Decidable XPath Fragments in the Real World

David Baelde, Anthony Lick, Sylvain Schmitz

Decidable XPath Fragments in the Real World

David Baelde
Anthony Lick
Sylvain Schmitz
LSV, ENS Paris-Saclay & CNRS & Inria, Université Paris-Saclay, France

ABSTRACT

XPath is arguably the most popular query language for selecting elements in XML documents. Besides query evaluation, query satisfiability and containment are the main computational problems for XPath; they are useful, for instance, to detect dead code or validate query optimisations. These problems are undecidable in general, but several fragments have been identified over time for which satisfiability (or query containment) is decidable: CoreXPath 1.0 and 2.0 without so-called data joins, fragments with data joins but limited navigation, etc. However, these fragments are often given in a simplified syntax, and sometimes wrt. a simplified XPath semantics. Moreover, they have been studied mostly with theoretical motivations, with little consideration for the practically relevant features of XPath.

To investigate the practical impact of these theoretical fragments, we design a benchmark compiling thousands of real-world XPath queries extracted from open-source projects. These queries are then matched against syntactic fragments from the literature. We investigate how to extend these fragments with seldom-considered features such as free variables, data tests, data joins, and the `last()` and `id()` functions, for which we provide both undecidability and decidability results. We analyse the coverage of the original and extended fragments, and further provide a glimpse at which other practically-motivated features might be worth investigating in the future.

CCS CONCEPTS

• **Information systems** → **XPath**; XQuery; • **Theory of computation** → *Logic and verification*;

1 INTRODUCTION

The XPath language [41] is arguably the most popular querying language for selecting elements in XML documents. It is embedded in the XML processing languages XSLT [19] and XQuery [42], and widely used in general-purpose languages like Java or C# through third-party libraries. It combines the ability to navigate the XML tree—which finds its roots in modal logic [3]—with that of comparing data values found in several, distantly related attributes.

Alongside the *evaluation* of the set of elements selected in a document [e.g. 4, 10, 13, 30], the main computational problem associated with a query is the *satisfiability* problem: given an XPath query, and optionally a schema for the type of XML documents under consideration, does there exist at least one XML document on which this query would select some node? This abstract question actually allows to answer several questions on the reliability and performance of a query; to wit:

Usefulness: is the data query useful at all, i.e. will it select some parts of a well-formed document? The question is far from trivial in an environment where the data can be retrieved from external sources, prone to changes in their structure. This allows simple optimisations in batch processes, with significant savings [31].

Optimisation: can a query be replaced by another, simpler query? This is a classical question in query rewriting in database theory. This type of optimisation has for instance been applied to XPath queries without data joins by Genevès and Vion-Dury [28], with an appreciable impact on performance.

Unfortunately, the satisfiability problem is in general undecidable. Even CoreXPath 1.0, a simplified syntax based on the navigational fragment of XPath 1.0, is untractable, more exactly `EXP`-complete already when only using the `child` axis [3, 9, 40]. Thus with the exception of the work of Genevès et al. [27] on CoreXPath 1.0, most of the literature on the topic is of a theoretical nature [e.g. 2, 5, 12, 20–22, 25, 26, 34], and focuses on decidability and complexity questions in variants of CoreXPath 1.0 that allow limited forms of data joins.

While these results are technically impressive, it is not immediately clear how much is gained in practice by handling data joins. Indeed, data joins are not the sole source of difficulty in XPath: many real-life XPath queries perform calls to a standard library of functions [18]—including arithmetic and string-manipulating functions—that also lead to an undecidable satisfiability.

Also, XPath 1.0 dates back to 1999; the more recent versions 2.0 and 3.0 feature path intersections, `for` loops, etc. [see 45]. As it evolves in pace with XQuery, XPath includes more and more general programming constructs, and is arguably not just a domain-specific language for path navigation—quite tellingly, in our benchmark, we found that only half of the queries use navigation.

In this paper, we evaluate the practical applicability of XPath fragments proposed in the theoretical literature from a basic, syntactic perspective: how many queries are captured by these fragments? The first step to this end is the compilation of a benchmark of 21,141 real-world XPath queries. The queries are extracted from open-source projects that rely heavily on XSLT or XQuery. As described in Sec. 2, the benchmark offers for each XPath query an XML syntax tree representation in XQueryX [17]. The tools and the resulting benchmark are available from http://git.lsv.fr/schmitz/xpparser/ under open source licenses.

Our aim is then to check the queries in our benchmark against the syntax allowed in theoretical works on XPath satisfiability, namely by writing Relax NG specifications for PositiveXPath [26,

32], CoreXPath 1.0 [29], CoreXPath 2.0 [45], fragments of DataXPath [20, 21, 25], and fragments of XPath that can be interpreted in EMSO^2 [12] or using non-mixing MSO constraints [15] (see Sec. 4).

Naturally, the syntax defined in these works is simplified and was never meant to be used directly against concrete XPath inputs, while we need a concrete syntax for each one of these fragments in order to implement it in Relax NG. This leads to the interesting question of which XPath features can be 'reasonably' handled in these fragments without losing a decidable satisfiability nor hampering its complexity (see Sec. 5.1). In turn, answering this type of question requires the definition of a palatable semantics for a substantial subset of XPath 3.0, which we provide in Sec. 3.

As could be expected, the coverage of the naive implementations of the syntactic fragments from the literature is not very good (Sec. 4.2), and one should extend them whenever reasonable. In Sec. 5, we propose six extensions of the original fragments and evaluate their coverage on the benchmark. These extensions are root navigation, free variables, data tests against constants, positive data joins, and restricted calls to the functions `last()` and `id()`. Just as interestingly, we exhibit several cases where these extensions *cannot* be handled.

We analyse our experimental results in Sec. 6, notably concluding that higher coverage is obtained through basic extensions than by using complex academic fragments. We also identify increased function support as a promising direction for improved practical satisfiability checking, with an especially high potential for XPath queries from XSLT sources. We conclude in Sec. 7.

2 A REAL-WORLD BENCHMARK

We explain here the technical aspects of the construction of a benchmark of 21,141 queries: the parser we developed to this end (Sec. 2.1), the sources we employed (Sec. 2.2), and the way we processed the benchmark to check whether a given query belongs to a syntactic XPath fragment (Sec. 2.3). We finish the section by mentioning the limitations of the current benchmark.

2.1 Parser

We have modified the W3C parser for XQuery 3.0,[1] which is (almost) a superset of XPath 3.0, so that we can also use it for XPath queries extracted from XSLT documents. This parser uses a grammar automatically extracted from the language specification, so we are confident in its results. Our implementation further

(1) extracts XPath queries from XQuery files, by selecting 'maximal XPath subtrees' in the abstract syntax tree of the XQuery document, and
(2) outputs syntax trees in the standard XML format XQueryX [17]; this is what we process to determine to which XPath fragments each query belongs.

See App. B for an example XQuery document and the corresponding parser output.

2.2 Sources

Our choice of sources for the benchmark relied on searches through open source GitHub projects containing XSLT or XQuery files, selecting the most popular projects from which we could extract at least 50 queries. We also added one large project not hosted on GitHub, namely DocBook XSL.

The XSLT projects aim to translate enriched text documents between different formats. The XQuery projects we include are most often libraries. The detailed composition is presented in Tab. 6 in the appendices. We make no formal claim about the coverage of the benchmark, but rather see it as a first release, to be enriched later as part of a community-driven effort.

[1] https://www.w3.org/2013/01/qt-applets/

Table 1: The benchmark's composition.

Sources	Queries	Coverage			
		XPath 1.0	XPath 2.0	XPath 3.0	XPath 3.0 std
XSLT	14,675	98.4%	100.0%	100.0%	91.3%
XQuery	6,466	76.1%	87.4%	99.8%	46.7%
Total	21,141	91.6%	96.1%	99.9%	77.7%

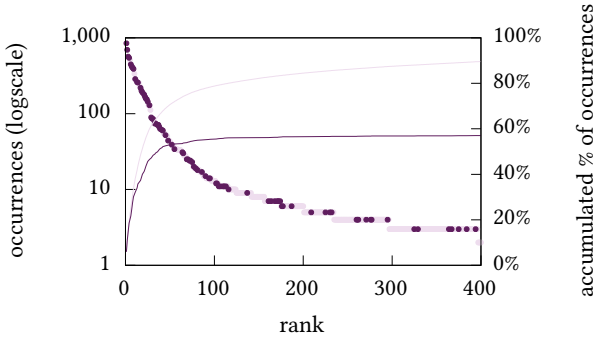

Figure 1: Occurrences of function calls.

2.3 Properties of the Benchmark

Standard Coverage. We exploit this benchmark by validating the syntax trees in XQueryX format against Relax NG [16] specifications. Table 1 presents the number of queries that fall within the scope of the three major revisions of the XPath standard. We can observe that the queries extracted from XSLT files are nearly all XPath 1.0 queries, which contrasts with queries extracted from XQuery sources, which rely more often on advanced XPath features from XPath 2.0 and XPath 3.0.[2]

Note that the coverage of XPath 1.0, 2.0, and 3.0 given in Tab. 1 does not restrict function calls to the standard library [18]. The last column 'XPath 3.0 std' shows the coverage of XPath 3.0 when restricted to standard functions. We see here an essential limitation of analysing XPath queries in isolation, without support for non-standard functions, and in particular for user-defined functions: more than half of the queries extracted from XQuery documents are beyond the scope of our analyses.

Functions. We show in Fig. 1 the number of occurrences for each of the 400 most frequently occurring functions (among 1,600) and the associated accumulated percentage of the total number of function calls. Darker dots correspond to standard XPath functions,

[2] The output XQueryX representations of a handful of queries do not validate against XPath 3.0, due to out-of-bounds constant numerals; this is a very marginal effect.

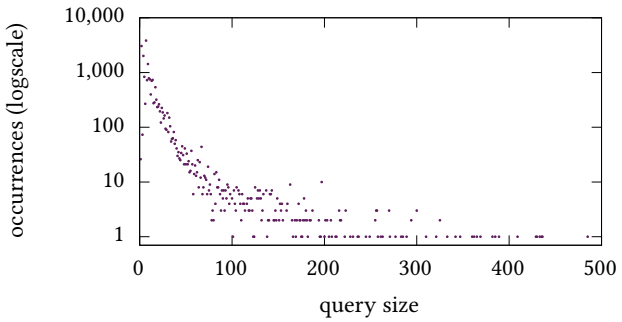

Figure 2: Distribution of query sizes.

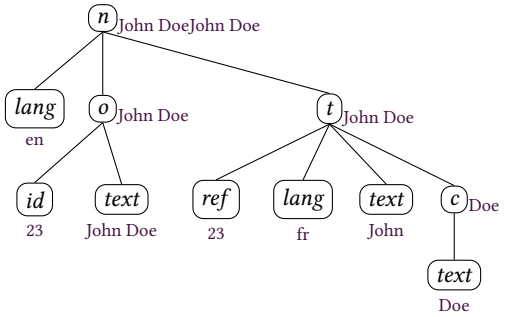

Figure 3: A data tree; labels from Σ are shown in the nodes, data from $\mathbb{D}$ in violet next to them.

and the darker line corresponds to the accumulated percentage achieved by these standard functions. Arithmetic operations do not figure here as they are classified syntactically as *operators* in XPath. They occur more than the tenth most frequent function.

The standard functions only represent 57.23% of the function calls in the benchmark. This is mostly due to queries from XQuery sources, which routinely use functions defined in the surrounding XQuery programs: when restricting to these sources, standard XPath functions represent 42.93% of the function calls. By contrast, when restricting to XSLT sources, we find only 210 functions and standard XPath functions represent 76.32% of the calls. Moreover, the 16 functions with more than 100 occurrences each all belong to the XSLT or XPath standard, and account for 78.35% of the occurrences of function calls. In the XSLT sources, there are 4,650 queries (31.69%) performing at least one function call, roughly as many as the 4,556 queries (70.46%) found in the XQuery sources.

Size. Figure 2 shows the distribution of query sizes, defined as the size of their syntax trees, measured as the number of nodes in the tree. As might be expected, most of the queries are of modest size and a majority of the queries have size at most 8, but there are nevertheless 256 queries of size 100 or more.

2.4 Limitations

The benchmark is made of uncurated data, thus no distinction is made between tiny XPath queries and more interesting ones. For instance, only 9,852 of the queries in the benchmark use at least one axis step.

Also, as seen in Tab. 1, the numbers of queries from the various sources are not balanced, which means that results on the whole benchmark might not be very telling, and that one should distinguish the XSLT sources from the XQuery ones.

Finally, the benchmark was compiled specifically for investigating the coverage of syntactic fragments of XPath. It is currently not really suitable for other ends:

- **XPath satisfiability:** in both XSLT and XQuery files, no schema information on the XML to be processed is available. Furthermore, it seems likely that most queries are satisfiable—quite possibly all of them.
- **XPath evaluation:** similarly, the benchmark does not provide examples of input XML documents on which the XSLT or XQuery should be evaluated.

These limitations could probably be lifted by adding manual annotations.

3 XPATH 3.0

The XPath 3.0 specification is arguably too complex to be reasoned about directly. We work instead with a well-defined sub-language, designed to capture accurately the constructions we witnessed in the benchmark. In order to be compatible with the semantics in the XPath literature, we provide a semantics on data trees, but in Sec. 3.6 we show how to capture the actual XPath semantics on XML documents.

3.1 Data Trees

Our models are an abstraction of XML DOM trees called *data trees*, which are finite trees where each node carries both a *label* from a finite alphabet Σ and a *datum* from an infinite countable domain $\mathbb{D}$ equipped with an order $<$.

Formally, a data tree is a finite rooted ordered unranked tree with labels in $\Sigma \times \mathbb{D}$: it is a pair $t = (\ell, \delta)$ of functions $\ell\colon N \to \Sigma$, $\delta\colon N \to \mathbb{D}$ with a common non-empty finite set of nodes $N \subseteq \mathbb{N}^*$ as domain; N must be prefix-closed (if $p \cdot i \in N$ for some $p \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then $p \in N$) and predecessor-closed (if $p \cdot (i+1) \in N$ for some $p \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then $p \cdot i \in N$); in particular it contains a root node ε. Nodes in N, being finite sequences of natural numbers, are totally ordered by the lexicographic ordering, which is known in this context as the *document order* and denoted by $\ll$. Figure 3 displays an example of a data tree.

When working with first-order or monadic second-order logic, a data tree is a relational structure $(N, \downarrow, \rightarrow, \sim, (P_a)_{a\in\Sigma}, (P_d)_{d\in\mathbb{D}})$ where

$$\rightarrow \stackrel{\text{def}}{=} \{(p\cdot i, p\cdot(i+1)) \in N^2 \mid p \in \mathbb{N}^*, i \in \mathbb{N}\}\,,$$
$$\downarrow \stackrel{\text{def}}{=} \{(p, p\cdot i) \in N^2 \mid p \in \mathbb{N}^*, i \in \mathbb{N}\}\,, \quad P_a \stackrel{\text{def}}{=} \{p \in N \mid \ell(p) = a\},$$
$$\sim \stackrel{\text{def}}{=} \{(p, p') \in N^2 \mid \delta(p) = \delta(p')\}\,, \qquad P_d \stackrel{\text{def}}{=} \{p \in N \mid \delta(p) = d\}$$

denote respectively the child relation, the next-sibling relation, data equivalence, and the labelling and data predicates.

3.2 Syntax

While our implementation works with concrete syntax (see e.g. the example in App. B), for the sake of readability we use an abstract syntax throughout the paper. It is nevertheless fully compatible

$$[\![\texttt{self}]\!]_{\mathsf{A}} \stackrel{\text{def}}{=} \{(p,p) \mid p \in N^2\}$$

$$[\![\texttt{child}]\!]_{\mathsf{A}} \stackrel{\text{def}}{=} \downarrow \qquad [\![\texttt{parent}]\!]_{\mathsf{A}} \stackrel{\text{def}}{=} \downarrow^{-1}$$

$$[\![\texttt{following-sibling}]\!]_{\mathsf{A}} \stackrel{\text{def}}{=} \rightarrow^{+} \qquad [\![\texttt{ancestor}]\!]_{\mathsf{A}} \stackrel{\text{def}}{=} (\downarrow^{-1})^{+}$$

$$[\![\texttt{preceding-sibling}]\!]_{\mathsf{A}} \stackrel{\text{def}}{=} (\rightarrow^{-1})^{+} \qquad [\![\texttt{descendant}]\!]_{\mathsf{A}} \stackrel{\text{def}}{=} \downarrow^{+}$$

Figure 4: The semantics of XPath axes.

with XPath 3.0: all the examples in this paper are written in actual XPath.

Let $\mathcal{X}$ be a countable infinite set of variables, and $\mathcal{F}$ be a ranked alphabet of function names; we denote by $\mathcal{F}_n$ its subset of symbols with arity n. As usual, our language has multiple sorts: *axes* denote directions in the data tree, with abstract syntax

$$\alpha ::= \texttt{self} \mid \texttt{child} \mid \texttt{descendant} \mid \texttt{following-sibling} \mid \texttt{parent} \mid \texttt{ancestor} \mid \texttt{preceding-sibling}$$

path expressions describe binary relations between the nodes in a data tree, with abstract syntax

$$\pi ::= \alpha{::}* \mid /\pi \mid \pi/\pi \mid \pi[\varphi] \mid \pi \texttt{ union } \pi \mid f(\pi_1,\dots,\pi_n) \mid \texttt{\$x} \mid \texttt{let \$x :=} \pi \texttt{ return } \pi' \mid \texttt{for \$x in } \pi \texttt{ return } \pi$$

where $\texttt{\$x}$ ranges over $\mathcal{X}$, n over $\mathbb{N}$, and f over $\mathcal{F}_n$, while *node expressions* describe sets of nodes, with abstract syntax

$$\varphi ::= \pi \mid a \mid \texttt{false}() \mid \texttt{not}(\varphi) \mid \varphi \texttt{ or } \varphi \mid \pi \texttt{ is } \pi \mid \pi \vartriangle \pi \mid \pi \vartriangle^{+} d$$

where a ranges over Σ, d over $\mathbb{D}$, $\vartriangle$ over $\{\texttt{eq}, \texttt{ne}\}$ and $\vartriangle^{+}$ over $\{\texttt{eq}, \texttt{ne}, \texttt{le}, \texttt{lt}, \texttt{ge}, \texttt{gt}\}$. Note that only `eq` and `ne` are allowed when comparing paths, while the ordered structure of $\mathbb{D}$ is only available when comparing a path and a data constant: none of the fragments we consider is known to allow the richer path comparisons.

3.3 Data Tree Semantics

For a fixed data tree of domain N, we give in figures 4 and 5 the semantics of axes $[\![\alpha]\!]_{\mathsf{A}}$, path and node expressions $[\![\pi]\!]^{\nu}_{\mathsf{P}}$ and $[\![\varphi]\!]^{\nu}_{\mathsf{N}}$. The semantics is relative to a current variable valuation $\nu\colon \mathcal{X} \to 2^N$. The semantics of node expressions are sets of nodes of N, while those of axes and path expression are sets of pairs of nodes.

In order to interpret functions, we assume that each function symbol f from $\mathcal{F}_n$ comes with a semantics $[\![f]\!]_{\mathcal{F}}\colon (2^N)^n \to 2^N$. For instance, `false()` and `not()` are technically XPath functions, with semantics $[\![\texttt{false}]\!]_{\mathcal{F}} \stackrel{\text{def}}{=} \emptyset$ and $[\![\texttt{not}]\!]_{\mathcal{F}}(S) \stackrel{\text{def}}{=} N \setminus S$ for all $S \subseteq N$. Likewise, we interpret each comparison operator $\vartriangle^{+}$ as a data relation $\overset{\vartriangle^{+}}{\sim} \subseteq \mathbb{D}\times\mathbb{D}$: $\overset{\text{eq}}{\sim}$ is the equality $=$ over $\mathbb{D}$, and $\overset{\text{ne}}{\sim}$ the disequality $\neq$, $\overset{\text{lt}}{\sim}$ the strict order $<$, etc. For binary relations R, R', we employ relational compositions $R \fatsemi R' \stackrel{\text{def}}{=} \{(p,p'') \mid \exists p'.(p,p') \in R \wedge (p',p'') \in R'\}$, transitive closures R^{+}, converses $R^{-1} \stackrel{\text{def}}{=} \{(p',p) \mid (p,p') \in R\}$, and images $R(p) \stackrel{\text{def}}{=} \{p' \mid (p,p') \in R\}$.

Beware that the semantics of a path expression π changes when seen as a node expression. In particular, a variable $\texttt{\$x}$ is a *path expression*, and when seen as a node formula $[\![\texttt{\$x}]\!]^{\nu}_{\mathsf{N}} = N$ unless $\nu(\texttt{\$x}) = \emptyset$. XPath provides two quantifiers: `for` expressions bind singleton node sets, while `let` expressions bind node sets.

Example 3.1. Evaluating `for $x in child::*[`o `or` t`] return $x[self::* eq $x/child::*]` at the root of the data tree in Fig. 3 binds $x to each of the two children nodes labelled o or t in succession, and returns the o node. Evaluating `let $x := child::*[`o `or` t`] return $x[self::* eq $x/child::*]` at the root binds $x to the set containing both o and t, and returns both of them.

Our semantics is in line with the ones found in the literature. However, we note that it slightly differs from the actual XPath semantics. In particular, we only account for pure functions acting on paths, while functions from XPath's large standard library [18] may be polymorphic and have side-effects—two features that are anyway out of the reach of the current decidable fragments. Also, variables in XPath are bound to ordered collections of nodes and data values, while we only consider sets of nodes. This simpler semantics is not restrictive for our purposes, as discussed further in Sec. 3.6.

3.4 The Satisfiability Problem

In this paper, we focus on the *satisfiability problem*: given a node expression φ, does there exist a data tree t such that $t \models \varphi$? As path expressions are also node expressions, this also captures the satisfiability of path expressions. In presence of a DTD, the data tree t should additionally belong to the DTD's language.

Remark 3.2. A related problem is *query containment*: for node expressions φ and φ', we say that φ is *contained* in φ' if, for all data trees and all variable valuations ν, $[\![\varphi]\!]^{\nu}_{\mathsf{N}} \subseteq [\![\varphi']\!]^{\nu}_{\mathsf{N}}$. This is equivalent to asking the unsatisfiability of φ `and not(`φ'`)`, so it reduces to the satisfiability problem when negation is allowed—which will not always be our case. The problem of *path containment* asks the same question for path expressions π and π', and is not captured by satisfiability. Furthermore, one might also consider variants of these problems where we ask instead whether for all data trees and variable valuations ν and ν', $[\![\varphi]\!]^{\nu}_{\mathsf{N}} \subseteq [\![\varphi']\!]^{\nu'}_{\mathsf{N}}$, which can lead to rather different results [40].

3.5 Syntactic Sugar

XPath comes with some handy syntactic sugar.

3.5.1 XPath 1.0 Sugar. Beside standard definitions like `true()` $\stackrel{\text{def}}{=}$ `not(false())` or φ `and` φ' $\stackrel{\text{def}}{=}$ `not(not(`φ`) or not(`φ'`))`, we may use `.` $\stackrel{\text{def}}{=}$ `self::*` for referring to the current point of focus, `..` $\stackrel{\text{def}}{=}$ `parent::*` for its parent, $\alpha{::}a \stackrel{\text{def}}{=} \alpha{::}*[a]$ for testing the label found after an axis step, and a single label a as a path formula for `child::`a.

The syntax also features four more axes: `descendant-or-self::*` is defined as `descendant::* union self::*` and `ancestor-or-self` is defined similarly. The axis `following::*` is defined as the path `ancestor-or-self::*/following-sibling::*/descendant-or-self::*`. The `preceding` axis is defined similarly. The shorthand $\pi//\pi'$ stands for π`/descendant-or-self::*/`π', and $//\pi$ is defined similarly. Conditional node expressions, while only available in XPath 2.0 and later, are also easily handled: `if (`φ`) then` φ' `else` φ'' $\stackrel{\text{def}}{=}$ (φ `and` φ') `or` (`not(`φ`) and` φ'').

3.5.2 XPath 2.0 Sugar. As shown by ten Cate and Lutz [45], path intersection and complementation (introduced in XPath 2.0) can be

$$\begin{aligned}
[\![\alpha :: *]\!]^\nu_{\mathsf{P}} &\stackrel{\text{def}}{=} [\![\alpha]\!]_{\mathsf{A}} & [\![\pi]\!]^\nu_{\mathsf{N}} &\stackrel{\text{def}}{=} \{p \in N \mid \exists p' \in [\![\pi]\!]^\nu_{\mathsf{P}}(p)\}\\
[\![/\pi]\!]^\nu_{\mathsf{P}} &\stackrel{\text{def}}{=} N \times [\![\pi]\!]^\nu_{\mathsf{P}}(\varepsilon) & [\![a]\!]^\nu_{\mathsf{N}} &\stackrel{\text{def}}{=} P_a\\
[\![\pi/\pi']\!]^\nu_{\mathsf{P}} &\stackrel{\text{def}}{=} [\![\pi]\!]^\nu_{\mathsf{P}} \fatsemi [\![\pi']\!]^\nu_{\mathsf{P}} & [\![\texttt{false()}]\!]^\nu_{\mathsf{N}} &\stackrel{\text{def}}{=} \emptyset\\
[\![\pi[\varphi]]\!]^\nu_{\mathsf{P}} &\stackrel{\text{def}}{=} [\![\pi]\!]^\nu_{\mathsf{P}} \cap (N \times [\![\varphi]\!]^\nu_{\mathsf{N}}) & [\![\texttt{not}(\varphi)]\!]^\nu_{\mathsf{N}} &\stackrel{\text{def}}{=} N \setminus [\![\varphi]\!]^\nu_{\mathsf{N}}\\
[\![\pi \texttt{ union } \pi']\!]^\nu_{\mathsf{P}} &\stackrel{\text{def}}{=} [\![\pi]\!]^\nu_{\mathsf{P}} \cup [\![\pi']\!]^\nu_{\mathsf{P}} & [\![\varphi \texttt{ or } \varphi']\!]^\nu_{\mathsf{N}} &\stackrel{\text{def}}{=} [\![\varphi]\!]^\nu_{\mathsf{N}} \cup [\![\varphi']\!]^\nu_{\mathsf{N}}\\
[\![f(\pi_1,\dots,\pi_n)]\!]^\nu_{\mathsf{P}} &\stackrel{\text{def}}{=} \{(p, [\![f]\!]_{\mathcal{F}}([\![\pi_1]\!]^\nu_{\mathsf{P}}(p),\dots,[\![\pi_n]\!]^\nu_{\mathsf{P}}(p)) \mid p \in N\} & [\![\pi \texttt{ is } \pi']\!]^\nu_{\mathsf{N}} &\stackrel{\text{def}}{=} \{p \in N \mid \exists p' \,.\, \{p'\} = [\![\pi]\!]^\nu_{\mathsf{P}}(p) = [\![\pi']\!]^\nu_{\mathsf{P}}(p)\}\\
[\![\$\mathsf{x}]\!]^\nu_{\mathsf{P}} &\stackrel{\text{def}}{=} N \times \nu(\$\mathsf{x}) & [\![\pi \mathbin{\triangle^+} d]\!]^\nu_{\mathsf{N}} &\stackrel{\text{def}}{=} \{p \in N \mid \exists p' \in [\![\pi]\!]^\nu_{\mathsf{P}}(p) \,.\, \delta(p') \stackrel{\triangle^*}{\sim} d\}\\
[\![\texttt{let } \$\mathsf{x} := \pi \texttt{ return } \pi']\!]^\nu_{\mathsf{P}} &\stackrel{\text{def}}{=} \{(p, [\![\pi']\!]^{\nu[\$\mathsf{x}\mapsto[\![\pi]\!]^\nu_{\mathsf{P}}(p)]}_{\mathsf{P}}(p)) \mid p \in N\} & [\![\pi \mathbin{\triangle} \pi']\!]^\nu_{\mathsf{N}} &\stackrel{\text{def}}{=} \{p \in N \mid \exists p', p'' \,.\, p' \in [\![\pi]\!]^\nu_{\mathsf{P}}(p)\\
[\![\texttt{for } \$\mathsf{x} \texttt{ in } \pi \texttt{ return } \pi']\!]^\nu_{\mathsf{P}} &\stackrel{\text{def}}{=} \{(p,p'') \in [\![\pi']\!]^{\nu[\$\mathsf{x}\mapsto\{p'\}]}_{\mathsf{P}} \mid p' \in [\![\pi]\!]^\nu_{\mathsf{P}}(p)\} & &\qquad \wedge p'' \in [\![\pi']\!]^\nu_{\mathsf{P}}(p) \wedge \delta(p') \stackrel{\triangle}{\sim} \delta(p'')\}
\end{aligned}$$

Figure 5: The semantics of XPath path expressions and node expressions.

expressed using `for` loops: π `intersect` π' is defined by

$$\texttt{for } \$\mathsf{x} \texttt{ in } \pi \texttt{ return for } \$\mathsf{y} \texttt{ in } \pi' \texttt{ return } \$\mathsf{x}[\$\mathsf{x} \texttt{ is } \$\mathsf{y}] \tag{1}$$

and π `except` π' by

$$\texttt{for } \$\mathsf{x} \texttt{ in } \pi \texttt{ return } .[\texttt{not}(\pi'[. \texttt{ is } \$\mathsf{x}])]/\$\mathsf{x} \tag{2}$$

Similarly, node quantification `some` $\$\mathsf{x}$ `in` π `satisfies` φ can be expressed using

$$\texttt{for } \$\mathsf{x} \texttt{ in } \pi \texttt{ return } .[\varphi] \tag{3}$$

and `every` $\$\mathsf{x}$ `in` π `satisfies` φ is defined dually. By first defining the non-standard $\mathit{future}(\varphi) \stackrel{\text{def}}{=} (\texttt{following::*} \texttt{ union } \texttt{descendant::*})[\varphi]$ and $\mathit{singleton}(\pi) \stackrel{\text{def}}{=} \pi \texttt{ and } \texttt{not}(\pi \texttt{ intersect } \pi/\mathit{future}(\texttt{true}()))$, then we can also express standard *node comparisons* $\pi \ll \pi'$ with

$$\begin{aligned}&\mathit{singleton}(\pi) \texttt{ and } \mathit{singleton}(\pi')\\ &\qquad\qquad \texttt{and } \pi' \texttt{ intersect } (\pi/\mathit{future}(\texttt{true}()))\end{aligned} \tag{4}$$

In XPath, a path expression can be forced to end on the last possible node for the document order. This often appears as a predicate $\pi[\texttt{last}()]$ or $\pi[\texttt{position}() = \texttt{last}()]$ that calls the nullary `last()` function [18]. However, this syntax cannot be handled with our simplified semantics—and is a bit problematic—, thus we shall only consider the one-argument version of `last()` [18], with semantics $[\![\texttt{last}]\!]_{\mathcal{F}}(S) \stackrel{\text{def}}{=} \max_{\ll}(S)$ for any $S \subseteq N$. Then $\texttt{last}(\pi)$ can be expressed in XPath 2.0 by

$$\pi \texttt{ except } (\pi/\texttt{ancestor::*} \texttt{ union } \texttt{preceding::*}) \tag{5}$$

Example 3.3. In the data tree of Fig. 3, when evaluated at the *c* node, the path expression `last(ancestor-or-self::*/child::`*lang*`)` returns the *lang* node with data value 'fr.'

We will discuss the `last()` function further in Sec. 5.2.5.

3.6 XML Semantics

The XPath data model [47] specifies that nodes can fall into several categories, with multiple types and accessors. In data trees, there are only nodes, and two accessors ℓ and δ. Nevertheless, a large part of the XPath data model can be handled.

3.6.1 Data Tree of an XML Document. Elements, attributes, text nodes, and comments can all be encoded using distinguished labels: we let $\Sigma = E \uplus A \uplus \{\mathit{text}, \mathit{comment}\}$ where E is the set of element labels and A of attribute labels.

We see all the values as belonging to $\mathbb{D}$. The data in $\mathbb{D}$ associated with attribute nodes, text nodes, and comment nodes is their string value; for an element node, it is the concatenation of the string values of all its element and text children. The following XML document corresponds to the data tree of Fig. 3:

```
<n lang="en"><o id="23">John Doe</o><t
 ref="23" lang="fr">John <c>Doe</c></t></n>
```

3.6.2 XML-Specific Syntax. When considering XML documents rather than data trees as models, some additional features of XPath become meaningful. We enrich the syntax with the axis

$$\alpha ::= \cdots \mid \texttt{attribute}$$

and five *node tests*

$$\begin{aligned}\tau &::= \texttt{attribute}() \mid \texttt{comment}() \mid \texttt{element}() \mid \texttt{text}()\\ \varphi &::= \cdots \mid \tau\\ \pi &::= \cdots \mid \alpha\texttt{::node}()\end{aligned}$$

meant to select exactly the appropriate node category, and new syntactic sugar: $\alpha::\tau \stackrel{\text{def}}{=} \alpha\texttt{::node}()[\tau]$ and $@a \stackrel{\text{def}}{=} \texttt{attribute::*}[a]$ for $a \in A$.

3.6.3 Interpretation into Data Trees. Given an XPath node expression φ with the XML semantics of [41], we interpret it as an XPath node expression $\ulcorner\varphi\urcorner \texttt{ and } \texttt{not}(\texttt{//.}[\mathit{notxml}])$ that uses the data tree semantics of Sec. 3.3.

The first conjunct $\ulcorner\varphi\urcorner$ is defined by induction on φ; the case of node tests τ is straightforward:

$$\ulcorner\texttt{attribute}()\urcorner \stackrel{\text{def}}{=} \texttt{or}_{a\in A}\, a \qquad \ulcorner\texttt{comment}()\urcorner \stackrel{\text{def}}{=} \mathit{comment}$$
$$\ulcorner\texttt{element}()\urcorner \stackrel{\text{def}}{=} \texttt{or}_{e\in E}\, e \qquad \ulcorner\texttt{text}()\urcorner \stackrel{\text{def}}{=} \mathit{text}$$

The semantics of atomic steps is modified to only visit element nodes, except when using the `attribute` axis or the `node()` test, and to forbid horizontal axes in attribute nodes: $\ulcorner\alpha\texttt{::node}()\urcorner \stackrel{\text{def}}{=} \alpha\texttt{::*}$, while $\ulcorner\alpha\texttt{::*}\urcorner$ is defined as $\texttt{child::*}[\texttt{or}_{a\in A}\, a]$ if $\alpha = \texttt{attribute}$, as $\alpha\texttt{::*}[\texttt{or}_{e\in E}\, e]$ if $\alpha = \texttt{parent}$ or $\alpha = \texttt{ancestor}$, and for all the other

axes as $.[\texttt{not}(\texttt{or}_{a\in A}\, a)]/\alpha::*[\texttt{or}_{e\in E}\, e]$. The remaining cases of the induction are by identity homomorphism.

The second conjunct $\texttt{not}(//.[\mathit{notxml}])$ ensures that the data tree is indeed the encoding of an XML document, by forbidding the node expression *notxml* everywhere in the tree. We define it by first ensuring that attribute, comments, and text nodes are leaves:

$$(\mathit{comment}\ \texttt{or}\ \mathit{text}\ \texttt{or}\ \texttt{or}_{a\in A}\, a)\ \texttt{and}\ \texttt{child::}* \tag{6}$$

Note that this does not enforce the XML standard of having at most one a-labelled attribute for every element. This might actually be desirable, for instance for handling set-valued attributes, like the `class` ones in HTML 5. But it can be otherwise remedied with a disjunction with (7):

$$\cdots\ \texttt{or}\ \texttt{or}_{a\in A}(\texttt{for \$x in child::}a\ \texttt{return}\ \texttt{for \$y in child::}a\ \texttt{return .[not(\$x is \$y)]}) \tag{7}$$

In case we are working with a fragment without `for` and `is`, but with data joins, (8) ensures instead that all the a-labelled attributes share the same data

$$\cdots\ \texttt{or}\ \texttt{or}_{a\in A}(\texttt{child::}a\ \texttt{ne child::}a) \tag{8}$$

We might want to ensure that identifiers are unique. To simplify matters, let us assume that all the unique identifiers use the attribute name $\texttt{id} \in A$; then we add

$$\cdots\ \texttt{or}\ \big(\texttt{id and (. eq }\mathit{future}(\texttt{id}))\big) \tag{9}$$

Finally, we should also ensure that data values are consistent throughout the tree. Remember that the value of an element node should be the concatenation of the values of its element and text children. There is no way to do this without access to string-processing functions, so the XML semantics and data tree semantics do not quite coincide. Most of the literature accordingly restricts data joins $\pi \vartriangle \pi'$ and data tests $\pi \vartriangle^{+} d$ to paths ending with an attribute step: only $\pi/\texttt{@}a \vartriangle \pi'/\texttt{@}a'$ and $\pi/\texttt{@}a \vartriangle^{+} d$ are allowed in their syntax. We do not enforce this restriction in our concrete syntax specifications, but the effect is limited; for instance, there are only 381 occurrences in the benchmark of a data test $\pi \vartriangle^{+} d$ where π is neither a function call nor a variable and does not end with an attribute step.

4 XPATH FRAGMENTS

We present here the fragments with decidable satisfiability and containment we have considered in our experiments. As there is such an abundant literature on the topic [e.g. 9, 10, 20–22, 25, 29, 34, 40, 43, 45], this is clearly an incomplete sample, but we think it is representative of the main lines of investigation.

The fragments we consider in our experiments and their inclusions are shown in Fig. 6, along with the complexity of satisfiability in each fragment. Regarding complexity, we use the DAG-size of the input expression, where isomorphic sub-expressions are shared. Note that Fig. 6 reports the complexity for the original logics, thus for EMSO^2 and non-mixing MSO constraints, the complexity of the XPath fragments we translate into the logics might be lower.

4.1 Decidable XPath Fragments

4.1.1 Positive XPath. Some of the earliest-studied fragments of XPath are based on tree patterns [32]. Geerts and Fan [26, Thm. 4]

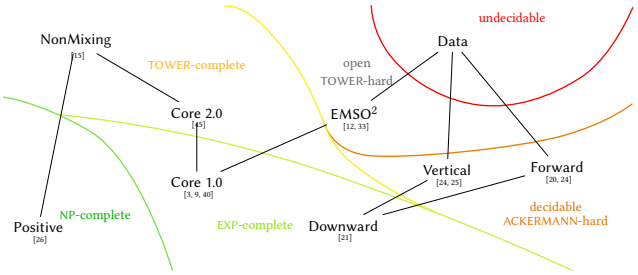

Figure 6: Inclusions and complexities of the fragments of Sec. 4.

show that the following PositiveXPath fragment is NP-complete, even in presence of a DTD,

$$\pi ::= \alpha::* \mid \pi/\pi \mid \pi[\varphi] \mid \pi\ \texttt{union}\ \pi \mid \pi\ \texttt{intersect}\ \pi$$
$$\varphi ::= \pi \mid a \mid \texttt{false()} \mid \texttt{true()} \mid \varphi\ \texttt{or}\ \varphi \mid \varphi\ \texttt{and}\ \varphi \mid \pi \vartriangle \pi$$

where a ranges over Σ and $\vartriangle$ over $\{\texttt{eq}, \texttt{ne}\}$. This fragment has a semantics-preserving translation into the existential positive fragment of first-order logic with signature $(\downarrow, \downarrow^*, \rightarrow, \rightarrow^*, (P_a)_{a\in\Sigma}, \sim, \nsim)$.

4.1.2 Core XPath 1.0. A landmark fragment is the language of Gottlob and Koch [29], known in the literature as 'CoreXPath'. This language is akin to propositional dynamic logic on trees [3], and is defined by the abstract syntax

$$\pi ::= \alpha::* \mid \pi/\pi \mid \pi[\varphi] \mid \pi\ \texttt{union}\ \pi$$
$$\varphi ::= \pi \mid a \mid \texttt{false()} \mid \texttt{not}(\varphi) \mid \varphi\ \texttt{or}\ \varphi$$

where a ranges over Σ. CoreXPath has an EXP-complete satisfiability problem, also in presence of a DTD. To the best of our knowledge, this is the only fragment in this section for which an implementation of a satisfiability procedure exists [27].

4.1.3 Core XPath 2.0. The main feature of XPath 2.0 was the introduction of `for` loops. Having `for` loops further enriches XPath with variable quantification, and provides a significant jump in expressiveness (see Sec. 3.5.2). Ten Cate and Lutz [45] study the extension of CoreXPath with

$$\pi ::= \cdots \mid \texttt{\$x} \mid \texttt{for \$x in}\ \pi\ \texttt{return}\ \pi$$
$$\varphi ::= \cdots \mid \texttt{\$x is \$y} \mid \texttt{. is \$x}$$

where \$x and \$y range over $\mathcal{X}$; we call the resulting fragment CoreXPath 2.0. The syntax of node identity tests in [45] is slightly more restrictive than ours, but this can be fixed by seeing π `is` π' as a shorthand for

$$\mathit{singleton}(\pi)\ \texttt{and}\ \mathit{singleton}(\pi')\ \texttt{and for \$x in}\ \pi\ \texttt{return}\ \texttt{for \$y in}\ \pi'\ \texttt{return .[\$x is \$y]} \tag{10}$$

A deeper difference lies in the semantics of variables: ten Cate and Lutz [45] assume that valuations map to single nodes. This does not make any difference regarding bound variables, since in CoreXPath 2.0 they must be bound by a `for` expression, but it does make one for free variables. This is not an issue, since the decision procedure for CoreXPath 2.0 can handle those.

Indeed, satisfiability in CoreXPath 2.0 is decidable in time bounded by a tower of exponentials, whose height depends on the size of the expression. This is seen by reducing the problem to satisfiability in $\mathrm{MSO}(\downarrow, \rightarrow, (P_a)_{a\in\Sigma})$, using the usual *standard translation* of CoreXPath expressions into MSO formulæ [see e.g. 3, 39]; the resulting TOWER complexity upper bound is tight [45, Thm. 31]. Thus our free XPath variables translate directly into free second-order variables in $\mathrm{MSO}(\downarrow, \rightarrow, (P_a)_{a\in\Sigma})$.

4.1.4 Data XPath. A well-studied XPath fragment with the ability to test data equality and disequality is DataXPath [26]. It is obtained by adding *joins* to the syntax of CoreXPath as follows, where $\triangle$ ranges over $\{\mathtt{eq}, \mathtt{ne}\}$:

$$\varphi ::= \cdots \mid \pi \triangle \pi$$

Although the satisfiability of DataXPath is undecidable [26], restricting the navigational power restores decidability [23]. The first decidable fragment we consider is VerticalXPath, shown decidable by Figueira and Segoufin [25, Thm. 2.1], which restricts the syntax of DataXPath to only allow vertical navigation:

$$\alpha ::= \mathtt{self} \mid \mathtt{child} \mid \mathtt{descendant} \mid \mathtt{parent} \mid \mathtt{ancestor}$$

Another decidable fragment is ForwardXPath, shown decidable by Figueira [20, Thm. 6.4], where navigation is restricted to forward axes only:

$$\alpha ::= \mathtt{self} \mid \mathtt{child} \mid \mathtt{descendant} \mid \mathtt{following\text{-}sibling}$$

We also consider DownwardXPath [21, Thm. 6.4], the intersection of VerticalXPath and ForwardXPath, where only downward navigation is allowed:

$$\alpha ::= \mathtt{self} \mid \mathtt{child} \mid \mathtt{descendant}$$

As seen in Fig. 6, the complexity of the satisfiability problem in these three fragments varies considerably: DownwardXPath is EXP-complete, but VerticalXPath and ForwardXPath are ACKERMANN-hard [24]. It is also notable that satisfiability of DownwardXPath also becomes ACKERMANN-hard in presence of DTDs [24].

4.1.5 Existential MSO². Bojańczyk, Muscholl, Schwentick, and Segoufin [12] investigate the satisfiability of formulæ of the form $\exists X_1 \cdots \exists X_n \,.\, \psi$, where $X_1, \ldots, X_n$ are monadic second-order variables and ψ is a first-order formula in the two-variable fragment

- over the signature $(\downarrow, \rightarrow, (P_a)_{a\in\Sigma}, \sim)$, which they denote by $\mathrm{EMSO}^2(\sim, +1)$, or
- over the signature $(\downarrow, \downarrow^+\!\rightarrow, \rightarrow^+, (P_a)_{a\in\Sigma}, \sim)$, which they denote by $\mathrm{EMSO}^2(\sim, <, +1)$.

In the first instance, they prove the decidability of satisfiability in 3-NEXP [12, Thm. 3.1], while the best known lower bound is NEXP-hardness, which holds already for $\mathrm{FO}^2(\downarrow, (P_a)_{a\in\Sigma})$ [7, Thm. 5.1]. In the second instance, decidability is open, and equivalent to the reachability problem in an extension of branching vector addition systems [33], with a TOWER lower bound [36].

These results can be exploited for a fragment EMSO^2XPath of DataXPath: Bojańczyk et al. [12, Thm. 6.1] allow the following restricted joins in CoreXPath

$$\pi ::= \cdots \mid \pi \triangle /\pi \mid \,.\, \triangle \alpha{::}*[\varphi]$$

where $\triangle$ ranges over $\{\mathtt{eq}, \mathtt{ne}\}$. When the above π, π', and α are restricted to using the axes `self`, `child`, and `parent`, this can be translated into $\mathrm{EMSO}^2(\sim, +1)$, and the general form into $\mathrm{EMSO}^2(\sim, <, +1)$.[3] In spite of the unknown decidability status of $\mathrm{EMSO}^2(\sim, <, +1)$, we have run our benchmarks against the full logic.

4.1.6 Non-Mixing MSO Constraints. Czerwinski, David, Murlak, and Parys [15] define *MSO constraints* as formulæ of the form $\psi(\bar{x}) \implies \eta_\sim(\bar{x}) \wedge \eta_{\not\sim}(\bar{x})$, where ψ is an $\mathrm{MSO}(\downarrow, \rightarrow, (P_a)_{a\in\Sigma})$ formula with the first-order variables $\bar{x}$ as its free variables, and $\eta_\sim$ and $\eta_{\not\sim}$ are positive Boolean combinations of atoms, over the respective signatures $(\sim, (P_d)_{d\in\mathbb{D}})$ and $(\not\sim, (\neg P_d)_{d\in\mathbb{D}})$. Hence data tests and data joins are permitted, as long as they are not *mixed*. Satisfiability is called consistency in this context, and is decidable [15, Thm. 4]; better complexities are achievable when restricting ψ to conjunctive queries.

To quote Czerwinski et al. [15], their 'results imply decidability… of the containment problem in the presence of a schema for unions of XPath queries without negation, where each query uses either equality or inequality, but never both.' Here is indeed a NonMixingXPath fragment of XPath, which can be translated to MSO constraints:

$$\varphi_c ::= \varphi_{\mathtt{eq}} \mid \varphi_{\mathtt{ne}} \mid \varphi_c \text{ or } \varphi_c$$

where $\triangle$-*expressions*, for $\triangle$ in $\{\mathtt{eq}, \mathtt{ne}\}$, are defined by

$$\pi_\triangle ::= \alpha{::}* \mid \pi_\triangle/\pi_\triangle \mid \pi_\triangle[\varphi_\triangle] \mid \pi_\triangle \text{ union } \pi_\triangle$$
$$\varphi_\triangle ::= \mathtt{true}() \mid \mathtt{false}() \mid \pi_\triangle \mid \varphi \mid \varphi_\triangle \text{ or } \varphi_\triangle \mid \varphi_\triangle \text{ and } \varphi_\triangle \mid \pi_\triangle \triangle \pi_\triangle \mid \pi_\triangle \triangle d$$

where φ is any CoreXPath 2.0 node expression (see App. C for the translation into MSO constraints).

4.2 Baseline Benchmark Results

We have implemented the fragments of this section as Relax NG schemas. In each case we included obvious extensions, such as the syntactic sugars discussed in Sec. 3.5 (in particular, the `last()` function is included in CoreXPath 2.0 and NonMixingXPath).

The results of these fragments on the benchmark are presented in grey in figures 7 and 8 (p. 10). The fragments allowing free variables, namely CoreXPath 2.0, EMSO^2XPath, and NonMixingXPath, have the best baseline coverage. We see here the practical interest of a fragment like NonMixingXPath with restricted negation but some support for variables, data tests $\pi \triangle d$, and data joins $\pi \triangle \pi$. The other fragments have an essentially negligible coverage in the XQuery benchmarks (Fig. 8). The support for unrestricted joins $\pi \triangle \pi'$ in the fragments of Sec. 4.1.4 has a very limited effect, and indeed we only found 65 relevant instances in the entire benchmark, i.e. where neither π nor π' is a variable or a function call.

Of course, the fragments defined in the literature were not meant to be run against concrete XPath queries; we will see in the next section that several extensions can be made to these fragments.

5 EXTENSIONS

In this section, we introduce several extensions of the fragments from Sec. 4, while preserving the decidability and complexity of

[3] Bojańczyk et al. [12] actually also allow joins of the form $@a \triangle \alpha{::}*/@b$, but this is subject to a semantic condition.

Table 2: Occurrence counts of the syntactic extensions of Sec. 5 in the entire benchmark.

	Basic		Advanced		
$/\pi$	$\$x$	$\pi \triangle^+ d$	$\pi \triangle \pi$	last()	id()
272	12,476	4,791	853	1,203	31

the corresponding satisfiability problems. As seen in Tab. 2, we consider first 'basic' extensions with considerable impact on the benchmark coverage in sections 5.2.1 to 5.2.3, and then 'advanced' ones with smaller impact in sections 5.2.4 to 5.2.6.

5.1 Handling an Extension in a Fragment

A first way to prove that an extension can be handled is, when given a node expression φ in the extended syntax, to compute an *equivalent* node expression φ' in the original fragment, i.e. such that for all data trees and valuations ν, $[\![\varphi]\!]^\nu_{\mathsf{N}} = [\![\varphi']\!]^\nu_{\mathsf{N}}$—and similarly for path expressions. In such a case, we say that the extension can be *expressed* in the fragment, and *polynomially* so if φ' can be computed in polynomial time.

A second way is instead, when given a node expression φ in the extended syntax, to compute an *equisatisfiable* node expression φ' in the original fragment, i.e. such that there exists t with $t \models \varphi$ if and only if there exists t' with $t' \models \varphi'$. In such a case, we say that the extension can be *encoded* in the fragment, and that it can be *polynomially encoded* if φ' can be computed in polynomial time. Clearly, an extension that can be (polynomially) expressed can also be (polynomially) encoded, but the converse might not hold.

Last of all, the proof techniques employed to show the decidability of satisfiability in a fragment might allow to handle the extension at hand.

5.2 Extensions

5.2.1 $/\pi$: Root Navigation. In XPath, navigation to the root is possible through the $/\pi$ construct as well as using the nullary root() function [18], with semantics $[\![\texttt{root}]\!]_{\mathcal{F}} \stackrel{\text{def}}{=} \{\varepsilon\}$.

We naturally allow these features in CoreXPath 1.0 and 2.0 as well as in the NonMixingXPath and EMSO^{2}XPath fragments, where navigation to the root is captured by

$$\texttt{ancestor-or-self::*[not(parent::*)]} \tag{11}$$

The same goes for VerticalXPath but not for the two other DataXPath fragments, where one cannot navigate upwards. It is clear that root navigation is not expressible in these fragments. In fact, it cannot even be encoded in ForwardXPath, since it becomes undecidable when extended with navigation to the root, as can be seen by adapting the proofs from [24] (see App. D).

Proposition 5.1. *Satisfiability in ForwardXPath extended with root navigation is undecidable.*

We leave open the question whether there is a (polynomial) encoding of root navigation in DownwardXPath.

Finally, regarding PositiveXPath, Hidders's original fragment [32] allowed root navigation, and his proof of a small model property (showing the satisfiability problem to be in NP) applies mutatis mutandis to the fragment with data joins defined in Sec. 4.1.1.

5.2.2 $\$x$: Free Variables. The XPath specification mentions that all variables are essentially second-order. More precisely, a variable is interpreted as an ordered collection of items which may be nodes or data values. In practice, queries extracted from XQuery or XSLT applications contain variables that are bound by the host language. They may be bound to nodes, node collections, or data values. It is out of the scope of the present work to recover such information to consider a more specific satisfiability problem. Rather, we interpret all variables as unordered collections of nodes, as can be seen in our semantics.

In most of our fragments, free variables are admissible. Indeed, any formula φ over Σ and $\mathbb{D}$ with a (necessarily finite) set of free variables $X \subseteq \mathcal{X}$ can be translated into an equisatisfiable formula φ^X over $\Sigma \times 2^X$ and $\mathbb{D}$ with no free variables. Let us write a_S for $(a, S) \in \Sigma \times 2^X$. The key translation steps are:

$$(\$x)^X \stackrel{\text{def}}{=} //.\Big[\bigvee_{a\in\Sigma,\ \$x\in S\subseteq 2^X} a_S\Big] \qquad (a)^X \stackrel{\text{def}}{=} \bigvee_{a\in\Sigma,\ S\subseteq 2^X} a_S \tag{12}$$

Assuming without loss of generality that the variables bound by constructs such as for or let do not belong to X, they are not affected by this translation.

Proposition 5.2. *Free variables can be encoded in PositiveXPath, CoreXPath 1.0, CoreXPath 2.0, VerticalXPath, NonMixingXPath, and EMSO^{2}XPath.*

Note that although the encoding is exponential, the extension does not actually impact the complexity of satisfiability: in PositiveXPath and CoreXPath 1.0, a polynomial encoding can be obtained, leveraging a variant of the semantics where multiple propositions may hold at a node. In VerticalXPath, satisfiability is ACKERMANN-hard, so an exponential blow-up will not have an effect on the worst-case complexity. The decision procedures for the fragments CoreXPath 2.0, NonMixingXPath, and EMSO^{2}XPath are based on second-order logic, hence they actually allow XPath variables in their baseline version.

We finally observe that this translation is not available in DownwardXPath and ForwardXPath, because they cannot express the root path of (12). In fact, free variables cannot be encoded in ForwardXPath; this is similar to Prop. 5.1 (see App. D).

Proposition 5.3. *Satisfiability in ForwardXPath extended with one free variable is undecidable.*

5.2.3 $\pi \triangle^+ d$: Data Tests against Constants. We now consider the extension with direct comparisons against constants from $\mathbb{D}$, assuming that $<$ is a dense total order:[4]

$$\varphi ::= \cdots \mid \pi \triangle^+ d$$

where $\triangle^+ \in \{\texttt{eq}, \texttt{ne}, \texttt{le}, \texttt{lt}, \texttt{ge}, \texttt{gt}\}$ and $d \in \mathbb{D}$.

Proposition 5.4. *Data tests can be polynomially encoded in CoreXPath 1.0, CoreXPath 2.0, VerticalXPath, DownwardXPath, ForwardXPath, and EMSO^{2}XPath.*

We show in App. E that any formula over Σ and $\mathbb{D}$ featuring comparisons against constants in a finite subset $D \subseteq \mathbb{D}$ can be

[4] These assumptions are not met in the actual XPath model where comparisons are undefined between numeric and arbitrary string values, but this problem can be avoided e.g. when the type of attributes is known from a schema.

transformed into an equisatisfiable formula over an extended labelling set $\Sigma \times C_D$. This is similar to the treatment of free variables in Sec. 5.2.2, but requires to include data consistency constraints in the encoded formula when the fragment at hand supports data joins. Crucially, these consistency constraints are mixing, and thus not available in NonMixingXPath. Regarding PositiveXPath, the small model property [32, Lem. 1] still holds in the presence of data tests, hence satisfiability remains in NP for this extension.

Going slightly further, we allow comparisons against constant data *expressions* in our fragments, where constant expressions are built from constant values and the deterministic context-insensitive functions of the XPath specification [18]; e.g., `@n eq 3 + 1` is allowed.

5.2.4 $\pi \triangle \pi$: Positive Data Joins. We observe that most of our fragments can be extended to allow restricted occurrences of data joins. Intuitively, we allow data joins in positions that guarantee that the join will be evaluated only once during satisfaction checking, which allows us to replace it by two tests against a specially chosen data constant. Hence, we extend any fragment with

$$\begin{aligned}\pi_+ ::= {} & \pi \mid /\pi_+ \mid \pi_+/\pi_+ \mid \pi_+[\varphi_+] \mid \pi_+ \texttt{ union } \pi_+ \\ & \mid \pi_+ \texttt{ intersect } \pi_+ \mid \pi_+ \texttt{ except } \pi \\ & \mid \texttt{some \$x in } \pi_+ \texttt{ satisfies } \varphi_+ \\ \varphi_+ ::= {} & \varphi \mid \pi_+ \mid \varphi_+ \texttt{ or } \varphi_+ \mid \varphi_+ \texttt{ and } \varphi_+ \\ & \mid \pi_+ \texttt{ is } \pi_+ \mid \pi_+ \triangle \pi_+ \mid \pi_+ \triangle d\end{aligned}$$

Productions for constructs such as navigation to the root, intersection, node comparison, etc. should only be considered in fragments where they are allowed. We justify this extension in App. F for all relevant fragments: NonMixingXPath does not support the mixing data tests required in our encoding, and PositiveXPath already supports positive data joins in its baseline version.

Proposition 5.5. *Positive joins are expressible in CoreXPath 1.0, CoreXPath 2.0 and EMSO^{2}XPath.*

5.2.5 `last()`: Positional Predicates. The typical use of `last()` in XPath is through a *positional predicate* π[`position() = last()`] or π[`last()`] that only keeps the last node in the document order among all those selected by π. We can also check whether a node is the ith one for some $i > 0$ with $\pi[i]$, or not the last one with π[`position() != last()`] or not the ith one with π[`position() != `i]. As seen in Tab. 2, these constructions are quite frequent in the benchmark. Here we discuss the case of `last()` and its negation, but the other positional predicates can be handled in a similar fashion.

As explained in Sec. 3.5.2, this kind of predicates is not supported in our simplified semantics, thus we rather focus on the one-argument functions `last`(π) and *notlast*(π)—the latter is not a standard function—, with semantics $[\![\texttt{last}]\!]_{\mathcal{F}}(S) \stackrel{\text{def}}{=} \max_{\ll} S$ and $[\![\mathit{notlast}]\!]_{\mathcal{F}}(S) \stackrel{\text{def}}{=} S \setminus \{\max_{\ll} S\}$ for any $S \subseteq N$.

Recall from Sec. 3.5.2 that `last()` can be expressed natively in CoreXPath 2.0 and thus in NonMixingXPath. The question here is to which extent it can be handled in the other fragments. Our first result is that in some cases, the `last()` and *notlast*() functions cannot be expressed.

Proposition 5.6. *The expression* `last(ancestor::`a`) [child::`b`]` *is not expressible in VerticalXPath.*

This result, proved in App. G.1 using bisimulation techniques, shows that `last()` cannot be expressed in VerticalXPath nor in DownwardXPath, even for simple one-step paths. Furthermore, we can show that it cannot be polynomially encoded in DownwardXPath by adapting the hardness proofs of [24]; see App. D.

Proposition 5.7. *Satisfiability in DownwardXPath extended with* `last(descendant-or-self::*)` *and notlast*(`descendant-or-self::*`) *is* ACKERMANN*-hard.*

However, we can still look for some uses of `last()` that can be reasonably allowed. In particular, the path expressions from the statement of Prop. 5.7 can be handled in ForwardXPath—or at least in its *regular* extension [44] with new axes and the Kleene plus and Kleene star operators on paths

$$\begin{aligned}\alpha ::= {} & \cdots \mid \texttt{previous-sibling} \mid \texttt{next-sibling} \\ \pi ::= {} & \cdots \mid \pi^+ \mid \pi^*\end{aligned}$$

with semantics $[\![\texttt{previous-sibling}]\!]_{\mathsf{A}} \stackrel{\text{def}}{=} \rightarrow^{-1}$, $[\![\texttt{next-sibling}]\!]_{\mathsf{A}} \stackrel{\text{def}}{=} \rightarrow$, $[\![\pi^+]\!]^{\nu}_{\mathsf{P}} \stackrel{\text{def}}{=} ([\![\pi]\!]^{\nu}_{\mathsf{P}})^+$, and $[\![\pi^*]\!]^{\nu}_{\mathsf{P}} \stackrel{\text{def}}{=} ([\![\pi]\!]^{\nu}_{\mathsf{P}})^*$. As far as satisfiability is concerned, this extension comes 'for free' in CoreXPath 1.0 [3, 39], in ForwardXPath [20, Thm.6.4] (where `next-sibling` is allowed but `previous-sibling` is not), and in VerticalXPath [25, Thm. 2.1] and DownwardXPath [21, Thm. 6.4] (without the new axes): the complexity of satisfiability does not increase. We show in App. G.2 that we can handle using this regular extension `last`(π) and *notlast*(π) on any *one-step* path argument of the form $\pi = \alpha$::*[φ].

5.2.6 `id()`: Jumps. Another interesting XPath feature in the XML document model is the id/idref mechanism [10, 38]. The function `id()` takes a path as argument, and returns the nodes of the document (if any) that have an `@id` attribute matching a datum found at the end of the path given as argument:

$$[\![\texttt{id}]\!]_{\mathcal{F}}(S) \stackrel{\text{def}}{=} \{p \mid \exists p' \in S \,.\, \delta(p') = \delta([\![\texttt{@id}]\!]^{\nu}_{\mathsf{P}}(p))\}$$

The function `idref()` is its inverse. For instance, in Fig. 3, evaluating `id(@ref)` at the t node returns the o node.

Adding the `id()` function makes most of the fragments undecidable, as soon as we can also use (full) data joins or node tests π `is` π'. The proof in App. H reduces from Post's correspondence problem.

Proposition 5.8. *Satisfiability is undecidable in both DownwardXPath and CoreXPath 2.0 extended with* `id()`.

We nevertheless show in App. H that limited support for the `id()` function can be provided in VerticalXPath and EMSO^{2}XPath.

5.3 Extended Benchmark Results

We have implemented the extensions of this section as Relax NG schemas. The results on the benchmark are presented in violet in figures 7 and 8. Generally, the differences observed before still hold but are significantly lessened. Strikingly, the extensions even bring CoreXPath 1.0 above NonMixingXPath: the latter only supports non-mixing data tests. It also differentiates VerticalXPath from the two other DataXPath fragments, due to its support for root paths, which in turn allows to support free variables.

Looking at the influence of each extension separately, Tab. 3 shows that the 'basic' extensions of sections 5.2.1 to 5.2.3 contribute

Table 3: Number of new queries captured by the extensions of Sec. 5.2 in each fragment.

	basic	$\pi \triangle \pi$	`last()`	`id()`
Positive	7,653			
Core 1.0	7,895	+243	+54	
Core 2.0	4,309	+266		
Downward	1,993		+12	
Vertical	7,974		+25	+0
Forward	2,053		+26	
EMSO^2	4,760	+241		+11
NonMixing	136			

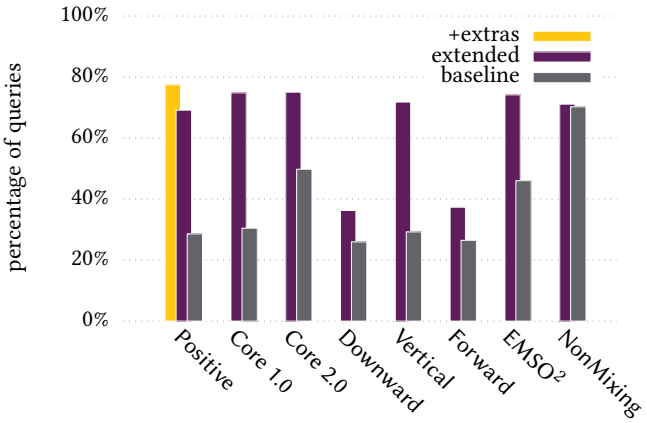

Figure 7: Coverage of the XSLT sources.

most of the gains, while adding positive joins, positional predicates, or `id()` jumps to the basic extensions only brings small improvements. Regarding positional predicates, we found that many occurrences are of the form `last($x)`, which is outside the scope of our treatment in Sec. 5.2.5. Regarding `id()`, Tab. 2 shows that there are *very* few occurrences of `id()` in the benchmark; one of these examples is shown in App. B. A quick investigation of the usage of the `id` attribute in the benchmark shows that developers rather interact with it through variables and data tests.

6 DISCUSSION

Figures 7 and 8 show the coverage of the benchmark: in grey for the baseline fragments from Sec. 4 and in violet for their extensions described in Sec. 5; the yellow 'extras' are the topic of Sec. 6.2.2. The combined coverage of the extended fragments on the full benchmark is of 12,867 queries (60.86%).

Leaving DownwardXPath and ForwardXPath aside, the extended versions of the remaining six fragments have a somewhat similar coverage: for XSLT queries, between 69.16% for PositiveXPath and 75.03% for CoreXPath 2.0, and for XQuery ones, between 26.35% for NonMixingXPath and 28.08% for CoreXPath 2.0. We look more closely at the differences between the fragments in Sec. 6.1.

Obviously, the coverage of XPath queries extracted from XQuery files is quite poor compared to that of XSLT files. Among the other factors, we see that the size of the query is (negatively) correlated

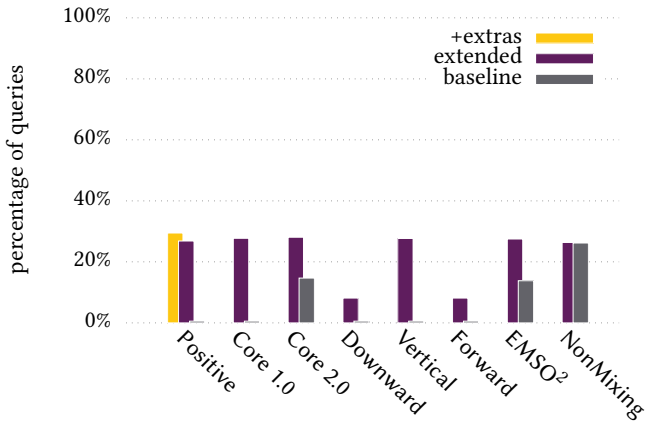

Figure 8: Coverage of the XQuery sources.

(Tab. 4). Another correlation is the presence of at least one axis step, where the combined coverage is of 74.50%, but only 48.79% for queries without any axis step. The main factor we identify is however the presence of non-standard or unsupported function calls in the query, which we discuss in Sec. 6.2.

6.1 Comparisons Between Fragments

In the case of the extended fragments of Sec. 5, the inclusions of Fig. 6 are slightly changed: CoreXPath 2.0 now contains NonMixingXPath and PositiveXPath is included into CoreXPath 2.0 but disjoint from NonMixingXPath.

These theoretical inclusions are reflected in the difference matrix[5] shown in Tab. 5 and the accompanying chord graph of Fig. 9 in the appendix. There are three maximal incomparable fragments, namely CoreXPath 2.0, VerticalXPath, and EMSO^{2}XPath. Moreover, the coverage of the extended CoreXPath 2.0 is almost as large as the combined coverage: only 30 queries from VerticalXPath are not captured by CoreXPath 2.0, and they all contain data joins under a negation; only 12 queries from EMSO^{2}XPath are not captured, which include 11 queries with `id()` plus one of the previous 30.

From a more practical perspective, we think that the extended versions of PositiveXPath and CoreXPath 1.0 are the most promising ones: satisfiability has a more manageable complexity (NP-complete and EXP-complete, resp.), and the coverage is not too far behind CoreXPath 2.0 (with 942 and 57 fewer queries, resp.). Note that PositiveXPath is nearly included into CoreXPath 1.0, with only four queries (featuring intersections) not captured by CoreXPath 1.0.

6.2 Supporting Functions

Due to the large number of calls to non-standard functions in the benchmark, the coverage of 'XPath 3.0 std' (cf. Tab. 1) is an upper bound on the achievable coverage. With respect to the number of queries captured by 'XPath 3.0 std', the combined coverage is 78.33%, and the precise coverage varies between 75.71% for PositiveXPath and 82.14% for CoreXPath 2.0 in XSLT sources and between 56.30% for NonMixingXPath and 60.00% for CoreXPath 2.0 in XQuery

[5] Cell (i, j) shows the number of queries covered by fragment i but not fragment j.

Table 4: Combined coverage of the extended fragments by query size.

sizes	1–4	5–8	9–12	13–16	17–20	21–24	25–28	29–32	33–36	37–40	41–44	45–48	≥ 49
queries	5,146	5,661	3,359	1,996	1,330	806	590	503	283	231	117	131	988
coverage	97.9%	58.7%	42.0%	44.2%	54.8%	44.9%	53.0%	42.7%	40.9%	47.6%	35.8%	46.5%	25.7%

Table 5: Difference matrix for the extended fragments.

	Positive	Core 1.0	Core 2.0	Downward	Vertical	Forward	EMSO^2	NonMixing
Positive	0	4	0	6,283	300	6,159	23	331
Core 1.0	889	0	0	6,917	469	6,765	124	681
Core 2.0	942	57	0	6,974	526	6,822	181	689
Downward	251	0	0	0	0	0	12	83
Vertical	746	30	30	6,478	0	6,478	73	671
Forward	279	0	0	152	152	0	26	85
EMSO^2	796	12	12	6,817	400	6,679	0	639
NonMixing	584	49	0	6,368	478	6,218	119	0

sources: the latter sources are more complex even when leaving aside their higher reliance on non-standard functions.

A remaining issue is the support of standard functions. For instance, the four functions that occur the most frequently in the benchmark are in decreasing order `count()`, `concat()`, `local-name()`, and `contains()`, and they are all standard; `local-name()` is supported in our fragments, but the remaining three are not.

6.2.1 Aggregation. CoreXPath 1.0 extended with node expressions $\texttt{count}(\pi)\ \Delta^{+}\ i$ for an integer i can be translated into the two-variable fragment of first-order logic with counting on trees, which has an EXPSPACE decision procedure [6]. There are 314 occurrences of such expressions out of the 624 occurrences of `count()` in the benchmark, but unfortunately not a single query is gained by adding this feature to the extended CoreXPath 1.0 fragment. Capturing more occurrences of `count()` requires arithmetic operations, and leads to an undecidable fragment akin to AggXPath [10].

6.2.2 String Processing and Arithmetic. A promising direction for supporting more functions is the move to SMT solvers—even though it might also mean moving to semi-deciding satisfiability. Linear arithmetic is supported by all solvers, while theories comprising string concatenation, string length, and substring operations are also supported [e.g. 1, 35, 37, 46, 48]. SMT solvers have already been used in [8] to check XQuery inputs, using the classical interval encoding of trees, and the approach could be enriched to cover basic arithmetic and string support. Furthermore, a custom finite tree theory may be added to SMT solvers for more efficiency [e.g. 11, 14].

These considerations lead us to adding support in the extended PositiveXPath for linear arithmetic, the standard functions `concat()`, `contains()`, `string-length()`, and similar ones such as `ends-with()`. We view this fragment as a good candidate for practical satisfiability checking. At 62.75%, the coverage of this fragment, shown in yellow in figures 7 and 8, bests the combined coverage of our other fragments. This translates in particular to 84.77% of the subset of XSLT queries captured by 'XPath 3.0 std'.

This new fragment is incomparable with the others, with a new combined coverage of 67.40% (83.55% for XSLT sources). If we restrict our attention to the 14,732 queries (69.68%) that only use `not()` and the functions supported in this new fragment, the new combined coverage reaches 96.69% (98.10% for XSLT sources). Thus our fragments cover nearly all the queries that do not use unsupported or non-standard functions, from which we conclude that the support for more standard functions is the most promising research avenue if one wishes to improve upon the extended fragments described in Sec. 5.

7 CONCLUDING REMARKS

We have designed a benchmarking infrastructure for testing the practical relevance of XPath fragments, based on the XQueryX format and Relax NG schemas. We have used this benchmark of over 20,000 queries, extracted from the XSLT and XQuery files of open-source projects, to evaluate the syntactic coverage of state-of-the-art XPath fragments for which decidability is known (or still open in the case of EMSO^{2}XPath). Concerning the benchmark itself, it would of course be interesting to incorporate new sources, to confirm our observations on a larger scale.

Our analysis shows that, in a hypothetical satisfiability checker for XPath, the differences between the fragments defined in the theoretical literature are not as important as the differences introduced by the *front-end* translating real XPath inputs to the restricted syntax on which the decision procedure operates. Among the features that such a front-end should support, the most impactful ones would be free variables, data tests against constants (and constant expressions), positive joins, and positional predicates.

According to our benchmark results, such a front-end combined with the decidable fragments from the literature would cover about 70%–75% of the XPath queries found in XSLT files. However, due to the high reliance on user-defined or ill-supported functions, this drops to less than 30% for XPath queries from XQuery files: full-blown program analysis techniques seem necessary for XQuery. As the the support of XPath functions is a key factor, a promising

approach would be to harness the power of modern SMT solvers to handle string-manipulating functions and linear arithmetic, which might cover 77.44% of the XPath queries from the XSLT section of the benchmark.

ACKNOWLEDGMENTS

Work funded by ANR-14-CE28-0005 PRODAQ.

REFERENCES

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2015. Norn: An SMT solver for string constraints. In *CAV '15 (Lect. Notes in Comput. Sci.)*, Vol. 9206. Springer, 462–469. https://doi.org/10.1007/978-3-319-21690-4_29

[2] Sergio Abriola, Diego Figueira, and Santiago Figueira. 2017. Logics of repeating values on data trees and branching counter systems. In *FoSSaCS '17 (Lect. Notes in Comput. Sci.)*, Vol. 10203. Springer, 196–212. https://doi.org/10.1007/978-3-662-54458-7_12

[3] Loredana Afanasiev, Patrick Blackburn, Ioanna Dimitriou, Bertrand Gaiffe, Evan Goris, Maarten Marx, and Maarten de Rijke. 2005. PDL for ordered trees. *J. Appl. Non-Classical Log.* 15, 2 (2005), 115–135. https://doi.org/10.3166/jancl.15.115-135

[4] Diego Arroyuelo, Francisco Claude, Sebastian Maneth, Veli Mäkinen, Gonzalo Navarro, Kim Nguyễn, Jouni Sirén, and Niko Välimäki. 2015. Fast in-memory XPath search using compressed indexes. *Softw. Pract. & Exp.* 45, 3 (2015), 399–434. https://doi.org/10.1002/spe.2227

[5] David Baelde, Simon Lunel, and Sylvain Schmitz. 2016. A sequent calculus for a modal logic on finite data trees. In *CSL 2016 (Leibniz Int. Proc. Inf.)*, Vol. 62. LZI, Article 32. https://doi.org/10.4230/LIPIcs.CSL.2016.32

[6] Bartosz Bednarczyk, Witold Charatonik, and Emanuel Kieronski. 2017. Extending two-variable logic on trees. In *CSL '17 (Leibniz Int. Proc. Inf.)*, Vol. 82. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Article 11. https://doi.org/10.4230/LIPIcs.CSL.2017.11

[7] Saguy Benaim, Michael Benedikt, Witold Charatonik, Emanuel Kieroński, Rastislav Lenhardt, Filip Mazowiecki, and James Worrell. 2016. Complexity of two-variable logic on finite trees. *ACM Trans. Comput. Logic* 17, 4, Article 32 (2016). https://doi.org/10.1145/2996796

[8] Michael Benedikt and James Cheney. 2010. Destabilizers and independence of XML updates. In *Proc. VLDB Endow.*, Vol. 3. VLDB Endowment, 906–917. https://doi.org/10.14778/1920841.1920956

[9] Michael Benedikt, Wenfei Fan, and Floris Geerts. 2008. XPath satisfiability in the presence of DTDs. *J. ACM* 55, 2, Article 8 (2008). https://doi.org/10.1145/1346330.1346333

[10] Michael Benedikt and Christoph Koch. 2009. XPath leashed. *ACM Comput. Surv.* 41, 1, Article 3 (2009). https://doi.org/10.1145/1456650.1456653

[11] Patrick Blackburn, Wilfried Meyer-Viol, and Maarten de Rijke. 1996. A proof system for finite trees. In *CSL '95 (Lect. Notes in Comput. Sci.)*, Vol. 1092. Springer, 86–105. https://doi.org/10.1007/3-540-61377-3_33

[12] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2009. Two-variable logic on data trees and XML reasoning. *J. ACM* 56, 3, Article 13 (2009). https://doi.org/10.1145/1516512.1516515

[13] Mikołaj Bojańczyk and Paweł Parys. 2011. XPath evaluation in linear time. *J. ACM* 58, 4, Article 17 (2011). https://doi.org/10.1145/1989727.1989731

[14] James Cheney. 2011. Satisfiability algorithms for conjunctive queries over trees. In *ICDT '11*. ACM, 150–161. https://doi.org/10.1145/1938551.1938572

[15] Wojciech Czerwinski, Claire David, Filip Murlak, and Paweł Parys. 2018. Reasoning about integrity constraints for tree-structured data. *Theor. Comput. Syst.* 62, 4 (2018), 941–976. https://doi.org/10.1007/s00224-017-9771-z

[16] James Clark (Ed.). 2002. *RELAX NG Compact Syntax.* OASIS Committee Specification. http://relaxng.org/compact.html

[17] Jim Melton (Ed.). 2014. *XQueryX 3.0.* W3C Recommendation. http://www.w3.org/TR/xqueryx-3/

[18] Michael Kay (Ed.). 2014. *XPath and XQuery Functions and Operators 3.0.* W3C Recommendation. https://www.w3.org/TR/xpath-functions-30/

[19] Michael Kay (Ed.). 2017. *XSL Transformations (XSLT) Version 3.0.* W3C Recommendation. http://www.w3.org/TR/xslt-30/

[20] Diego Figueira. 2012. Alternating register automata on finite words and trees. *Logic. Meth. in Comput. Sci.* 8, 1, Article 22 (2012). https://doi.org/10.2168/LMCS-8(1:22)2012

[21] Diego Figueira. 2012. Decidability of downward XPath. *ACM Trans. Comput. Logic* 13, 4, Article 34 (2012). https://doi.org/10.1145/2362355.2362362

[22] Diego Figueira. 2013. On XPath with transitive axes and data tests. In *PODS '13*. ACM, 249–260. https://doi.org/10.1145/2463664.2463675

[23] Diego Figueira. 2018. Automata column: Satisfiability of XPath on data trees. *SIGLOG News* 5, 2 (2018), 4–16. https://doi.org/10.1145/3212019.3212021

[24] Diego Figueira and Luc Segoufin. 2009. Future-looking logics on data words and trees. In *MFCS '09 (Lect. Notes in Comput. Sci.)*, Vol. 5734. Springer, 331–343. https://doi.org/10.1007/978-3-642-03816-7_29

[25] Diego Figueira and Luc Segoufin. 2017. Bottom-up automata on data trees and vertical XPath. *Logic. Meth. in Comput. Sci.* 13, 4, Article 5 (2017). https://doi.org/10.23638/LMCS-13(4:5)2017

[26] Floris Geerts and Wenfei Fan. 2005. Satisfiability of XPath queries with sibling axes. In *DBPL '05*. Springer, 122–137. https://doi.org/10.1007/11601524_8

[27] Pierre Genevès, Nabil Layaïda, Alan Schmitt, and Nils Gesbert. 2015. Efficiently deciding μ-calculus with converse over finite trees. *ACM Trans. Comput. Logic* 16, 2, Article 16 (2015). https://doi.org/10.1145/2724712

[28] Pierre Genevès and Jean-Yves Vion-Dury. 2004. Logic-based XPath optimization. In *DocEng '04*. ACM, 211–219. https://doi.org/10.1145/1030397.1030437

[29] Georg Gottlob and Christoph Koch. 2002. Monadic queries over tree-structured data. In *LICS '02*. IEEE, 189–202. https://doi.org/10.1109/LICS.2002.1029828

[30] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. 2005. The complexity of XPath query evaluation and XML typing. *J. ACM* 52, 2 (2005), 284–335. https://doi.org/10.1145/1059513.1059520

[31] Jinghua Groppe and Sven Groppe. 2006. A prototype of a schema-based XPath satisfiability tester. In *DEXA '06 (Lect. Notes in Comput. Sci.)*, Vol. 4080. Springer, 93–103. https://doi.org/10.1007/11827405_10

[32] Jan Hidders. 2004. Satisfiability of XPath expressions. In *DBPL 2003 (Lect. Notes in Comput. Sci.)*, Georg Lausen and Dan Suciu (Eds.), Vol. 2921. Springer, 21–36. https://doi.org/10.1007/978-3-540-24607-7_3

[33] Florent Jacquemard, Luc Segoufin, and Jérémie Dimino. 2016. $\mathrm{FO}^2(<, +1, \sim)$ on data trees, data tree automata and branching vector addition systems. *Logic. Meth. in Comput. Sci.* 12, 2, Article 3 (2016). https://doi.org/10.2168/LMCS-12(2:3)2016

[34] M. Jurdziński and R. Lazić. 2011. Alternating automata on data trees and XPath satisfiability. *ACM Trans. Comput. Logic* 12, 3, Article 19 (2011). https://doi.org/10.1145/1929954.1929956

[35] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2013. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4, Article 25 (2013). https://doi.org/10.1145/2377656.2377662

[36] Ranko Lazić and Sylvain Schmitz. 2015. Non-elementary complexities for branching VASS, MELL, and extensions. *ACM Trans. Comput. Logic* 16, 3, Article 20 (2015). https://doi.org/10.1145/2733375

[37] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2016. An efficient SMT solver for string constraints. *Form. Methods in Syst. Des.* 48, 3 (2016), 206–234. https://doi.org/10.1007/s10703-016-0247-6

[38] Maarten Marx. 2003. XPath and modal logics of finite DAG's. In *TABLEAUX '03 (Lect. Notes in Comput. Sci.)*, Vol. 2796. Springer, 150–164. https://doi.org/10.1007/978-3-540-45206-5_13

[39] Maarten Marx. 2005. Conditional XPath. *ACM Trans. Datab. Sys.* 30, 4 (2005), 929–959. https://doi.org/10.1145/1114244.1114247

[40] Frank Neven and Thomas Schwentick. 2006. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logic. Meth. in Comput. Sci.* 2, 3, Article 1 (2006). https://doi.org/10.2168/LMCS-2(3:1)2006

[41] Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson (Eds.). 2014. *XML Path Language (XPath) 3.0.* W3C Recommendation. http://www.w3.org/TR/xpath-3/

[42] Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson (Eds.). 2014. *XQuery 3.0: An XML Query Language.* W3C Recommendation. http://www.w3.org/TR/xquery-30/

[43] Thomas Schwentick. 2004. XPath query containment. *SIGMOD Rec.* 33, 1 (2004), 101–109. https://doi.org/10.1145/974121.974140

[44] Balder ten Cate. 2006. The expressivity of XPath with transitive closure. In *PODS '06*. ACM, 328–337. https://doi.org/10.1145/1142351.1142398

[45] Balder ten Cate and Carsten Lutz. 2009. The complexity of query containment in expressive fragments of XPath 2.0. *J. ACM* 56, 6, Article 31 (2009). https://doi.org/10.1145/1568318.1568321

[46] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A symbolic string solver for vulnerability detection in Web applications. In *CCS '14*. ACM, 1232–1243. https://doi.org/10.1145/2660267.2660372

[47] Norman Walsh, Anders Berglund, and John Snelson (Eds.). 2014. *XQuery and XPath Data Model 3.0.* W3C Recommendation. http://www.w3.org/TR/xpath-datamodel-30/

[48] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based string solver for web application analysis. In *ESEC/FSE '13*. ACM, 114–124. https://doi.org/10.1145/2491411.2491456

A ADDITIONAL FIGURES

We have implemented an experimental interactive web interface rendering this information in a graphical way and allowing to explore the different benchmarks; see http://www.lsv.fr/~schmitz/xpparser. It mainly features a *chord graph* rendering of difference matrices, like the one shown in Fig. 9. A chord between fragments i and j has thickness proportional to entry (i, j) on its i end, and to entry (j, i) on its j end; the color is the one of the 'winning' value. Chords with both ends thick, like the one between NonMixingXPath and VerticalXPath, denote fragments with significantly different coverage.

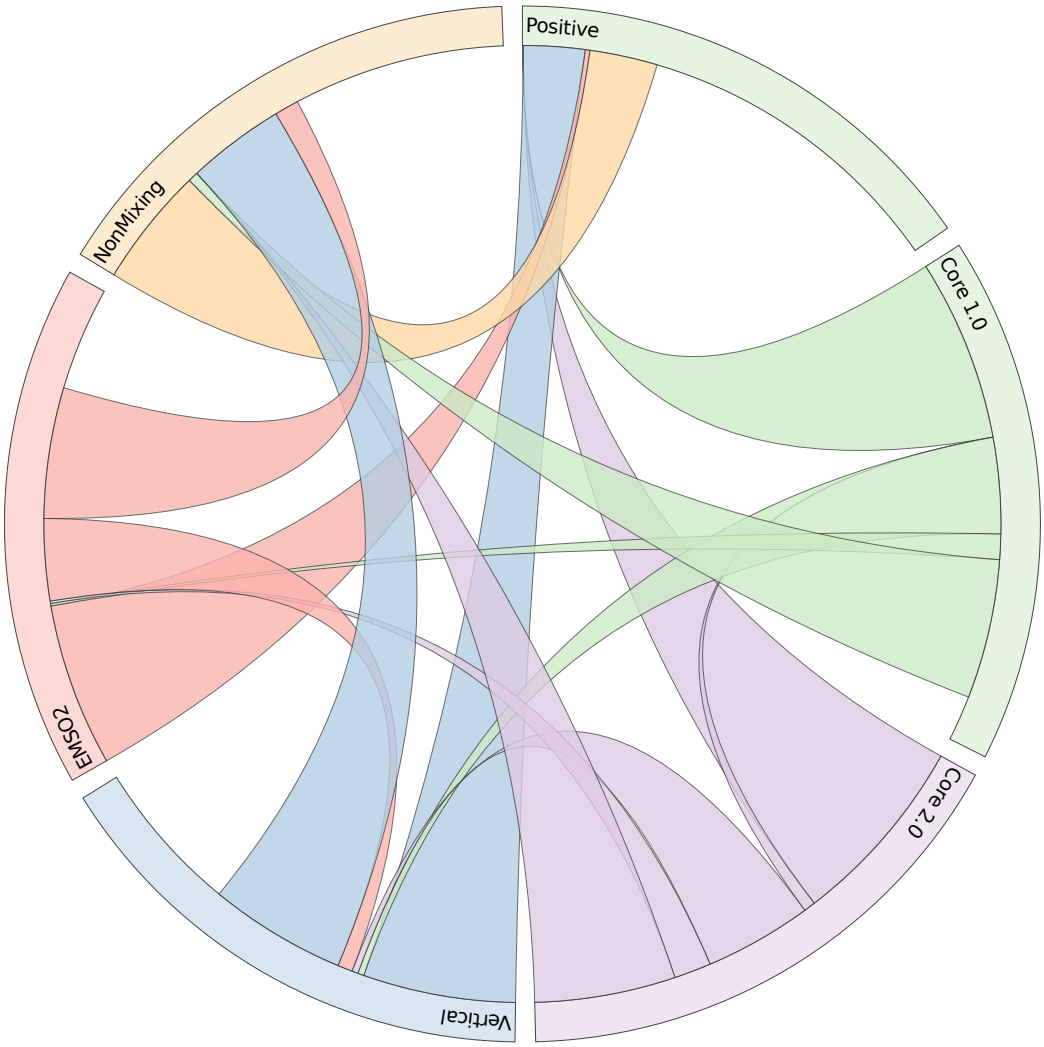

Figure 9: Chord graph of the difference matrix of Tab. 5; we omitted DownwardXPath and ForwardXPath.

Table 6: The benchmark's list of sources.

Source	Queries	Coverage				
		XPath 1.0	XPath 2.0	XPath 3.0	XPath 3.0 std	Core 2.0 extended
DocBook http://docbook.sourceforge.net/	7,620	100.0%	100.0%	100.0%	95.6%	79.3%
TEIXSL https://github.com/TEIC/Stylesheets	6,303	96.4%	100.0%	100.0%	86.1%	70.8%
HTMLBook https://github.com/oreillymedia/HTMLBook	752	100.0%	100.0%	100.0%	92.0%	66.4%
Total (XSLT)	14,675	98.4%	100.0%	100.0%	91.3%	75.0%
XQuery parser https://github.com/jpcs/xqueryparser.xq	1,659	83.1%	85.2%	99.9%	15.7%	12.6%
eXist-db https://github.com/eXist-db	1,151	76.7%	88.0%	100.0%	64.8%	33.1%
HisTEI https://github.com/odaata/HisTEI	483	74.7%	97.5%	100.0%	62.5%	30.2%
transform.xq https://github.com/jpcs/transform.xq	365	64.3%	70.4%	99.7%	45.7%	20.5%
ml-enrich https://github.com/freshie/ml-enrich	302	74.1%	96.3%	100.0%	55.2%	39.4%
xquerydoc https://github.com/xquery/xquerydoc	269	87.3%	98.8%	100.0%	52.7%	28.9%
openinfoman https://github.com/openhie/openinfoman	261	65.1%	96.1%	100.0%	47.8%	26.8%
Oxford Dict API https://github.com/AdamSteffanick/od-api-xquery	207	85.5%	97.5%	100.0%	57.4%	55.5%
MarkLogic Commons https://github.com/marklogic/commons	196	70.9%	93.8%	97.4%	45.4%	24.4%
datascience https://github.com/adamfowleruk/datascience	184	77.1%	91.8%	100.0%	40.7%	21.1%
Link Management BaseX https://github.com/dita-for-small-teams/dfst-linkmgmt-basex/	154	72.0%	96.7%	100.0%	73.3%	36.3%
Semantic Web https://github.com/HeardLibrary/semantic-web/	149	86.5%	93.9%	100.0%	81.2%	51.0%
eXist annotation store https://github.com/telic/exist-annotation-store/	133	80.4%	86.4%	100.0%	64.6%	48.8%
xqtest https://github.com/irinc/xqtest/	130	70.0%	99.2%	100.0%	46.9%	38.4%
data.xq https://github.com/jpcs/data.xq/	119	32.7%	33.6%	100.0%	34.4%	16.8%
graphxq https://github.com/apb2006/graphxq/	92	73.9%	78.2%	100.0%	76.0%	47.8%
ml-invoker https://github.com/fgeorges/ml-invoker/	92	89.1%	89.1%	100.0%	36.9%	32.6%
treedown https://github.com/biblicalhumanities/treedown/	92	94.5%	97.8%	100.0%	80.4%	59.7%
XQJSON https://github.com/joewiz/xqjson/	90	74.4%	100.0%	100.0%	67.7%	55.5%
fots BaseX https://github.com/LeoWoerteler/fots-basex/	73	65.7%	71.2%	100.0%	63.0%	28.7%
GPXQuery https://github.com/dret/GPXQuery/	57	73.6%	98.2%	100.0%	64.9%	29.8%
rbtree.qx https://github.com/jpcs/rbtree.xq/	57	22.8%	28.0%	100.0%	24.5%	0%
xquery-libs https://github.com/adamretter/xquery-libs/	53	79.2%	88.6%	100.0%	60.3%	49.0%
Guid-O-Matic https://github.com/baskaufs/guid-o-matic/	51	84.3%	96.0%	100.0%	56.8%	41.1%
functional.xq https://github.com/jpcs/functional.xq/	47	12.7%	14.8%	100.0%	21.2%	2.1%
Total (XQuery)	6,466	76.1%	87.4%	99.8%	46.7%	28.0%
Total	21,141	91.6%	96.1%	99.9%	77.7%	60.6%

B EXAMPLE PARSER OUTPUT

Here is an XQuery snippet from the file `functx.xqy` of HisTEI:

```
module namespace  functx = "http://www.functx.com" ;

declare function functx:id-from-element
  ( $element as element()? )  as xs:string? {

  data(($element/@*[id(.) is ..])[1])
 } ;
```

The parser identifies 'data(($element/@*[id(.) **is** ..])[1])' as an XPath query inside this program, and returns the following information (note the normalisation of some of the syntactic sugar, like '@*' and '..'), including the syntax tree in XQueryX format [17] inside the <ast> element:

```
<?xml version="1.0"?>
<benchmark>
<xpath column="45" filename="benchmark/example-histei.xqy" line="4">
  <query>data(($element/attribute::*[(id(.) is parent::node())])[1])</query>
  <ast size="30">
    <xqx:functionCallExpr xmlns:xqx="http://www.w3.org/2005/XQueryX">
      <xqx:functionName>data</xqx:functionName>
      <xqx:arguments>
        <xqx:pathExpr>
          <xqx:stepExpr>
            <xqx:filterExpr>
              <xqx:sequenceExpr>
                <xqx:pathExpr>
                  <xqx:stepExpr>
                    <xqx:filterExpr>
                      <xqx:varRef>
                        <xqx:name>element</xqx:name>
                      </xqx:varRef>
                    </xqx:filterExpr>
                  </xqx:stepExpr>
                  <xqx:stepExpr>
                    <xqx:xpathAxis>attribute</xqx:xpathAxis>
                    <xqx:Wildcard/>
                    <xqx:predicates>
                      <xqx:isOp>
                        <xqx:firstOperand>
                          <xqx:functionCallExpr>
                            <xqx:functionName>id</xqx:functionName>
                            <xqx:arguments>
                              <xqx:contextItemExpr/>
                            </xqx:arguments>
                          </xqx:functionCallExpr>
                        </xqx:firstOperand>
                        <xqx:secondOperand>
                          <xqx:pathExpr>
                            <xqx:stepExpr>
                              <xqx:xpathAxis>parent</xqx:xpathAxis>
                              <xqx:anyKindTest/>
                            </xqx:stepExpr>
                          </xqx:pathExpr>
                        </xqx:secondOperand>
                      </xqx:isOp>
                    </xqx:predicates>
                  </xqx:stepExpr>
                </xqx:pathExpr>
              </xqx:sequenceExpr>
            </xqx:filterExpr>
            <xqx:predicates>
              <xqx:integerConstantExpr>
                <xqx:value>1</xqx:value>
              </xqx:integerConstantExpr>
            </xqx:predicates>
          </xqx:stepExpr>
        </xqx:pathExpr>
      </xqx:arguments>
    </xqx:functionCallExpr>
  </ast>
</xpath>
</benchmark>
```

This syntax tree in XQueryX format can be validated against XML Schemas or Relax NG syntactic specifications in order to check whether it fits in some XPath fragments.

C NON-MIXING MSO CONSTRAINTS

In this section, we show how to translate a formula from our NonMixingXPath fragment presented in Sec. 4.1.6 to a formula of the fragment of Czerwinski et al. [15], that is a formula of the form $\bigvee_i(\exists \bar{x_i} \,.\, \alpha_i(\bar{x_i}) \wedge \eta^i_\triangle(\bar{x_i}))$, where the α_i are MSO formulæ and $\eta^i_\triangle$ positive Boolean combinations of $\triangle$ tests. The general idea of this translation is to first describe the tree structure of a tree satisfying the query (by using enough variables), which will provide the formulæ α_i, and then state all the data constraints that should hold between these variables, which will yield the formulæ $\eta^i_\triangle$.

The translation extends the standard translation for CoreXPath 2.0 with a top-level translation for $\triangle$-expressions:

$$\mathrm{ST}_{x,y}(\alpha{::}*) \stackrel{\text{def}}{=} x \,[\![\, \alpha]\!]_{\mathrm{A}}\, y$$

where the semantics of all the axes, as defined in Fig. 4, are indeed definable in the logic $\mathrm{MSO}(\downarrow, \rightarrow, (P_a)_{a\in\Sigma}, (P_d)_{d\in\mathbb{D}})$;

$$\begin{aligned}
\mathrm{ST}_{x,y}(\pi/\pi') &\stackrel{\text{def}}{=} \exists z \,.\, \mathrm{ST}_{x,z}(\pi) \wedge \mathrm{ST}_{z,y}(\pi') && (z \text{ fresh})\\
\mathrm{ST}_{x,y}(\pi[\varphi]) &\stackrel{\text{def}}{=} \mathrm{ST}_{x,y}(\pi) \wedge \mathrm{ST}_{y}(\varphi)\\
\mathrm{ST}_{x,y}(\pi \texttt{ union } \pi') &\stackrel{\text{def}}{=} \mathrm{ST}_{x,y}(\pi) \vee \mathrm{ST}_{x,y}(\pi')\\
\mathrm{ST}_{x}(\texttt{true()}) &\stackrel{\text{def}}{=} \top\\
\mathrm{ST}_{x}(\texttt{false()}) &\stackrel{\text{def}}{=} \bot\\
\mathrm{ST}_{x}(\pi) &\stackrel{\text{def}}{=} \exists y \,.\, \mathrm{ST}_{x,y}(\pi) && (y \text{ fresh})\\
\mathrm{ST}_{x}(\varphi \texttt{ or } \varphi) &\stackrel{\text{def}}{=} \mathrm{ST}_{x}(\varphi) \vee \mathrm{ST}_{x}(\varphi')\\
\mathrm{ST}_{x}(\varphi \texttt{ and } \varphi') &\stackrel{\text{def}}{=} \mathrm{ST}_{x}(\varphi) \wedge \mathrm{ST}_{x}(\varphi)\\
\mathrm{ST}_{x}(\texttt{not}(\varphi)) &\stackrel{\text{def}}{=} \neg\mathrm{ST}_{x}(\varphi)\\
\mathrm{ST}_{x}(\pi_\triangle \triangle \pi'_\triangle) &\stackrel{\text{def}}{=} \exists y \exists z \,.\, \mathrm{ST}_{x,y}(\pi_\triangle) \wedge \mathrm{ST}_{x,z}(\pi'_\triangle) \wedge y \triangle z && (y, z \text{ fresh})
\end{aligned}$$

Since this last case must occur positively, we will always be able to extract the atoms of the form $y \triangle z$ and regroup them in a formula $\eta_\triangle$.

D LOWER BOUNDS IN FORWARD/DOWNWARD XPATH

The following statements are simple consequences of the proofs of Figueira and Segoufin [24]. We first prove the undecidability of the following extensions of ForwardXPath.

Proposition 5.1. *Satisfiability in ForwardXPath extended with root navigation is undecidable.*

Proof. This is similar to the proof of Cor. 4 in Figueira and Segoufin [24].

Their Thm. 2 shows the ACKERMANN-hardness of the satisfiability of simple 1-register freeze LTL over data words, with strict future temporal modalities (denoted by $\mathsf{sLTL}^{\downarrow}_1(\mathsf{F_s})$). As explained in their Prop. 1, any formula from this fragment of freeze LTL can be translated into an equivalent XPath formula. In ForwardXPath, we can force the data trees to consist of a single branch, i.e. to be data words.

As seen in the proof of their Thm. 3, the only reason ForwardXPath is 'only' ACKERMANN-hard instead of undecidable is that, in their construction in Thm. 2 of formulæ simulating runs of Minsky counter machines, one cannot check in ForwardXPath that every decrement was preceded by a matching increment higher in that tree. But we can check that no matching increment occurs down that point with

$$\texttt{not}\Big(\texttt{//.}\Big[\textsc{dec}(i) \texttt{ and (. eq .//.[@])}\Big]\Big) \tag{13}$$

and thus the following ensures that any decrement has a matching increment closer to the root

$$\texttt{not}\Big(\texttt{//.}\Big[\textsc{dec}(i) \texttt{ and not(. eq //.[@])}\Big]\Big) \tag{14}$$

where $\textsc{dec}(i)$ and $@$ are labels in Σ introduced in their construction. □

Proposition 5.3. *Satisfiability in ForwardXPath extended with one free variable is undecidable.*

Proof. We reduce the satisfiability of ForwardXPath + root() over data trees with a single branch to the satisfiability of ForwardXPath with a single variable, which is therefore undecidable by Prop. 5.1.

Let $r be the free variable, and let r be a fresh label not found in Σ. We first ensure that all the nodes in the valuation of $r have r as label, with the constraint not($r[not($r)]).

Since we work with data trees consisting of a single branch, there is a lowest node in $\nu(\$r)$: this is the node selected uniquely by the path expression $r[not(.//$r)]. We use it as our root and perform the entire construction of [24, Thm. 3] and Prop. 5.1 below that node; (14) becomes $\texttt{not}\Big(\$r[\texttt{not}(.//r)]//.\Big[\textsc{dec}(i)$ and not(. eq //. [@])$\Big]\Big)$. □

The following shows that DownwardXPath becomes non primitive-recursive when last() and *notlast*() are added.

Proposition 5.7. *Satisfiability in DownwardXPath extended with* last(descendant-or-self::∗) *and notlast*(descendant-or-self::∗) *is* ACKERMANN*-hard.*

Proof. We adapt the proof of Figueira and Segoufin [24, Cor. 1]. Our aim is to build our formula so as to ensure that the entire simulation of the incrementing counter machine is performed along a single branch of the tree—this will be the rightmost branch. We pick a fresh label last (which is also used in [24, Thm. 2]) and first require

$$\texttt{last(descendant-or-self::}*\texttt{)[}\textsc{last}\texttt{]} \tag{15}$$

so that the last leaf of the (sub)tree below the point of evaluation, in the document order, is labelled by last. We then make sure that no other node in the (sub)tree is not labelled by last

$$\texttt{not(}\textit{notlast}\texttt{(descendant-or-self::}*\texttt{)[}\textsc{last}\texttt{])} \tag{16}$$

We then apply the construction of [24, Prop. 1], but whenever a step descendant::∗ would be used, we replace it by descendant::∗[descendant-or-self::last] to ensure that we only move along the rightmost branch. □

E DATA TESTS AGAINST CONSTANTS

We sketch here the encoding of data tests in XPath.

Proposition 5.4. *Data tests can be polynomially encoded in CoreXPath 1.0, CoreXPath 2.0, VerticalXPath, DownwardXPath, ForwardXPath, and EMSO^2 XPath.*

Consider expressions of any of the considered fragments, extended with data tests against constants taken in a finite subset $D \subseteq \mathbb{D}$. We assume that $(\mathbb{D}, <)$ is dense and unbounded[6]. For convenience, we assume wlog. that $D = \{d_1, \ldots, d_n\}$ with $d_i < d_j$ when $i < j$.

We first define a translation $(\cdot)^D$ which maps node and path expressions over Σ with data tests in D to expressions without data tests but over

$$\Sigma_D \stackrel{\text{def}}{=} \Sigma \times C_D \text{ with } C_D \stackrel{\text{def}}{=} \{-\infty, +\infty\} \cup D \cup \{(d_i, d_{i+1}) \mid 1 \leq i < n\}.$$

As before, compound labels $(a, c) \in \Sigma_D$ are written a_c. Intuitively, the extra information will classify the value x held by a node: either $x < d_1$, or $d_n < x$, or $x = d_i$ or $d_i < x < d_{i+1}$ for some i. It will be convenient to define, for any $d_i \in D$,

$$C_D^{<d_i} \stackrel{\text{def}}{=} \{-\infty\} \cup \{d_j \mid j < i\} \cup \{(d_j, d_{j+1}) \in C_D \mid j < i\}.$$

We give below the key translation steps:

$$\begin{aligned}
(a)^D &\stackrel{\text{def}}{=} \bigvee_{d \in C_D} a_d \\
(\pi \texttt{ eq } d)^D &\stackrel{\text{def}}{=} (\pi)^D\Big[\bigvee_{a \in \Sigma} a_d\Big] \\
(\pi \texttt{ ne } d)^D &\stackrel{\text{def}}{=} (\pi)^D\Big[\bigvee_{a \in \Sigma} \bigvee_{c \in C_D, c \neq d} a_c\Big] \\
(\pi \texttt{ lt } d)^D &\stackrel{\text{def}}{=} (\pi)^D\Big[\bigvee_{a \in \Sigma} \bigvee_{c \in C_D^{<d}} a_c\Big]
\end{aligned}$$

[6] Our argument can easily be adapted to work with the unboundedness assumption: $-\infty$ should simply be dropped from C_D when d_1 is minimal, and similarly for $+\infty$ when d_n is maximal. For target fragments without data joins, the density assumption can be dropped if we also remove (d_i, d_{i+1}) from C_D when $[(d_i, d_{i+1})] = \emptyset$. To get rid of density in other fragments, we would need to know when there exists only finitely many values in some $[c]$ for $c \in C_D$ *and* could express in XPath that nodes with this comparison tag should not carry more than this many distinct values. Unfortunately, the latter does not seem to be feasible.

The cases of gt, le and ge are similar. The translation is homomorphic wrt. all other constructs including data joins ($\pi \triangle \pi'$ with $\triangle \in \{\mathsf{eq}, \mathsf{ne}\}$).

E.1 Equisatisfiability for Data-Consistent Trees

Define for all $c \in C_D$ its interpretation $[c] \subseteq \mathbb{D}$ in the natural way:

$$[-\infty] \stackrel{\text{def}}{=} \{d \in \mathbb{D} \mid d < d_1\}, \quad [d] \stackrel{\text{def}}{=} \{d\}, \quad [(d_i, d_{i+1})] \stackrel{\text{def}}{=} \{d \in \mathbb{D} \mid d_i < d < d_{i+1}\}, \quad \text{etc.}$$

Given a data tree $t = (\ell, \delta)$, we define $t^D \stackrel{\text{def}}{=} (\ell^D, \delta)$ with $\ell^D(p) \stackrel{\text{def}}{=} \ell(p)_c$ when $\delta(p) = d \in D$ and c is the unique element of C_D such that $d \in [c]$. We say that trees of the form t^D are data-consistent, because their data values respect their comparison tags.

Proposition E.1. *For all φ with data tests in D, for all t and p, $t, p \models \varphi$ iff $t^D, p \models \varphi^D$ (and similarly for path expressions).*

Proof. Immediate by induction over expressions. □

E.2 Equisatisfiability

In CoreXPath 1.0 and CoreXPath 2.0, the encoded formula φ^D is insensitive to data values, hence any model t' of φ^D can be modified into a model of the form t^D by changing the data values according to the compound labels. This shows that φ and φ^D are equisatisfiable in these fragments.

For the fragments with data joins under consideration, i.e. VerticalXPath, DownwardXPath, and ForwardXPath, we show that φ is equisatisfiable with the following formula:

$$(\varphi)^D \text{ and } \mathsf{not}\Big(\bigvee_{d \in D,\, a,b \in \Sigma} //a_d \text{ ne } //b_d\Big) \text{ and } \mathsf{not}\Big(\bigvee_{a,b \in \Sigma} \bigvee_{c \neq c' \in C_D} //a_c \text{ eq } //b_{c'}\Big) \tag{17}$$

The above translation is obviously in VerticalXPath when $(\varphi)^D$ is in that fragment. For DownwardXPath and ForwardXPath, because they cannot visit any node above the initial evaluation point and do not allow free variables, it is safe to allow expressions of the form $\mathsf{not}(//.[\mathsf{not}(\varphi)])$ and φ' (like (17)), meaning that φ holds everywhere in the tree and φ' at the point of evaluation. Indeed, this is equivalent in these fragments to $\mathsf{not}(.//.[\mathsf{not}(\varphi)])$ and φ'.

Assuming that this formula admits a model, we show that it has a model of the form t^D, i.e. a model in which data values are consistent with compound labels. We use the fact that modifying the data values of a model in an injective way yields another model, because $(\varphi)^D$ only performs (dis)equality tests on data (through $\pi \triangle \pi'$ constructs):

- Thanks to the extra constraints in our formula we know that, for all $d \in D$, all nodes with a tag in $\Sigma \times \{d\}$ have the same value, and that this value is not present in other nodes of the tree. Thus we can assume wlog. that we have a model t' such that, for any node n of t' and $d \in D$, $\delta(n) = d$ iff $\ell(n) \in \Sigma \times \{d\}$.
- Further, there exists a mapping $f : \mathbb{D} \to \mathbb{D}$ which maps, for any $c \in C_D$, data values occurring in nodes with tag $\Sigma \times \{c\}$ to distinct data values in $[c]$. This relies on the fact that our encoded formula forbids the same data value to occur in nodes with distinct comparison tags. Moreover, by density of the order, we can take this mapping to be injective. Applying this data renaming, we obtain a model of $(\varphi)^D$ of the form t^D, hence a model t of φ.

It is now clear why we could not add data tests in NonMixingXPath: it is not data-insensitive as CoreXPath fragments, but does not allow the (crucially mixed) axiomatization that was needed to obtain equisatisfiability for data-sensitive fragments. Note that this impossibility is slightly mitigated by the fact that some data tests are natively available in NonMixingXPath (in the form $\pi_\triangle \triangle d$).

F POSITIVE DATA JOINS

We detail further the claims of Sec. 5.2.4. We fix below an ambient fragment among CoreXPath 1.0, CoreXPath 2.0 and EMSO^{2}XPath. We consider formulas φ_+ of the fragment, extended with positive data joins, and where joins are decorated with distinct marks in $\mathcal{M}$. We shall encode such formulas to φ formulas in the fragment without data joins but with data tests against constants, which we have shown to be admissible. To justify this extension, we design a translation which associates to any φ_+ expression an equisatisfiable φ expression.

The translation actually works over *marked* φ_+ expressions. Given a query φ_+, we can annotate each occurrence of a data join with a unique mark m from a finite set $\mathcal{M}$. For example,

$$c[@a \text{ eq } @b[\text{. ne preceding:: } */@b]]$$

might be annotated using $\mathcal{M} = \{m, n\}$ as

$$\psi \stackrel{\text{def}}{=} c[@a \text{ eq}_m @b[\text{. ne}_n \text{ preceding:: } */@b]]$$

Then, given a valuation $\alpha\colon \mathcal{M} \to \mathbb{D}$, we define the translation $(\varphi_+)^\alpha$ as follows, showing only the key cases:

$$\begin{aligned}
(\pi)^\alpha &\stackrel{\text{def}}{=} \pi \\
(\pi_+ \text{ eq}_m \pi'_+)^\alpha &\stackrel{\text{def}}{=} \pi_+ \text{ eq } \alpha(m) \text{ and } \pi'_+ \text{ eq } \alpha(m) \\
(\pi_+ \text{ ne}_m \pi'_+)^\alpha &\stackrel{\text{def}}{=} \pi_+ \text{ eq } \alpha(m) \text{ and } \pi'_+ \text{ ne } \alpha(m)
\end{aligned}$$

Continuing the previous example with $\alpha(m) = d$ and $\alpha(n) = d'$, we have:

$$\begin{aligned}
(\psi)^\alpha = c[&d \text{ eq } @a \text{ and} \\
&d \text{ eq } @b[\text{. eq } d' \text{ and } d' \text{ ne preceding:: } */@b]]
\end{aligned}$$

Note that this formula is satisfiable iff $d = d'$.

Obviously, $t, p \models (\varphi_+)^\alpha$ implies $t, p \models \varphi_+$ for any α. Conversely, if $t, p \models \varphi_+$, we show that there exists α such that $t, p \models (\varphi_+)^\alpha$. Roughly, α is chosen to assign to each m the data value that made the corresponding join pass. We conclude the argument by observing that there are only finitely (but exponentially) many $(\varphi_+)^\alpha$ up to satisfiability, hence $\mathsf{or}_\alpha(\varphi_+)^\alpha$ is well-defined and equisatisfiable with φ_+.

As for variables, the encoding proposed here is not polynomial, but the added feature does not bring any complexity jump. We can actually use fragment-specific encodings that avoid the explicit constants of $(\varphi)^\alpha$: this can be achieved either by using second-order variables when available or similarly, in CoreXPath 1.0, thanks to the ability to have multiple propositional variables satisfied at a node in the decision procedure.

Lemma F.1. *For any t, p, φ_+ and α we have that $t, p \models (\varphi_+)^\alpha$ implies $t, p \models \varphi_+$ (and similarly for path expressions).*

Proof. This follows easily by induction on φ_+. Consider for instance the case where $\varphi_+ = \pi_+ \text{ ne}_m \pi'_+$, we have $t, p, q \models (\pi_+)^\alpha$ and $t, p, q' \models (\pi'_+)^\alpha$ with $\delta(q) \sim \alpha(m)$ and $\alpha(m) \nsim \delta(q')$. By induction hypothesis we have we have $t, p, q \models \pi_+$ and $t, p, q' \models \pi'_+$, and the data has not changed, which allows us to conclude. By construction of the φ_+ fragment, data joins can only occur under "positive" constructs, hence all other cases go well. For instance, consider (in fragment where it is relevant) the case where $\varphi_+ = \pi_+ \text{ except } \pi$. We have $t, p, q \models (\pi_+)^\alpha$ for some q for which $t, p, q \not\models \pi$. By induction hypothesis, we obtain $t, p, q \models \pi_+$ which allows us to conclude. □

We did not use the unicity of marks in φ_+. This comes into play in the next lemma.

Lemma F.2. *For any t, p and φ_+ such that $t, p \models \varphi_+$, there exists α over the marks of φ_+ such that $t, p \models (\varphi_+)^\alpha$ (and similarly for path expressions).*

Proof. We proceed again by induction on expressions. Consider the case where $\varphi_+ = \pi^1_+ \vartriangle_m \pi^2_+$. We have $t, p, q_1 \models \pi^1_+$ and $t, p, q_2 \models \pi^2_+$, with $\delta(q_1)$ and $\delta(q_2)$ related according to $\vartriangle$. By induction hypotheses we obtain $t, p, q_1 \models (\pi^1_+)^{\alpha_1}$ and $t, p, q_2 \models (\pi^2_+)^{\alpha_2}$. Moreover, α_1 and α_2 have disjoint domains. We set $\alpha = \alpha_1 \uplus \alpha_2 \uplus \{m \mapsto \delta(q_1)\}$ and conclude easily that $t, p \models (\varphi_+)^\alpha$. Other cases are similar. □

We are ready to conclude.

Proposition 5.5. *Positive joins are expressible in CoreXPath 1.0, CoreXPath 2.0 and EMSO^2 XPath.*

Proof. Given an initial query φ_+ in any of these fragments, there are infinitely many $(\varphi_+)^\alpha$, but we shall see that only finitely many of these formulas is enough for our purpose.

Let $D(\varphi_+)$ be the data values occurring in data tests against constants in φ_+. Let M be the finite set of marks that occur in φ_+, and let D_M be a subset of $\mathbb{D}$ of cardinal $|M|$ and disjoint from $D(\varphi_+)$. We claim that if φ_+ is satisfiable, then there is α with images in $D_M \cup D(\varphi_+)$ such that $(\varphi_+)^\alpha$ is satisfiable. Starting with a model t' of φ_+, it suffices to take α' provided by the previous proposition, transform the model t' into t by rename values in the image of α' and outside the desired range, to obtain a suitable t and α.

Given two valuations α and β in $M \to D_M \cup D(\varphi_+)$, define $\alpha \approx \beta$ to hold when

(1) for all $m, m' \in M$, $\alpha(m) \sim \alpha(m')$ iff $\beta(m) \sim \beta(m')$, and
(2) for all $m \in M$, $d \in D(\varphi_+)$, $\alpha(m) \sim d$ iff $\beta(m) \sim d$.

It can be shown that $(\varphi_+)^\alpha$ and $(\varphi_+)^\beta$ are equisatisfiable whenever $\alpha \approx \beta$. Since $\approx$ has only finitely many equivalence classes, the formula $\mathsf{or}_\alpha(\varphi_+)^\alpha$ is well-defined and equisatisfiable with φ_+. □

G EXPRESSIVENESS RESULTS ON LAST()

G.1 Last() is not Expressible in the Vertical Fragment

Proposition 5.6. *The expression* `last(ancestor::`a`) [child::`b`]` *is not expressible in VerticalXPath.*

We prove that `last()` is not expressible in general in VerticalXPath. To do so, we focus on proving the non expressivity of the query

$$\texttt{last(ancestor::}a\texttt{)[child::}b\texttt{]} \tag{18}$$

If this query could be expressed by a formula φ in VerticalXPath, φ could be assumed not to contain any data test, since the evaluation of the query (18) on a data tree does not depend on the tree's data.

Hence, to study this problem, we can forget about the data in our models, and we will look at a small fragment of CoreXPath 1.0 containing only the vertical axes, noted $\mathrm{XPath}(\downarrow, \downarrow^{-1}, \downarrow^+, (\downarrow^{-1})^+)$ and defined by the abstract syntax

$$\begin{aligned}
\alpha &::= \texttt{child} \mid \texttt{descendant} \mid \texttt{parent} \mid \texttt{ancestor} \\
\pi &::= \alpha\texttt{::*} \mid \pi/\pi \mid \texttt{.}[\varphi] \\
\varphi &::= \pi \mid a \mid \texttt{false()} \mid \texttt{not(}\varphi\texttt{)} \mid \varphi \ \texttt{or} \ \varphi
\end{aligned}$$

To study whether `last()` is expressible in this fragment, we will define and use data-free *bisimulations*. These bisimulations can be seen as Ehrenfeucht-Fraïssé games. Let ℓ, ℓ' be two labelled trees $N \to \Sigma$ (or XML documents for which we will ignore the data), let p (resp. p') be a node from ℓ (resp. ℓ'). Two players take part in the game, called Spoiler and Duplicator. The game starts with a pebble on p and a pebble on p'. If those two nodes are not labelled by the same letter, Spoiler wins the game. A game's step goes as follows:

(1) Spoiler moves one of the pebbles according to an axis α among `child`, `parent`, `descendant`, `ancestor`.
(2) Duplicator must move the other pebble according to the same axis, and on a node labelled by the same letter then the node chosen by Spoiler. If he cannot make such a move, Spoiler wins.

This game corresponds to the following bisimulation:

Definition G.1. Let ℓ and ℓ' be two labelled trees of domains N and N'. Let $Z \subseteq N \times N'$. Z is a *bisimulation* if, for all $p \in N, p' \in N'$ if $p \ Z \ p'$, then:

Harmony p and p' have the same label,
Zig for all $p \star y$, there exists $p' \star y'$ such that $y \ Z \ y'$ (for $\star \in \{\downarrow, \downarrow^{-1}, \downarrow^+, (\downarrow^{-1})^+\}$), and
Zag for all $p' \star y'$, there exists $p \star y$ such that $y \ Z \ y'$ (for $\star \in \{\downarrow, \downarrow^{-1}, \downarrow^+, (\downarrow^{-1})^+\}$)

Theorem G.2. *Let ℓ and ℓ' be two labelled trees, and let p (resp. p') be a node from ℓ (resp. ℓ'). If the nodes p and p' are bisimilar, then p and p' are logically equivalent for XPath$(\downarrow, \downarrow^{-1}, \downarrow^+, (\downarrow^{-1})^+)$.*

The version of the game restricted to n rounds corresponds to this definition of n-bisimulation:

Definition G.3. Let ℓ and ℓ' be two labelled trees. Let $(Z_i)_{i \le n}$ be a sequence of relations between N and N'. For all j, $(Z_i)_{i \le j}$ is a *j-bisimulation* if, for all $p \in N, p' \in N'$ if $p \ Z_j \ p'$, then

Harmony p and p' have the same label,
Zig for all $p \star y$, there exists $p' \star y'$ such that $y \ Z_{j-1} \ y'$ and $(Z_i)_{i \le j-1}$ is a $(j-1)$-bisimulation (for $\star \in \{\downarrow, \downarrow^{-1}, \downarrow^+, (\downarrow^{-1})^+\}$), and
Zag for all $p' \star y'$, there exists $p \star y$ such that $y \ Z_{j-1} \ y'$ and $(Z_i)_{i \le j-1}$ is a $(j-1)$-bisimulation (for $\star \in \{\downarrow, \downarrow^{-1}, \downarrow^+, (\downarrow^{-1})^+\}$)

In order to get a result linking n-bisimulation and logical equivalence, we must first define the set of formulas using at most n navigational steps:

Definition G.4. We define a function ns by induction on the formulas of $\mathsf{XPath}(\downarrow, \downarrow^{-1}, \downarrow^{+}, (\downarrow^{-1})^{+})$:

$$\begin{aligned}
ns(a) &\stackrel{\text{def}}{=} 0 \\
ns(\alpha::*) &\stackrel{\text{def}}{=} 1 && \text{(where } \alpha \in \{\texttt{parent}, \texttt{child}, \texttt{ancestor}, \texttt{descendant}\}\text{)} \\
ns(\varphi \texttt{ or } \varphi') &\stackrel{\text{def}}{=} ns(\varphi \texttt{ or } \varphi') \stackrel{\text{def}}{=} \max\{ns(\varphi), ns(\varphi')\} \\
ns(\texttt{not}(\varphi)) &\stackrel{\text{def}}{=} ns(\varphi) \\
ns(.[\varphi]) &\stackrel{\text{def}}{=} ns(\varphi) \\
ns(\pi/\pi') &\stackrel{\text{def}}{=} ns(\pi) + ns(\pi')
\end{aligned}$$

If $ns(\varphi) = n$, we say that φ has n *nested steps*. We note $\mathsf{XPath}(\downarrow, \downarrow^{-1}, \downarrow^{+}, (\downarrow^{-1})^{+})^{n} \stackrel{\text{def}}{=} \{\varphi \in \mathsf{XPath}(\downarrow, \downarrow^{-1}, \downarrow^{+}, (\downarrow^{-1})^{+}) \mid ns(\varphi) \leq n\}$.

Definition G.5. Let ℓ, ℓ' be two labelled trees, and let p (resp. p') be a node of ℓ (resp. ℓ'). We note $p \equiv_n p'$ if p and p' are *logically equivalent* for $\mathsf{XPath}(\downarrow, \downarrow^{-1}, \downarrow^{+}, (\downarrow^{-1})^{+})^{n}$.

Lemma G.6. *Let $n \geq 0$. Let ℓ and ℓ' be two labelled trees, let p (resp. p') be a node of ℓ (resp. ℓ'), and let $(Z_i)_{i \leq n}$ be an n-bisimulation between ℓ and ℓ'. If $p\ Z_j\ p'$ for some $j \leq n$, then $p \equiv_j p'$.*

Proof. We prove this theorem by induction on j:

- $j = 0$: the only formulas are label tests, and 0-bisimulation between two nodes require that the nodes have the same label. So 0-bisimilar nodes are logically equivalent for formulas of 0 nested steps.
- Let us assume that $p\ Z_j\ p'$ and that the theorem is true for i-bisimulations with $i < j$. First, since $p\ Z_j\ p'$ implies $p\ Z_{j-1}\ p'$, then $p \equiv_{j-1} p'$ by induction hypothesis. Now, let us consider a formula $\varphi = \alpha::*/\varphi'$ with $\star$ being the relation corresponding to α. If φ is true on p, then there exists $p \star y$ such that φ' is true at y. Then we choose $p' \star y'$ such that $y\ Z_{j-1}\ y'$ (such a node exists because $(Z_i)_{i \leq j}$ is a j-bisimulation). And now, because $ns(\varphi') = j - 1$, by induction hypothesis, φ' is true at y', so φ is true at p'. Symmetrically, we can prove that if φ is true at p', then it is also true at p. Now for the case $\varphi = .[\varphi']$ we study φ' instead, and the Boolean combinations are handled easily. At the end, we have $p \equiv_j p'$. □

Now, let us assume that the query (18) could be expressed by a formula φ in $\mathsf{XPath}(\downarrow, \downarrow^{-1}, \downarrow^{+}, (\downarrow^{-1})^{+})$. Let $n \stackrel{\text{def}}{=} ns(\varphi)$. Let us show that there exist a tree ℓ_n and two nodes $p, p' \in N_n$ such that p and p' are n-bisimilar, but such that p satisfies the query (18), and p' does not.

The tree ℓ_n and the bisimulations are represented on figures 10 and 11. We define the tree ℓ_n as follows:

- We partition its set N_n of nodes into the subsets
  - $A_n \stackrel{\text{def}}{=} \{a_i \mid i \in [-2n+1, 2n-1]\}$
  - $B_n \stackrel{\text{def}}{=} \{b_{2i+1} \mid i \in [-n, n-1]\}$
  - $C_n^i \stackrel{\text{def}}{=} \{c_{i,k} \mid k \in [-n, n]\}$ for $i \in [-2n+1, 2n-2]$
- The relation $\downarrow$ holds between the following nodes:
  - $a_{2i+1} \downarrow b_{2i+1}$, for $i \in [-n, n-2]$
  - $c_{i,k} \downarrow c_{i,k-1}$ pour $i \in [-2n+1, 2n-2]$ et $k \in [-n+1, n]$ (the $c_{i,k}$ are forming a chain of length $2n+1$)
  - $b_{2i+1} \downarrow c_{2i,n}$ et $c_{2i,-n} \downarrow a_{2i}$ for $i \in [-n+1, n-1]$ (the c chains link the lones a)
  - $a_{2i+2} \downarrow c_{2i+1,n}$ et $c_{2i+1,-n} \downarrow a_{2i+1}$ for $i \in [-n, n-2]$ (the c chains link the groups $a - b$)

We now describe the bisimulation relations $(Z_i)_{i \leq n}$ between the nodes of ℓ_n:

- $c_{-1,0}\ Z_n\ c_{0,0}$ (they are the starting nodes of our n rounds game)
- Duplicator must be able to simulate small moves of Spoiler locally around the starting nodes:
  - $\forall i \in [1, n],\ c_{-1,i}\ Z_{n-i}\ c_{0,i}$
  - $\forall i \in [1, n],\ c_{-1,-i}\ Z_{n-i}\ c_{0,-i}$
- Duplicator can simulate bigger moves from Spoiler by doing a shift:
  - $\forall i \in [0, n-2],\ a_{2i}\ Z_{n-1-i}\ a_{2i+2}$
  - $\forall i \in [0, n-2],\ a_{2i+1}\ Z_{n-2-i}\ a_{2i+3}$
  - $\forall i \in [-n+1, -1],\ a_{2i}\ Z_{n+i}\ a_{2i+2}$
  - $\forall i \in [-n, -1],\ a_{2i+1}\ Z_{n+i}\ a_{2i+3}$
  - $\forall i \in [0, n-2],\ c_{2i-1,k}\ Z_{n-1-i}\ c_{2i+1,k}$
  - $\forall i \in [0, n-2],\ c_{2i,k}\ Z_{n-1-i}\ c_{2i+2,k}$
  - $\forall i \in [-n+1, -1],\ c_{2i-1,k}\ Z_{n+i}\ c_{2i+1,k}$

  - $\forall i \in [-n+1, -1],\ c_{2i,k}\ Z_{n+i}\ c_{2i+2,k}$
- Duplicator can simulate too big moves from Spoiler by using the identity relation: $\forall p \in N_n,\ p\ Z_n\ p$
- Finally, every relation must contain the smaller ones:
  $\forall (p, p') \in N_n^2, \forall i < j$, if $p\ Z_j\ p'$ then $p\ Z_i\ p'$.

Lemma G.7. *$(Z_i)_{i \leq n}$ is an n-bisimulation.*

Proof. Let us prove by induction on j that $(Z_i)_{i \leq j}$ is a j-bisimulation.

- $j = 0$: every nodes in relation for Z_0 do have the same labels, so Z_0 is a 0-bisimulation.
- Let us assume that $(Z_i)_{i \leq j}$ is a j-bisimulation for some $j < n$, and let us prove that $(Z_i)_{i \leq j+1}$ is a $(j+1)$-bisimulation. Let p and p' be two nodes from ℓ_n such that $p\ Z_{j+1}\ p'$. Then:
  (1) By definition of Z_{j+1}, p and p' have the same label.
  (2) Let $\star \in \{\downarrow^{-1}, \downarrow, (\downarrow^{-1})^+, \downarrow^+\}$. We must prove that for every $p \star y$, there exists a $p' \star y'$ such that $y\ Z_j\ y'$. The cases $\star = \downarrow^{-1}$ and $\star = \downarrow$ are easy because we padded ℓ_n with long enough c-chains. Let us focus on the case $\star = (\downarrow^{-1})^+$ (the case $\star = \downarrow^+$ is similar): let $y \in N_n$ such that $p(\downarrow^{-1})^+ y$. Now, if $p'(\downarrow^{-1})^+ y$ then we can take $y' = y$ and use the identity relation which is contained in Z_j. Else, by choosing y' above y with the same distance between p and y than between p' and y', we will almost always get a node y' such that $y\ Z_j\ y'$. The only case where this won't work is if $p = c_{-1,k}$, $p' = c_{0,k}$ for some k. In that case, if y is at distance d of p, we will choose y' at distance $d + 2n + 3$ of p' instead. This is because the distance between bisimilar nodes around the starting points is smaller than elsewhere in the tree, so this bigger jump is required to close the gap.
  (3) Similarly, we can prove that for every $\star \in \{\downarrow^{-1}, \downarrow, (\downarrow^{-1})^+, \downarrow^+\}$, for every $p' \star y'$, there exists a $p \star y$ such that $y\ Z_j\ y'$. □

This contradicts the fact that φ was equivalent to the query (18), since $c_{-1,0}\ Z_n\ c_{0,0}$ but only $c_{0,0}$ satisfies the query (18). Hence, this query is not expressible in XPath$(\downarrow, \downarrow^{-1}, \downarrow^+, (\downarrow^{-1})^+)$, and so it is not expressible in VerticalXPath.

G.2 Expressing Last()

G.2.1 Expressing Last() in RegularXPath. In this section, we show how to express $\texttt{last}(\pi)$ for *one-step* paths of the form $\pi = \alpha{::}*[\varphi]$. In some cases, it is easily expressible:

$$\begin{aligned}
\texttt{last}(\texttt{parent::}*[\varphi]) &\equiv \texttt{parent::}*[\varphi] \\
\texttt{last}(\texttt{self::}*[\varphi]) &\equiv \texttt{self::}*[\varphi] \\
\texttt{last}(\texttt{child::}*[\varphi]) &\equiv \texttt{child::}*[\varphi \texttt{ and not}(\texttt{following-sibling::}*[\varphi])] \\
\texttt{last}(\texttt{following-sibling::}*[\varphi]) &\equiv \texttt{following-sibling::}*[\varphi \texttt{ and } \texttt{not}(\texttt{following-sibling::}*[\varphi])] \\
\texttt{last}(\texttt{following::}*[\varphi]) &\equiv \texttt{following::}*[\varphi \texttt{ and not}(\texttt{following::}*[\varphi]) \\
&\qquad \texttt{and not}(\texttt{descendant::}*[\varphi])]
\end{aligned}$$

The other cases can be handled by using RegularXPath:

$$\begin{aligned}
\texttt{last}(\texttt{ancestor::}*[\varphi]) &\equiv (\texttt{parent::}*[\texttt{not}(\varphi)])^*/\texttt{parent::}*[\varphi] \\
\texttt{last}(\texttt{descendant::}*[\varphi]) &\equiv (\texttt{child::}*[\texttt{descendant-or-self::}*[\varphi] \texttt{ and} \\
&\qquad \texttt{not}(\texttt{following-sibling::}*/\texttt{descendant-or-self::}*[\varphi])])^+ \\
&\qquad [\varphi \texttt{ and not}(\texttt{descendant::}*[\varphi])] \\
\texttt{last}(\texttt{preceding-sibling::}*[\varphi]) &\equiv (\texttt{previous-sibling::}*[\texttt{not}(\varphi)])^*/\texttt{previous-sibling::}*[\varphi] \\
\texttt{last}(\texttt{preceding::}*[\varphi]) &\equiv (\texttt{parent::}*[\texttt{not}((\texttt{previous-sibling::}*)^+/(\texttt{child::}*)^*[\varphi])])^*/ \\
&\qquad (\texttt{previous-sibling::}*[\texttt{not}((\texttt{child::}*)^*[\varphi])])^+/ \\
&\qquad\quad (\texttt{child::}*[\texttt{descendant-or-self::}*[\varphi] \texttt{ and} \\
&\qquad\quad \texttt{not}(\texttt{following-sibling::}*/\texttt{descendant-or-self::}*[\varphi])])^* \\
&\qquad\quad [\varphi \texttt{ and not}(\texttt{descendant::}*[\varphi])]
\end{aligned}$$

Note that the axis `previous-sibling` used above is not a proper XPath axis, but exists in RegularXPath. It corresponds to the relation $\rightarrow^{-1}$ of our models.

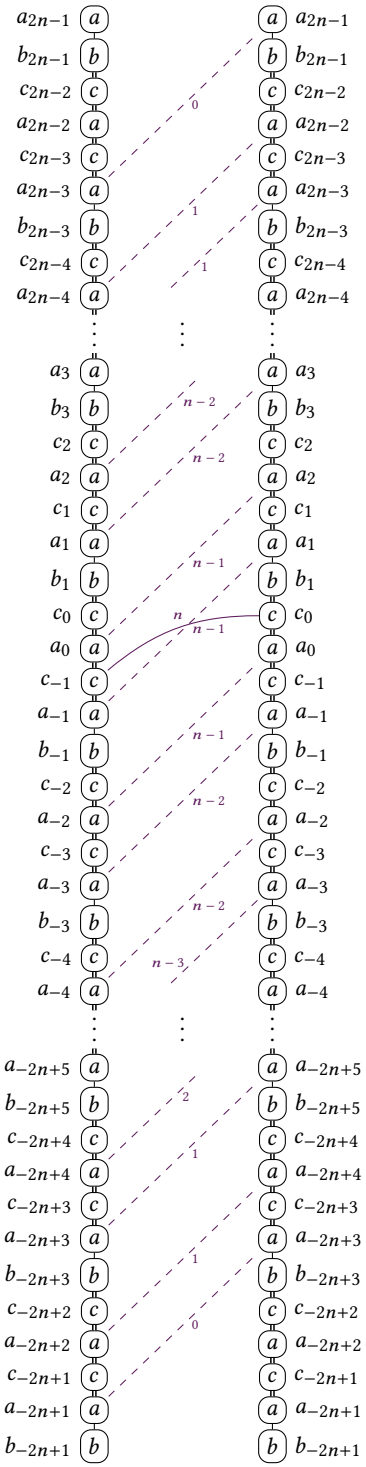

Figure 10: n-bisimulation inside ℓ_n (a double edge represents a chain of n c nodes)

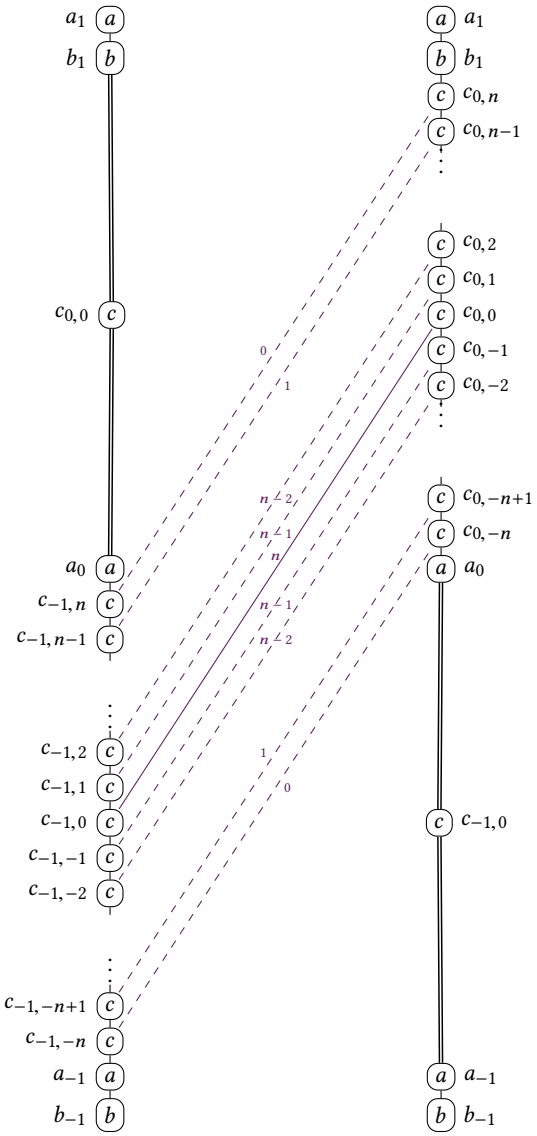

Figure 11: Zoom around the starting points

G.2.2 Expressing Notlast() in Core XPath. In this section, we show how to express queries of the form $notlast(\alpha{::}*[\varphi])$ in CoreXPath 1.0.

$$
\begin{aligned}
notlast(\texttt{parent::}*[\varphi]) &\equiv \texttt{false()}\\
notlast(\texttt{self::}*[\varphi]) &\equiv \texttt{false()}\\
notlast(\texttt{child::}*[\varphi]) &\equiv \texttt{child::}*[\varphi\ \texttt{and following-sibling::}*[\varphi]]\\
notlast(\texttt{following-sibling::}*[\varphi]) &\equiv \texttt{following-sibling::}*[\varphi\ \texttt{and following-sibling::}*[\varphi]]\\
notlast(\texttt{following::}*[\varphi]) &\equiv \texttt{following::}*[\varphi\ \texttt{and (following::}*[\varphi]\ \texttt{or descendant::}*[\varphi])]\\
notlast(\texttt{ancestor::}*[\varphi]) &\equiv \texttt{ancestor::}*[\varphi]\texttt{/ancestor::}*[\varphi]\\
notlast(\texttt{preceding-sibling::}*[\varphi]) &\equiv \texttt{preceding-sibling::}*[\varphi]\texttt{/preceding-sibling::}*[\varphi]\\
notlast(\texttt{descendant::}*[\varphi]) &\equiv \texttt{descendant::}*[\varphi\ \texttt{and descendant::}*[\varphi]]\ \texttt{union}\\
&\qquad \texttt{descendant::}*[\\
&\qquad\quad \texttt{following-sibling::}*\texttt{/descendant-or-self::}*[\varphi]\\
&\qquad]\texttt{/descendant-or-self::}*[\varphi]\\
notlast(\texttt{preceding::}*[\varphi]) &\equiv \texttt{preceding::}*[\varphi]\texttt{/preceding::}*[\varphi]\ \texttt{union}\\
&\qquad \texttt{preceding::}*[\varphi\ \texttt{and descendant::}*[\varphi]]
\end{aligned}
$$

G.3 Expressing Position() in Regular XPath

In this section, we show how to translate predicates of the form $\alpha{::}*[\varphi][\texttt{position() = i}]$ in RegularXPath. This query selects the i-th node for the document order among the nodes that would have been selected by $\alpha{::}*[\varphi]$. We represent such a predicate by a function pos_i. We give a translation for pos_i by induction on i.

$$
\begin{aligned}
pos_1(\texttt{parent::}*[\varphi]) &\equiv \texttt{parent::}*[\varphi]\\
pos_1(\texttt{self::}*[\varphi]) &\equiv \texttt{self::}*[\varphi]\\
pos_1(\texttt{child::}*[\varphi]) &\equiv \texttt{child::}*[\varphi\ \texttt{and not (preceding-sibling::}*[\varphi])]\\
pos_1(\texttt{preceding-sibling::}*[\varphi]) &\equiv \texttt{preceding-sibling::}*[\varphi\ \texttt{and not (preceding-sibling::}*[\varphi])]\\
pos_1(\texttt{following-sibling::}*[\varphi]) &\equiv (\texttt{next-sibling::}*[\texttt{not}(\varphi)])^*\texttt{/next-sibling::}*[\varphi]\\
pos_1(\texttt{ancestor::}*[\varphi]) &\equiv \texttt{ancestor::}*[\varphi\ \texttt{and not (ancestor::}*[\varphi])]\\
pos_1(\texttt{descendant::}*[\varphi]) &\equiv (\texttt{child::}*[\varphi\ \texttt{and descendant::}*[\varphi]\ \texttt{and}\\
&\qquad \texttt{not}(\texttt{preceding-sibling::}*\texttt{/descendant-or-self::}*[\varphi])\\
&\quad])^*\texttt{/child::}*[\varphi\ \texttt{and not}(\texttt{preceding-sibling::}*\texttt{/}\\
&\quad \texttt{descendant-or-self::}*[\varphi])]\\
pos_1(\texttt{following::}*[\varphi]) &\equiv \texttt{ancestor-or-self::}*[\texttt{following::}*[\varphi]\ \texttt{and}\\
&\qquad \texttt{not}(\texttt{parent::}*\texttt{/following::}*[\varphi])]\texttt{/}\\
&\quad pos_1(\texttt{following-sibling::}*[\texttt{descendant-or-self::}*[\varphi]])\texttt{/}\\
&\quad (.[\varphi]\ \texttt{union}\ .[\texttt{not}(\varphi)]/pos_1(\texttt{descendant::}*[\varphi]))\\
pos_1(\texttt{preceding::}*[\varphi]) &\equiv \texttt{ancestor-or-self::}*[\texttt{preceding::}*[\varphi]\ \texttt{and}\\
&\qquad \texttt{not}(\texttt{parent::}*\texttt{/preceding::}*[\varphi])]\texttt{/}\\
&\quad pos_1(\texttt{preceding-sibling::}*[\texttt{descendant-or-self::}*[\varphi]])\texttt{/}\\
&\quad (.[\varphi]\ \texttt{union}\ .[\texttt{not}(\varphi)]/pos_1(\texttt{descendant::}*[\varphi]))
\end{aligned}
$$

$$
\begin{aligned}
pos_{i+1}(\texttt{parent::}*[\varphi]) &\equiv \texttt{false()}\\
pos_{i+1}(\texttt{self::}*[\varphi]) &\equiv \texttt{false()}\\
pos_{i+1}(\texttt{child::}*[\varphi]) &\equiv pos_{\texttt{i}}(\texttt{child::}*[\varphi])/pos_{\texttt{1}}(\texttt{following-sibling::}*[\varphi])\\
pos_{i+1}(\texttt{following-sibling::}*[\varphi]) &\equiv pos_{\texttt{i}}(\texttt{following-sibling::}*[\varphi])/\\
&\qquad pos_{\texttt{1}}(\texttt{following-sibling::}*[\varphi])\\
pos_{i+1}(\texttt{preceding-sibling::}*[\varphi]) &\equiv pos_{\texttt{1}}(\texttt{preceding-sibling::}*[\varphi\ \texttt{and}\\
&\qquad pos_{\texttt{i}}(\texttt{preceding-sibling::}*[\varphi])])\\
pos_{i+1}(\texttt{ancestor::}*[\varphi]) &\equiv pos_{\texttt{1}}(\texttt{ancestor::}*[\varphi\ \texttt{and}\\
&\qquad pos_{\texttt{i}}(\texttt{ancestor::}*[\varphi])])\\
pos_{i+1}(\texttt{descendant::}*[\varphi]) &\equiv pos_{\texttt{i}}(\texttt{descendant::}*[\varphi])[\texttt{descendant::}*[\varphi]]/\\
&\qquad pos_{\texttt{1}}(\texttt{descendant::}*[\varphi])\ \texttt{union}\\
&\qquad pos_{\texttt{i}}(\texttt{descendant::}*[\varphi])[\texttt{not(descendant::}*[\varphi])]/\\
&\qquad pos_{\texttt{1}}(\texttt{following::}*[\varphi])\\
pos_{i+1}(\texttt{following::}*[\varphi]) &\equiv pos_{\texttt{i}}(\texttt{following::}*[\varphi])[\texttt{descendant::}*[\varphi]]/\\
&\qquad pos_{\texttt{1}}(\texttt{descendant::}*[\varphi])\ \texttt{union}\\
&\qquad pos_{\texttt{i}}(\texttt{following::}*[\varphi])[\texttt{not(descendant::}*[\varphi])]/\\
&\qquad pos_{\texttt{1}}(\texttt{following::}*[\varphi])\\
pos_{i+1}(\texttt{preceding::}*[\varphi]) &\equiv \Pi_i\ \texttt{union}\ .[\texttt{not(}_{\texttt{i}}\texttt{)}]/pos_{\texttt{1}}(\texttt{preceding::}*[\varphi\ \texttt{and}\\
&\qquad pos_{\texttt{i}}(\texttt{preceding::}*[\varphi])])
\end{aligned}
$$

With :

$$\Pi_i \overset{\text{def}}{=} \underset{1\leq j\leq i}{\texttt{union}}\ pos_j(\texttt{preceding::}*[\varphi])/pos_{i+1-j}(\texttt{descendant::}*[\varphi])$$

H DECIDABILITY RESULTS ON ID()

H.1 Reducing from PCP Using Data Joins

In this section, we show that the satisfiability problem is undecidable for DownwardXPath+id. This will entail undecidability for ForwardXPath+id and VerticalXPath+id as well.

Let Σ be an alphabet, and let $\{(u_i, v_i) \mid 0 \leq i \leq n\} \subseteq (\Sigma^+)^2$ be a PCP problem over Σ. We note $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$, and we give ourselves a fresh symbol # that will mark the start of a PCP domino. We will encode a potential solution to this PCP problem as a branch in a data tree over $\Sigma \cup \overline{\Sigma}$ for which the corresponding word will belong to $\{\#u_i\overline{v_i} \mid 0 \leq i \leq n\}^+$. Since we are working with the DownwardXPath fragment, we cannot prevent the models from having other branches, but we will be able to force all the branches to be identical.

We will use the id links to move around in this encoding: let $\tilde{w}$ be the word corresponding to our encoding branch of the tree, and let w (resp. $\overline{w}$) be the longest subword of $\tilde{w}$ in Σ (resp. $\overline{\Sigma}$). Every node w_i will have an attribute @succ for which the datum will match the @id key of the node w_{i+1}, and an attribute @sym for which the datum will match the @id key of the node $\overline{w}_i$. The nodes $\overline{w}_i$ will also have attributes @succ and @sym playing similar roles. Such a word $\tilde{w}$ will be the encoding of a solution of our PCP problem if and only if $w = \overline{w}$.

Hence, we need to find formulas that will force this encoding to be respected. At the end, a data tree will satisfy this set of formulas if and only if it encodes a solution to our PCP problem.

Let's start by forcing all the branches from the root to be identical:

- Ψ_1 := `not(//*[@id eq descendant::*/@id])` (the @id keys are unique along a branch).
- Ψ_2 := $\texttt{and}_{a\neq b\in\Sigma\cup\overline{\Sigma}\cup\{\#\}}$ `not(//*[id(@id)/child::a and id(@id)/child::b])` (if two nodes share the same @id key, then all their children are labelled by the same letter).
- Ψ_3 := `not(//*[id(@id)/child::*/@id ne id(@id)/child::*/@id])` (if two nodes share the same @id key, then all their children have the same @id key).
- Ψ_4 := `not(//*[(id(@id)/@sym ne id(@id)/@sym) or (id(@id)/@succ ne id(@id)/@succ)])` (if two nodes share the same @id key, then they have the same @sym data value and the same @succ data value).

- Ψ_5 := `not(//*[id(@id)[child::*] and id(@id)[not(child::*)]])` (a node has a child iff all the nodes sharing its `@id` key have a child).

Now, we can prove the following lemma:

Lemma H.1. *Let t be a data tree. Then, t satisfies all the Ψ_i formulas if and only if:*

(1) *all the nodes of t at the same depth level are identical (same label, same* `@id`, `@succ` *and* `@sym` *values) and have children with the same label and the same* `@id` *key.*
(2) *Two nodes from different depth levels do not share the same* `@id` *key.*

Proof. It is easy to see that a data tree satisfying the properties (1) and (2) will satisfy the formulas Ψ_i. Suppose now that a data tree t satisfies the formulas Ψ_i. We prove that the properties (1) and (2) are satisfied by induction on the depth level:

- Level depth 0: there is only the root node at this depth level. Thanks to the formula Ψ_1, and since the root node appears in every branch of the tree, there is no node somewhere else in the tree with the same `@id` key than the root so the property (2) is satisfied. And now, thanks to Ψ_2, all its children have the same label, and thanks to Ψ_3, all its children have the same `@id` key. So the property (1) is also satisfied.
- Let's assume that the properties (1) and (2) are satisfied down to depth level n, and let's prove they are also satisfied at depth level $n+1$. Since the nodes of depth n satisfy the property (1), then all the nodes of depth $n+1$ have the same label and the same `@id` key (they are children from nodes of depth n). Then, thanks to Ψ_4, they also have the same `@succ` and `@sym` values, so they are identical. Moreover, thanks to Ψ_2 and Ψ_3, all their children have the same label and `@id` value (because of the nodes of depth $n+1$ have the same `@id` key). So the property (1) is satisfied at depth level $n+1$. Furthermore, thanks to Ψ_1, no node above or below a node of depth $n+1$ (that is, no node from a different depth level) share the same `@id` key than the nodes of depth $n+1$, so the property (2) is also satisfied.

Thus, by induction on the depth level, the properties (1) and (2) are satisfied everywhere in the tree. □

This property allows us to only consider non-branching models, as any model will be logically equivalent to one of its branch for DownwardXPath with id.

Then, we start by giving us formulas expressing simple properties:

- φ_Σ := `self::`a_1 `union ... union self::`a_n, where $\Sigma = \{a_1, \dots a_n\}$ (checks that the label of the current node is in Σ).
- Symmetrically, we define the formula $\varphi_{\overline{\Sigma}}$.
- φ_{first} := `self::#` (checks that the current node is a marker at the beginning of a group $u_i\overline{v_i}$).
- φ_{last} := `not(child::*) or child::#` (checks that the current node is the last letter of a group $u_i\overline{v_i}$).

We can now express the marker property of the # symbol:

$$\Phi_1 := \texttt{not(//*[}\varphi_\Sigma\texttt{]/child::\#) and not(//\#/child::*[}\varphi_{\overline{\Sigma}}\texttt{]) and not(//\#/\#)}$$

Moreover, we can express that a given word appears correctly between markers, somewhere in the tree. Let $u_i\overline{v_i} = c_{i,0} \dots c_{i,m_i}$. We define the formula φ_i that checks that the current node is a marker #, followed by letters forming the word $u_i\overline{v_i}$, and that this word isn't followed by extra letters:

$$\varphi_i := \varphi_{first} \texttt{ and ./}c_{i,0}\texttt{/}c_{i,1}\texttt{/ . . . /(}c_{i,m_i}\texttt{[}\varphi_{last}\texttt{])}$$

Thus, we can express the fact that the word w corresponding to any branch of a model is in $\{\#u_i\overline{v_i} \mid 0 \leq i \leq n\}^+$:

$$\Phi_2 := \texttt{/*[}\varphi_{first}\texttt{] and not(//*[}\varphi_{first}\texttt{ and not(}\varphi_0\texttt{) and not ... and not(}\varphi_n\texttt{)])}$$

Then, we must express the encoding properties about the `id` links:

- Φ_3 := `not(//*[not(@succ) and ((`φ_Σ `and descendant::*[`φ_Σ`]) or (`$\varphi_{\overline{\Sigma}}$ `and descendant[`$\varphi_{\overline{\Sigma}}$`]))]) and not(//*[@succ and ((`$\varphi_\Sigma$ `and not(descendant::*[`φ_Σ`])) or (`$\varphi_{\overline{\Sigma}}$ `and not(descendant[`$\varphi_{\overline{\Sigma}}$`])))])` (every letter has a successor except the last letter of the longest subwords from Σ and $\overline{\Sigma}$).
- Φ_4 := `not(//*[not(@sym) and not(self::#)])` (every letter has a `@sym` attribute).
- Φ_5 := `not(//*[@succ and not(@succ eq descendant::*/@id)])` (`id(@succ)` is non decreasing).
- Φ_6 := `not(//*[(`φ_Σ `and id(@succ)[`$\varphi_{\overline{\Sigma}}$`]) or (`$\varphi_{\overline{\Sigma}}$ `and id(@succ)[`φ_Σ`])])`
  (`id(@succ)` doesn't link letters from Σ with letters from $\overline{\Sigma}$).
- Φ_7 := `not(//*[`φ_Σ `and @succ eq descendant::*[`φ_Σ`]/descendant::*/@id]) and not(//*[`$\varphi_{\overline{\Sigma}}$ `and @succ eq descendant::*[`$\varphi_{\overline{\Sigma}}$`]/descendant::*/@id])`
  (`id(@succ)` doesn't jump over a letter from the same alphabet).

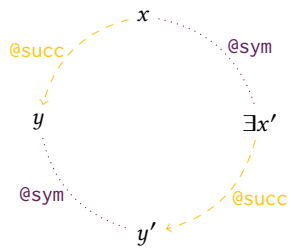

Figure 12: Φ_{10} express the compatibility between `id(@succ)` and `id(@sym)`: if x has a successor y, then the symmetric of x must also have a successor, which is the symmetric of y.

- Φ_8 := $\mathsf{and}_{a\in\Sigma}$(`not(//*[self::`$a$ `and not(id(@sym)/self::`$\overline{a}$`)])` `and` `not(//*[self::`$\overline{a}$ `and not(id(@sym)/self::`a`)])`)
  (`id(@sym)` links properly a letter to its bar version).
- Φ_9 := `not(//*[not(id(sym)/@sym eq @id)])` (`id(@sym)` is an involution).
- Φ_{10} := `not(//*[@succ and not((id(@succ)/@sym eq id(@sym)/@succ))])` (compatibility of `id(@succ)` and `id(@sym)`, cf. Fig. 12).

It is easy to check that if the PCP problem has a solution, encoding it as described above will give us a data tree satisfying all the formulas. Conversely, let's prove that if all these formulas are satisfied by a data tree t, then we can extract a solution of our PCP problem from t.

First, thanks to lemma H.1, we can assume for the sake of simplicity that t is a non-branching tree. Then, because t satisfies Φ_1 and Φ_2, the word $\tilde{w}$ of t belongs to $\{\#u_i\overline{v_i}\}^+$. Let's call w (resp. $\overline{w}$) the longest subword of t in Σ (resp. $\overline{\Sigma}$). Now, thanks to Φ_3 every letter of w and $\overline{w}$ has a `@succ` attribute (except their last letter), and thanks to Φ_4 every letter of w and $\overline{w}$ has a `@sym` attribute. Moreover, because t satisfies Φ_5, Φ_6 and Φ_7, we can prove that the `@succ` links are jumping as intended from a letter of w (resp. $\overline{w}$) to the next one. This give us two `@succ` chains which correspond to w and $\overline{w}$. We note $x\ \mathcal{R}_{succ}\ y$ if there is a `@succ` link from x to y.

We now need to check the `@sym` links. These links are forming a symmetric relation (thanks to Φ_9) that we will denote by $\mathcal{R}_{sym}$. First, let's consider the beginning of w: let j be the position in $\overline{w}$ such that $w_0\ \mathcal{R}_{sym}\ \overline{w}_j$, and let's assume that $j > 0$. Then, there is a node $\overline{w}_{j-1}$ above in t such that $\overline{w}_{j-1}\ \mathcal{R}_{succ}\ \overline{w}_j$. Now, because of the compatibility property expressed by Φ_{10}, there should be a node x above w_0 in the tree such that $\overline{w}_{j-1}\ \mathcal{R}_{sym}\ x\ \mathcal{R}_{succ}\ w_0$, which isn't possible by definition of w_0. So we have $w_0\ \mathcal{R}_{sym}\ \overline{w}_0$. Now, thanks to the same compatibility property, we can prove by induction on i that we have $w_i\ \mathcal{R}_{sym}\ \overline{w}_i$ for every i.

What could still happen is that w and $\overline{w}$ are not of the same length. This is also forbidden by Φ_{10}: let w_{n-1} be the last letter of w, and let's assume that $\overline{w}$ is of length $\geq n$ (the other case is symmetric). Then we have $w_{n-1}\ \mathcal{R}_{sym}\ \overline{w}_{n-1}$, and now if $\overline{w}_{n-1}$ had a successor $\overline{w}_n$, then by using Φ_{10} we could prove that there should be a node x such that $w_{n-1}\ \mathcal{R}_{succ}\ x\ \mathcal{R}_{sym}\ \overline{w}_n$, which isn't possible by assumption that w_{n-1} was the last letter of w. So w and $\overline{w}$ are of the same length.

Finally, thanks to Φ_8 we can prove that for $0 \leq i < n$, $\overline{(w_i)} = \overline{w}_i$, so t does represent a solution to our PCP problem.

Example H.2. Let's consider the following PCP problem, where $\Sigma = \{a, b\}$.
$(u_1, \bar{v_1}) = (a, b\bar{a}a)$, $(u_2, \bar{v_2}) = (ab, \bar{a}a)$, $(u_3, \bar{v_3}) = (bba, \bar{b}\bar{b})$. The sequence $(3, 2, 3, 1)$ is a solution, and its encoding is represented in Fig. 13.

H.2 Reducing from PCP using Node Tests

The encoding above can be easily adapted for a restricted version of CoreXPath 2.0 with only the downward axes, by replacing the data joins by node tests using `is`. Here is how one can transform the previous formulas to be in such a fragment:

(1) Since every data value is actually the `@id` key of a node, every data test of the form π`/@foo eq` π' can be transformed into π`/id(@foo)/@id eq` π'.
(2) Once all the data joins are of the form π`/@id eq` π'`/@id`, we can replace them by node tests of the form π `is` π' (since all the `@id` keys are unique).

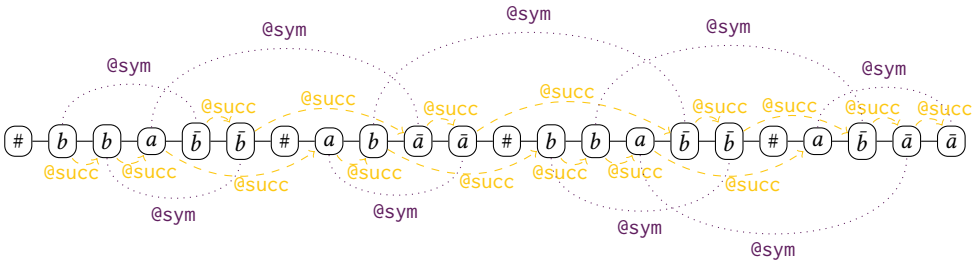

Figure 13: Example of the encoding of a PCP solution.

H.3 Decidable Fragments with `id`

In this section, we show how we can encode some use of `id` by using variables axiomatised with data joins. Let us recall what uses of `id` are allowed:

$$\pi ::= \cdots \mid /\pi_{id}$$
$$\pi_{id} ::= \pi \mid \pi_{id}/\pi_{id} \mid \pi_{id} \texttt{ union } \pi_{id} \mid \pi_{id}[\varphi] \mid \texttt{id}(\pi_{id})$$

This extension concerns VerticalXPath extended with variables, and EMSO^{2}XPath.

When an occurrence of `id()` is applied at the end of an absolute path $/\Pi$, it is translated into a variable $\$x_\Pi$ that will be forced to contain exactly the nodes to which the `id()` call would jump, using the following axioms:

- $\varphi_1(\$x_\Pi, \Pi) := \texttt{not(/[(not(. eq \$x/@id)) and . eq // * /@id])}$
- $\varphi_1(\$x_\Pi, \Pi) := \texttt{not(\$x[not(@id eq /)])}$

We will eliminate the `id` occurrences by using a translation function $\mathsf{T}_{\texttt{id}}$ taking two arguments: the first one is a residual path π, not containing any `id()` call, that will be used to axiomatise variables $\$x_\pi$, and the second one is the query that remains to be translated. In the following translation, whenever a formula denoted $\$x_\Pi$ is used, the corresponding axioms $\varphi_1(\$x_\Pi, \Pi)$ and $\varphi_2(\$x_\Pi, \Pi)$ will be added at top level.

$$\begin{aligned}
\mathsf{T}_{\texttt{id}}(\pi, \pi'/\pi_{id}) &= \mathsf{T}_{\texttt{id}}(\pi/\pi', \pi_{id}) && \text{(where } \pi\text{' does not contain any id() call)}\\
\mathsf{T}_{\texttt{id}}(\pi, \pi_{id}/\pi'_{id}) &= \mathsf{T}_{\texttt{id}}(\mathsf{T}_{\texttt{id}}(\pi, \pi_{id}), \pi'_{id})\\
\mathsf{T}_{\texttt{id}}(\pi, \texttt{id}(\pi')) &= \$x_\Pi && \text{(where } \$x_\Pi \text{ is fresh, and } \Pi = \pi/\pi')\\
\mathsf{T}_{\texttt{id}}(\pi, \texttt{id}(\pi_{id})) &= \mathsf{T}_{\texttt{id}}(\varepsilon, \texttt{id}(\mathsf{T}_{\texttt{id}}(\pi, \pi_{id})))\\
\mathsf{T}_{\texttt{id}}(\pi, \pi') &= \pi/\pi' && \text{(where } \pi\text{' does not contain any id() call)}\\
\mathsf{T}_{\texttt{id}}(\pi, \pi_{id} \texttt{ union } \pi'_{id}) &= \mathsf{T}_{\texttt{id}}(\pi, \pi_{id}) \texttt{ union } \mathsf{T}_{\texttt{id}}(\pi, \pi'_{id})\\
\mathsf{T}_{\texttt{id}}(\pi, \pi'[\varphi]) &= \mathsf{T}_{\texttt{id}}(\pi, \pi')[\mathsf{T}_{\texttt{id}}(\varepsilon, \varphi)]\\
\mathsf{T}_{\texttt{id}}(\varepsilon, \varphi \texttt{ and } \varphi') &= \mathsf{T}_{\texttt{id}}(\varepsilon, \varphi) \texttt{ and } \mathsf{T}_{\texttt{id}}(\varepsilon, \varphi')\\
\mathsf{T}_{\texttt{id}}(\varepsilon, \varphi \texttt{ or } \varphi') &= \mathsf{T}_{\texttt{id}}(\varepsilon, \varphi) \texttt{ or } \mathsf{T}_{\texttt{id}}(\varepsilon, \varphi')\\
\mathsf{T}_{\texttt{id}}(\varepsilon, \texttt{not}(\varphi)) &= \texttt{not}(\mathsf{T}_{\texttt{id}}(\varepsilon, \varphi))
\end{aligned}$$

We denote by Ax all the axioms that must be added at top level:

$$\mathrm{Ax}(\mathsf{T}_{\texttt{id}}(\pi, \tilde{\pi})) = \bigwedge_{\$x_\Pi \text{ appearing in } \mathsf{T}_{\texttt{id}}(\pi, \tilde{\pi})} (\varphi_1(\$x_\Pi, \Pi) \wedge \varphi_2(\$x_\Pi, \Pi))$$

And we note $t, \epsilon, n_f \vDash_{Ax} \mathsf{T}_{\texttt{id}}(\pi, \tilde{\pi})$ when $t, \epsilon, n_f \vDash \mathsf{T}_{\texttt{id}}(\pi, \tilde{\pi}) \wedge \mathrm{Ax}(\mathsf{T}_{\texttt{id}}(\pi, \tilde{\pi}))$

Proposition H.3. *Let t be a data tree and let $n_f \in N$. Let π and $\tilde{\pi}$ be path formulas in our fragment extended with* `id`*. We have:*

$$t, \epsilon, n_f \vDash_{Ax} \mathsf{T}_{\texttt{id}}(\pi, \tilde{\pi}) \text{ iff } t, \epsilon, n_f \vDash \pi/\tilde{\pi}$$

Proof. We prove this theorem by induction on the lexicographic order over the number of occurrences of `id()` in $\tilde{\pi}$ and the size of $\tilde{\pi}$, showing only the non trivial cases:

- case $\tilde{\pi} = \pi_{id}/\pi'_{id}$:
  $t, \epsilon, n_f \vDash_{Ax} \mathsf{T}_{\texttt{id}}(\pi, \pi_{id}/\pi'_{id})$

$\iff t, \epsilon, n_f \vDash_{Ax} \mathsf{T}_{\mathtt{id}}(\mathsf{T}_{\mathtt{id}}(\pi, \pi_{id}), \pi'_{id})$
$\iff t, \epsilon, n_f \vDash_{Ax} \mathsf{T}_{\mathtt{id}}(\pi, \pi_{id})/\pi'_{id}$
$\iff \exists n \in N\ t, \epsilon, n \vDash_{Ax} \mathsf{T}_{\mathtt{id}}(\pi, \pi_{id})$ and $t, n, n_f \vDash \pi'_{id}$
$\iff \exists n \in N\ t, \epsilon, n \vDash \pi/\pi_{id}$ and $t, n, n_f \vDash \pi'_{id}$
$\iff t, \epsilon, n_f \vDash \pi/\pi_{id}/\pi'_{id}$

- case $\tilde{\pi} = \mathtt{id}(\pi_{id})$:
  $t, \epsilon, n_f \vDash_{Ax} \mathsf{T}_{\mathtt{id}}(\pi, \mathtt{id}(\pi_{id}))$
  $\iff t, \epsilon, n_f \vDash_{Ax} \mathsf{T}_{\mathtt{id}}(\varepsilon, \mathtt{id}(\mathsf{T}_{\mathtt{id}}(\pi, \pi_{id})))$
  $\iff t, \epsilon, n_f \vDash_{Ax} \mathtt{id}(\mathsf{T}_{\mathtt{id}}(\pi, \pi_{id}))$
  $\iff \exists n_{id} \in N,\ n_f \downarrow n_{id}$ and $\exists n'_f \in N,\ \delta(n_{id}) = \delta(n'_f)$ and $t, \epsilon, n'_f \vDash_{Ax} \mathsf{T}_{\mathtt{id}}(\pi, \pi_{id})$
  $\iff \exists n_{id} \in N,\ n_f \downarrow n_{id}$ and $\exists n'_f \in N,\ \delta(n_{id}) = \delta(n'_f)$ and $t, \epsilon, n'_f \vDash \pi/\pi_{id}$
  $\iff \exists n_{id} \in N,\ n_f \downarrow n_{id}$ and $\exists n'_f \in N,\ \delta(n_{id}) = \delta(n'_f)$ and $\exists n \in N$,
  $t, \epsilon, n \vDash \pi$ and $t, n, n'_f \vDash \pi_{id}$
  $\iff \exists n \in N\ t, \epsilon, n \vDash \pi$ and $t, n, n_f \vDash \mathtt{id}(\pi_{id})$
  $\iff t, \epsilon, n_f \vDash \pi/\mathtt{id}(\pi_{id})$
- case $\tilde{\pi} = \mathtt{id}(\pi')$ (with no occurrence of `id()` in π'):
  $t, \epsilon, n_f \vDash_{Ax} \mathsf{T}_{\mathtt{id}}(\pi, \mathtt{id}(\pi'))$
  $\iff t, \epsilon, n_f \vDash_{Ax}$ `$x`, where `$x` is a fresh variable axiomatised at top level with:

(1) $\varphi_1 :=$ `not(/π/π'[(. eq $x/@id) and . eq // ∗ /@id])`
(2) $\varphi_2 :=$ `not($x[not(@id eq /π/π')])`

The formula φ_1 ensures that $[\![/\pi/\mathtt{id}(\pi')]\!]^v_{\mathsf{p}} \subseteq [\![\$\mathtt{x}]\!]^v_{\mathsf{p}}$, and φ_2 ensures that $[\![\$\mathtt{x}]\!]^v_{\mathsf{p}} \subseteq [\![/\pi/\mathtt{id}(\pi')]\!]^v_{\mathsf{p}}$. Hence, the equivalence is true. □

In addition, for the EMSO^{2}XPath fragment, we can use a second-order variable to remember the starting point. In the translation of a query in an EMSO^2 formula, if the initial node is denoted by a first-order variable x, we can axiomatise a second-order variable X_{start} this way:

$$x \in X_{start} \wedge \forall y, y \in X_{start} \Rightarrow y = x$$

Then, if an `id()` is used in a non-rooted path, we can handle it by adapting the previous axiomatisation and testing for X_{start} instead of testing for the root:

(1) `not($X_start/[(not(. eq $x/@id)) and . eq // ∗ /@id])`
(2) `not($x[not(@id eq $X_start/)])`

But we must be careful not to allow such occurrences of `id()` in a node expression, as the example H.4 shows. In that sense, we must provide our grammar with a new starting symbol:

$$\pi' ::= \pi \mid \pi_{id}$$

Example H.4. The formula `/child::a/id(@x)` can be translated to `$x`, or equivalently `/child::a/$x` if we want to simulate the path it follows more accurately, where the variable `$x` axiomatised by:

(1) `not(/child::a/@x[not(. eq $x/@id) and . eq // ∗ /@id])`
(2) `not($x[not(@id eq /child::a/@x)])`

However, even though the translation will selects the same nodes as the original query, the variable `$x` will do a global jump directly to all the results, no matter what the current node is (cf. Fig. 14). Because of this, we cannot allow the use of `id` in a test: a test of the form `[id(@x)]` will be satisfied if a jump `id(@x)` can be done from the current node, but a test of the form `[$x]` will be satisfied if the variable `$x` is not empty, no matter what the current node is (cf. Fig. 15).

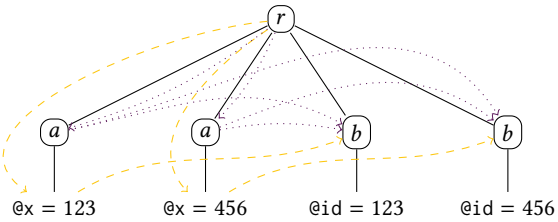

Figure 14: The paths followed by the query `/child::a/id(@x)` **in yellow, and by the query** `/child::a/$x` **in violet.**

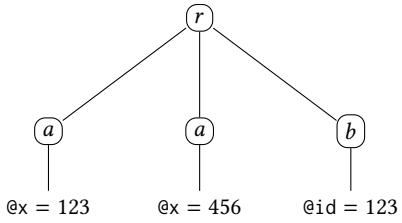

Figure 15: Counter example of a formula using `id` **in a node test: the query** `child::a[id(@x)]` **would only select the leftmost child of the root, but the query** `child::a[$x]` **with** `$x` **axiomatised as above would select all the children labelled by** *a*, **since** `$x` **would be non empty.**