, 1999.

[14] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of SIGMOD 1996*, pages 505–516. ACM Press, 1996.

[15] E. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Database Systems*, pages 33–64. Prentice-Hall, 1972.

[16] S. Dar and R. Agrawal. Extending SQL with generalized transitive closure. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):799–812, 1993.

[17] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath fragments. In *Proceedings of KRDB 2001*, volume 45 of *CEUR Workshop Series*. CEUR-WS.org, 2001.

[18] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of μ-calculus. In *Proceedings of CAV 1993*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 1993.

[19] J. Engelfriet and H. J. Hoogeboom. Automata with nested pebbles capture first-order logic with transitive closure. Tech. Report LIACS 2005-02, Leiden University, April 2005.

[20] EXSLT. Available from `http://www.exslt.org/`, 2005. As viewed on November 28, 2005.

[21] W. Fan, J. X. Yu, H. Lu, J. Lu, and R. Rastogi. Query translation from XPath to SQL in the presence of recursive DTDs. In *Proceedings of VLDB 2005*, pages 337–348. VLDB Endowment, 2005.

[22] M. J. Fisher and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[23] E. Goris and M. Marx. Looping caterpillars. In *Proceedings of LICS 2005*, pages 51–60. IEEE Computer Society, 2005.

[24] G. Gottlob and C. Koch. Monadic queries over tree-structured data. In *Proceedings of LICS 2002*, pages 189–202. IEEE Computer Society, 2002.

[25] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of PODS 2003*, pages 179–190. ACM Press, 2003.

[26] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive queries over trees. In *Proceedings of PODS 2004*, pages 189–200. ACM Press, 2004.

[27] V. Hirvisalo, E. Nuutila, and E. Soisalon-Soininen. Transitive closure algorithm MEMTC and its performance analysis. *Discrete Applied Mathematics*, 110(1):77–84, 2001.

[28] M. Kay. *XPath Programmer's reference*. Wrox, 2004.

[29] M. Lange. Model checking propositional dynamic logic with all extras. *Journal of Applied Logic*, 4(1):39–49, 2005.

[30] M. Marx. XPath with conditional axis relations. In *Proceedings of EDBT 2004*, volume 2992 of *Lecture Notes in Computer Science*. Springer, 2004.

[31] M. Marx. Conditional XPath. *ACM Transactions on Database Systems*, 30(4):929–959, 2005.

[32] M. Marx and M. de Rijke. Semantic characterizations of navigational XPath. *ACM SIGMOD Record*, 34(2):41–46, 2005.

[33] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the world wide web. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[34] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proceedings of PODS 2002*, pages 65–76. ACM Press, 2002.

[35] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.

[36] F. Neven. Automata, logic, and XML. In *Proceedings of CSL 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2002.

[37] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proceedings of PODS 2000*, pages 145–156. ACM Press, 2000.

[38] K. C. Ng. *Relation Algebras with Transitive Closure*. PhD thesis, University of California, Berkeley, 1984.

[39] K. C. Ng and A. Tarski. Relation algebras with transitive closure. *Notices of the American Mathematical Society*, 24:A29–A30, 1977.

[40] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proceedings of XMLDM 2002*, pages 109–127. Springer Verlag, 2002.

[41] SAXON: The XSLT and XQuery processor. version 6.5.5. Available from `http://saxon.sourceforge.net/`, 2005. As viewed on November 28, 2005.

[42] J. E. Simpson. *XPath and XPointer*. O'Reilly Media, Inc., 2002.

[43] M. Vardi. Reasoning about the past with two-way automata. In *Proceedings of ICALP 1998*, volume 1443 of *Lecture Notes in Computer Science*, pages 628–641. Springer, 1998.

[44] Xalan-java version 2.7.0. Available from `http://xml.apache.org/xalan-j/`, 2005. As viewed on November 28, 2005.