Efficient Evaluation of Nondeterministic Automata Using Factorization Forests*

Mikołaj Bojańczyk and Paweł Parys

University of Warsaw

Abstract. In the first part of the paper, we propose an algorithm which inputs an NFA $\mathcal{A}$ and a word $a_1 \cdots a_n$, does a precomputation, and then answers queries of the form: "is the infix $a_i \cdots a_j$ accepted by $\mathcal{A}$?". The precomputation is in time $\text{poly}(\mathcal{A}) \cdot n$, and the queries are answered in time $\text{poly}(\mathcal{A})$. This improves on previous algorithms that worked with the exponentially less succinct DFA's or monoids.
In the second part of the paper, we propose a transducer model for data trees. We show that the transducer can be evaluated in linear time. We use this result to evaluate XPath queries in linear time.
The algorithms in both parts of the paper use factorization forests.

This paper develops the use of factorization forests [8] for efficient evaluation of automata. The paper has two parts. The first part, which builds on [4, 2], uses factorization forests to evaluate automata on arbitrary infixes of a word in constant time, after a linear time precomputation. The second part, which builds on [3, 7], uses factorization forests to efficiently evaluate queries of XPath.

Infix evaluation. The first part of the paper studies the following problem, which is parametrized by a regular language $L \subseteq A^*$. For a word $a_1 \cdots a_n \in A^*$, we want to build a data structure. Then, we want to use the data structure to quickly answer queries of the form: given two positions $i \leq j$ in $\{1, \ldots, n\}$, answer if the infix $a_i \cdots a_j$ belongs to L. We call this the *infix evaluation problem*. A solution of the problem consists of two algorithms: the *preprocessing* that inputs $a_1 \cdots a_n$ and builds the data structure, and the *query answering* which inputs $i \leq j$ and outputs the answer to $a_i \cdots a_j \in L$.

A natural solution uses a divide and conquer approach. Suppose that L is recognized by a nondeterministic automaton with states Q. The preprocessing splits the word into halves, quarters, and so on. Each such infix is decorated with the set of state pairs that describe possible runs of the automaton over the infix. The preprocessing is in time $\text{poly}(Q) \cdot n$, while the query answering is in time $\text{poly}(Q) \cdot \log(n)$.

As observed by Thomas Colcombet in [4], a beautiful result of Imre Simon, called the Factorization Forest Theorem [8], can be used to answer the queries

* We acknowledge the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599. Work supported by Polish government grant no. N N206 380037.

in time independent of the word's length. The data structure uses an algebraic approach to regular languages, where a language is recognized by a homomorphism from A^* into a finite monoid M. The preprocessing is in time linear in $|M| \cdot n$, and the query answering is in time linear in $|M|$.

What if the language L is given by an automaton and not a homomorphism? We can always compile the automaton to a monoid and use the above result. From the point of view of the length n of the word, the preprocessing is in linear time, and the query answering is in constant time. However, compiling even a deterministic automaton into a monoid can yield an exponential blowup. This gives big constants in the linear and constant times.

We can do better. If the language L is given by a deterministic automaton with states Q, a fairly straightforward structure, called the *tape construction* in [2], can be used to solve the problem with preprocessing in time $\text{poly}(Q) \cdot |n|$ and query answering in time $\text{poly}(Q)$.

In this paper, we improve the results from [4] and [2]: we give an algorithm that works with nondeterministic automata. As with the tape construction, the preprocessing is in time $\text{poly}(Q) \cdot n$ and the query answering in time $\text{poly}(Q)$. The new algorithm does not use the tape construction, which does not seem to generalize from deterministic to nondeterministic automata. Instead, it builds on factorization forests.

XPath evaluation. The second part of the paper is about XPath evaluation. The input for an XPath query is an XML document, which we model as a *data tree*. A data tree is a tree where each node carries two pieces of information: a tag name or label from a finite alphabet A, as well as a *data value* from an infinite alphabet D (such as integers, or unicode strings). An XPath query says "yes" or "no" to each node in a data tree. The XPath evaluation problem is to find the nodes to which the query says "yes".

There are algorithms which can solve this problem in time polynomial in the size of the query φ and the number of nodes n in the data tree, see [1] for a survey. However, with large XML documents (e.g. dblp.xml is currently 674 megabytes and millions of nodes), an algorithm that is quadratic in n is impractical. In previous work [3, 7], we have developed algorithms which are linear in n.

The first algorithm, from [3], runs in time $\exp(\varphi) \cdot n$. The reason for the exponential complexity in the query is that parts of the query are represented by monoids. The algorithm works for an extension of XPath, called Regular XPath, which allows Kleene star in programs. The second algorithm, from [7], runs in time $\text{poly}(\varphi) \cdot n$. It works for XPath without the Kleene star. The general idea is that monoids can be avoided without the Kleene star. Both algorithms, especially the first one, use the ideas developed in the infix evaluation problem that is studied in the first part of this paper.

In the second part of this paper, we propose a new approach to XPath evaluation. We introduce an automaton model, which acts as an intermediate step between XPath and the evaluation algorithm. The automaton model is a type of transducer, which we call a *data aggregate transducer*. Given an input data tree, a data aggregate transducer produces new labels for the nodes, and does

not change the data values. A data aggregate transducer can evaluate a query by writing "yes" or "no" in the new label, depending on whether a node is selected. (Strictly speaking, we use compositions of data aggregate transducers to evaluate XPath queries.)

The advantage of this new approach is that the syntax of XPath is abstracted into a simple automaton model. This makes the evaluation algorithm easier to understand, and its structure more apparent. We also believe that the automata models we introduce, in the general form for data trees (data aggregate transducers), and in the restricted form for trees without data (which we call aggregate transducers), are of independent interest for evaluation algorithms.

The evaluation algorithm uses the algebraic techniques developed in the first part of the paper. Thanks to the efficient algorithms for nondeterministic automata (the path expressions in an XPath query are naturally modeled by nondeterministic automata), we get a new result: For data trees of bounded depth, a query φ of XPath with Kleene star can be evaluated in time $\text{poly}(\varphi) \cdot n$. In other words, for documents of bounded depth (a common situation), we can combine the efficient evaluation of [7] with the more powerful query language of [3].

1 Evaluating infix queries for nondeterministic automata

This section contains the first part of the paper, which talks about the infix evaluation problem. Instead of specifying an infix by its first x and last position y, we use the set of all of its positions $X = \{x, x+1, \ldots, y\}$. This way we can use set operations on infixes. If X is a set of positions in a word w, we write $w[X]$ for the subsequence of w consisting of positions from X, e.g. $a_1a_2a_3[\{1,3\}] = a_1a_3$. We use the name *factor* of w for a connected set of positions, and the name *infix* for the word $w[X]$ when X is a factor. (Of course, the algorithms represent factors by just keeping the first and last position.) We write x, y, z for positions, X, Y, Z for sets of positions, and F, G, H for factors (which are also sets of positions).

Factorization forests. A *factorization forest for a word w* is a family of factors that contains $\{x\}$ for every position x in w, and where every two factors are either disjoint, or one is contained in the other. There is a natural forest structure on the factors, so we can talk about descendants, parents, children and siblings, etc. The *level* of a factor is the number of its ancestors (including itself).

Suppose that $F_1, \ldots, F_n$ are consecutive factors (i.e. the first position of F_{i+1} is the next position after the last position of F_i). A *collation* of these factors is any union of these factors that is also a factor, i.e. any $F_i \cup \cdots \cup F_j$ for $i \leq j$. Consider a morphism $\alpha : A^* \to M$ into a finite monoid, which we use to map factors into M. We say that $F_1, \ldots, F_n$ are α-homogeneous if all of their collations have the same value under α. A factorization forest is called α-homogeneous if any choice of at least three consecutive siblings is α-homogeneous.

Suppose that w is a word with a factorization forest $\mathcal{F}$ that is α-homogeneous for a morphism $\alpha : A^* \to M$. Colcombet observed in [4] that for any factor I of w, not necessarily from $\mathcal{F}$, its image under α can be calculated in time linear in

the height of $\mathcal{F}$. Our work builds on this observation. As a first step, we show that the time can be even logarithmic in the height of $\mathcal{F}$.

Logarithmic querying. For the algorithms, we represent a factorization forest $\mathcal{F}$ as follows. Each factor $F \in \mathcal{F}$ is represented by a record with its first and last position, and its image under α. Each position x contains a pointer to the record of the factor $\{x\}$.

Each factor record stores a pointer to its parent factor record, but also to some other ancestors, as described below. Let n be a number from 0 to the logarithm of the height of the factorization forest. Consider a factor $F \in \mathcal{F}$. We create a pointer from the record of F to the record of the 2^n-parent of F, call it G. (The 2^n-parent is the ancestor 2^n levels above.) This pointer is called the *accelerating pointer of length* 2^n. It is decorated by two elements of M, which are the images under α of the two factors below.

- *left*(F, G): positions from G that are strictly before all positions from F.
- *right*(F, G): positions from G that are strictly after all positions from F.

The number of accelerating pointers, and the time required to compute them, is $|\mathcal{F}| \cdot \log h$, where h is the height of $\mathcal{F}$. From now on, we assume in our algorithms that factorization forests are equipped with accelerating pointers.

The benefit of accelerating pointers is that one can go from a factor to any of its ancestors by following a number of accelerating pointers that is logarithmic in the height of the forest. This observation, together with the original idea of using factorization forests homogeneous with a morphism to calculate images of infixes, gives the following result.

Lemma 1.1. *Let $\alpha : A^* \to M$ be a morphism, and let $\mathcal{F}$ be an α-homogeneous factorization forest for a word $w \in A^*$. Using the accelerating pointers, the image under α of any factor can be calculated in time logarithmic in the height of $\mathcal{F}$.*

Combining the above lemma with a divide and conquer approach, we get a solution for the infix evaluation problem that has querying in time $\log(\log(|w|))$. This is because a divide and conquer approach yields a factorization forest with binary branching, and such a forest is α-homogeneous for any α.

1.1 Monoid of binary relations.

Let Q be any finite set. We write M_Q for the monoid of binary relations Q, where the monoid operation is relation composition. In this section, we study factorization forests that are α-homogeneous, for some $\alpha : A^* \to M_Q$. The size of the monoid M_Q is exponential in the size of Q. The main result of the first part of this paper is that we can build a factorization forest without worrying about this exponential blowup.

Theorem 1.2. *Consider a morphism $\alpha : A^* \to M_Q$. For any word $w \in A^*$ we can find, in time* $\mathrm{poly}(Q) \cdot |w|$*, an α-homogeneous factorization forest for w of height at most*[1] $\mathrm{poly}(M_Q)$.

[1] The height can be even linear in $|M_Q|$, but it requires more care in the proof.

We describe the proof of this theorem in Section 1.2.

Corollary 1.3. *Let $L \subseteq A^*$ be a language recognized by a nondeterministic automaton with states Q. The infix evaluation problem for a word $w \in A^*$ can be solved with precomputation* $\mathrm{poly}(Q)\cdot|w|$ *and query answering in time* $\mathrm{poly}(Q)$.

Proof. The nondeterministic automaton can be identified with a morphism $\alpha : A^* \to M_Q$, which maps a word w to the set of pairs (p, q) such that the automaton has a run from p to q over the word. Using the above theorem, we can compute a factorization forest in time $\mathrm{poly}(Q)\cdot|w|$. The height of the forest may be exponential in Q, since the height is bounded by M_Q. However, we can use the logarithm from Lemma 1.1 to query answer infix queries in time $\mathrm{poly}(Q)$. □

1.2 Proof of Theorem 1.2

For the rest of this section, we fix the monoid M_Q and the morphism α. We write r, s, t for the binary relations which are elements of M_Q, and $r \circ s$ for composition of binary relations, which is the monoid operation.

Green's relations. Let r, s, t, t_1, t_2 below be elements of M_Q.

- r is called a prefix of s, written $r \geq_{\mathcal{R}} s$, if there is some t with $r \circ t = s$.
- r is called a suffix of s, written $r \geq_{\mathcal{L}} s$, if there is some t with $t \circ r = s$.
- r is called an infix of s, written as $r \geq_{\mathcal{J}} s$, if there are t_1, t_2 with $t_1 \circ r \circ t_2 = s$.
- If r is both a prefix and a suffix of s, we write $r \geq_{\mathcal{H}} s$.

These relations are called Green's relations. It is easy to see that each of Green's relations is a pre-order: it is both transitive and reflexive. The relations are not necessarily antisymmetric and therefore it makes sense to consider their connected components. For instance, we say that r and s are $\mathcal{R}$-equivalent, written $r \sim_{\mathcal{R}} s$, if both $r \geq_{\mathcal{R}} s$ and $s \geq_{\mathcal{R}} r$. An equivalence class is called an $\mathcal{R}$-class. Likewise for $\mathcal{L}$, $\mathcal{J}$ and $\mathcal{H}$.

In the algorithm, we will need to perform operations on M_Q in time $\mathrm{poly}(Q)$. One such operation is calculating composition $r \circ s$, this is easy to do. A problem that we will have to work around is that we do not know how to test $\mathcal{J}$-equivalence in time $\mathrm{poly}(Q)$. However, we can do this in some special cases, as stated in the following lemma.

Lemma 1.4. *Given $r, s \in M_Q$, we can calculate the following in time* $\mathrm{poly}(Q)$*:*

$$r \circ s, \qquad r \circ s \overset{?}{\sim}_{\mathcal{J}} r, \qquad r \circ s \overset{?}{\sim}_{\mathcal{J}} s.$$

Proof strategy. We present the proof strategy for Theorem 1.2.

The definition of α-homogeneous factors or factorization forests also makes sense in a more general setting, where α is any function that maps factors of $\mathcal{F}$ to some set, not necessarily a morphism. We use this generalization to define notions of $\mathcal{J}$-homogeneity and $\mathcal{H}$-homogeneity. Let $F_1, \ldots, F_n$ be consecutive factors. We say the factors are $\mathcal{J}$-homogeneous if they are f-homogeneous under

the function f that maps a factor to the $\mathcal{J}$-class of its image. (In general, f is not a morphism.) Likewise we define a $\mathcal{J}$-homogeneous factorization forests, and the same for $\mathcal{H}$.

Our proof strategy is to first compute a $\mathcal{J}$-homogeneous factorization forest, then upgrade it to an $\mathcal{H}$-homogeneous one, and then upgrade that one to an α-homogeneous one. The main difficulty is in the first step – computing a $\mathcal{J}$-homogeneous forest; we do this below in Lemma 1.5. The other steps are done using basically the same techniques as in the proof of the factorization forest theorem from [6], or to the proofs of [8, 4].

Lemma 1.5. *Let $w \in A^*$. One can compute a $\mathcal{J}$-homogeneous factorization forest $\mathcal{F}$ in time* $\mathrm{poly}(Q) \cdot |w|$. *The forest has height linear in M_Q.*

Proof. The algorithm processes word positions from left to right. We begin by describing the invariant.

The invariant. After processing position x, the algorithm will have computed a factorization forest $\mathcal{F}_x$ for the prefix $1, \ldots, x$. For each factor we remember one additional bit: if the factor is *open* or *closed*. All open factors have to contain the last processed position x. Open factors might grow when processing new positions. Once a factor becomes closed, it does not change. All singleton factors are closed. Suppose $F_1, \ldots, F_n \in \mathcal{F}_x$ is a maximal set of siblings (written from left to right). The invariant is that they satisfy the following property $\star$:

$\star$ The factors $F_1, \ldots F_{n-1}$, and the factor $F_1 \cup \cdots \cup F_{n-1}$ are all $\mathcal{J}$-equivalent.

Additionally, when they are children of an open factor $F \in \mathcal{F}_x$, the following property $\star\star$ is satisfied:

$\star\star$ F and F_1 are $\mathcal{J}$-equivalent.

The invariant is satisfied by the initial configuration $\mathcal{F}_1 = \{\{x\}\}$.

Once we have processed the whole word, it is not difficult to get a $\mathcal{J}$-homogeneous factorization forest from the one produced by the algorithm. For each maximal set of siblings $F_1, \ldots, F_n \in \mathcal{F}_x$, it is enough to add a factor $F_1 \cup \cdots \cup F_{n-1}$.

Updating the forest. Suppose we have computed $\mathcal{F}_{x-1}$, and we want to compute $\mathcal{F}_x$. Consider the factors open in $\mathcal{F}_{x-1}$:

$$x - 1 \in F_1 \subsetneq F_2 \subsetneq \cdots \subsetneq F_n.$$

There are also closed factors containing $x - 1$, at least one: $\{x - 1\}$. Let C be the biggest of them. We obtain $\mathcal{F}_x$ from $\mathcal{F}_{x-1}$ as follows.

- Add $\{x\}$.
- If C and F_1 are not $\mathcal{J}$-equivalent, or $n = 0$, add open factor $G_0 = C \cup \{x\}$.
- Replace the factors F_i by $G_i = F_i \cup \{x\}$, for $i \in \{1, \ldots, n\}$.
- When $G_i \backslash \{x\}$ and G_i are not $\mathcal{J}$-equivalent close G_i, for $i = 0$ (if G_0 was added) and for $i \in \{1, \ldots, n\}$.

The test on $\mathcal{J}$-equivalence in the second and the last step is done using Lemma 1.4, since we are testing $\mathcal{J}$-equivalence of a factor and its suffix or prefix. Below we argue that the invariant is preserved. Then, we show why the algorithm runs in the required time, and why the factorization forest has height linear in M_Q.

Correctness. Extending a factor does not impact on property $\star$, as it does not talk about a last sibling. Property $\star$ has to be checked only for the siblings of the newly added factor $\{x\}$. If G_0 is created, $\{x\}$ has only one sibling, so $\star$ is satisfied. Otherwise C is no longer the last sibling. This happens only when C is $\mathcal{J}$-equivalent to its parent F_1. As F_1 is open, it is $\mathcal{J}$-equivalent to its first child (from $\star\star$), hence to all its children (from $\star$), which gives $\star$ in the new forest.

Now check the property $\star\star$ for open factors. Factor G_0 stays open only when G_0 and $G_0\backslash\{x\} = C$ are $\mathcal{J}$-equivalent, which is exactly $\star\star$. Any other G_i stays open when it is $\mathcal{J}$-equivalent to F_i, which (from $\star\star$) is equivalent to its first child (which is also the first child of G_i).

Running time. A potential problem is the last step. Potentially we have to do n tests for $\mathcal{J}$-equivalence. However notice that when $G_i\backslash\{x\}$ and G_i are $\mathcal{J}$-equivalent for some i, then they are $\mathcal{R}$-equivalent (Lemma A.2), hence also $G_j\backslash\{x\}$ and G_j are $\mathcal{R}$-equivalent ($\mathcal{J}$-equivalent) for any $j > i$. Thus we may stop testing greater i when we detect an equivalence. The number of tests for $\mathcal{J}$-equivalence is bounded by the number of factors becoming closed (plus one). Since the total number of factors in a factorization forest is at most twice the length of the word, we have a limit on the total number of operations in the last step of the algorithm.

Two implementation problems remain. First, where do we get the images of the factors $F_1, \ldots, F_n$ that are used in the tests for $\mathcal{J}$-equivalence? The answer is that our algorithm maintains for each open factor F_j, the image of its closed part $F_j - F_{j-1}$. Second, what is the cost of adding x to the factors F_i? The answer is that this can achieved for free, if we do not store the ends of open factors, but we only keep in mind that they all end in the currently processed position x.

Height of the forest. Why is the height of the factorization forest linear in M_Q? It would be useful to look at the $\mathcal{J}$-class of the first child of each non-singleton factor. The following invariant is preserved by the algorithm: whenever a factor F in the factorization forest is the parent of a non-singleton factor G, then the first child of F has a smaller $\mathcal{J}$-class than the first child of G. It guarantees that the level of a factor is bounded by the position of its first child in the $\leq_{\mathcal{J}}$ order.

Why is the invariant satisfied? First observe an auxiliary property of the forest: every closed factor in the factorization forest (except singletons) has a different (smaller) $\mathcal{J}$-class than its first child. Indeed, when a factor G_i becomes closed, it has a different $\mathcal{J}$-class than $G_i\backslash\{x\}$, which contains the first child of G_i.

To prove the invariant notice that during execution of the algorithm, the first child of a factor is never modified. Hence it is enough to analyze each moment when a new pair of a parent and its child is created. It happens only in the second

step, when G_0 is created (creating $\{x\}$ does not matter, as the invariant does not talk about singleton factors). First compare G_0 with its only non-singleton child C. As C is closed, from the above we know that its first child has greater $\mathcal{J}$-class than C itself, which is the first child of G_0. Now compare G_0 with its parent G_1. The factor G_0 is created only when C (the first child of G_0 has greater $\mathcal{J}$-class than F_1. Because F_1 is open, from $\star\star$ we get that it is $\mathcal{J}$-equivalent with its first child (which is also the first child of G_1). $\square$

2 Aggregate Transducers

In this part of the paper, we introduce a new transducer model for data trees. This transducer is designed so that: a) it can compute interesting properties, such as XPath queries; b) it can be evaluated in linear time.

2.1 Trees without data

Basic definitions. We work on finite, labeled, sibling-ordered trees. The trees are unranked, which means that there is no restriction on the number of children of a node. We use the usual notions of node, root, child, parent, descendant, ancestor etc. We write $t(x)$ for the label assigned by the tree t to the node x. We write trees(A) for the set of trees labeled by alphabet A. To recognize tree languages, we use nondeterministic automata on unranked trees. The exact choice of automaton model is not important for the discussion here; we choose nondeterministic finite hedge automata as defined in Section 8.2.2 of [5].

Transducers. Let A be an input alphabet and B an output alphabet. If s and t are trees with the same nodes, over alphabets A and B, then we write $s \otimes t$ for the tree over alphabet $A \times B$ that has the same nodes as s, t and maps each node x to the pair $(s(x), t(x))$. Consider a tree language over the product alphabet $A \times B$. This language can interpreted as a binary relation

$$f \subseteq \text{trees}(A) \times \text{trees}(B)$$

which contains a pair of trees (s, t) if the tree $s \otimes t$ belongs to the language. Note that the relation only contains tree pairs that have the same nodes. This type of relation is called a *transducer*. We use functional notation for transducers, writing $f(s)$ for the set of trees t with $(s, t) \in f$. We say a tree automaton *represents* f if it represents the underlying tree language over alphabet $A \times B$.

Aggregation. Consider an alphabet B equipped with a linear order. Suppose that s and t are trees over B that have the same nodes. We use the linear order to define a new tree, written $s \sqcup t$, which we call the *aggregation of s and t*. The tree $s \sqcup t$ has the same nodes as s and t, it assigns to a node x the bigger of the labels $s(x), t(x)$. The aggregation operation is commutative and associative, and therefore it makes sense to talk about the aggregation $\sqcup S$ of a set S of trees which share the same nodes.

Aggregate transducers. Suppose that f is a transducer with input alphabet A and output alphabet B. Suppose also that B is equipped with a linear order. Consider the function, call it $\sqcup f$, defined as

$$s \in \text{trees}(A) \qquad \mapsto \qquad \sqcup f(s) = \bigsqcup_{t \in f(s)} t \quad \in \text{trees}(B).$$

The notation $\sqcup f(s)$ is unambiguous, since $(\sqcup f)(s)$ and $\sqcup(f(s))$ mean the same thing. If $f(s)$ is empty, we define $\sqcup f(s)$ to be the tree with nodes from s labeled by the minimal element of B. Note that while f maps each tree to a set of trees, the function $\sqcup f$ maps each tree to a single tree. Any function of the form $\sqcup f$ is called an *aggregate transducer*. We believe that aggregate transducers are of independent interest.

2.2 Trees with data

Data trees. Fix an infinite domain D of data values, e.g. $D = \mathbb{N}$. A *data tree* over a finite alphabet A is a tree over alphabet $A \times D$. The set of all data trees over an alphabet A is denoted $\text{dtrees}(A)$. We write such trees as $t \otimes \mu$, where t is a tree over A and μ a tree over D. The *label* of a node is its label in t, its *data value* is its label in μ. We use the name *class* for a set of nodes with the same data value. We assume that the data values are not greater than the number of nodes; thanks to this the classes can be found in time linear in the tree size. Data trees will be our document model for XPath queries[2].

Data aggregate transducer. We overload the $\otimes$ notation for sets as follows: if t is a tree over A and X is a set of nodes, we write $t \otimes X$ for the tree over $A \times \{0,1\}$, where the label of each node in t is enriched by a bit indicating membership in X. Consider a transducer f with input alphabet $A \times \{0,1\}$ and output alphabet B. Suppose also that B is equipped with a linear order so that trees over B can be aggregated. Consider the function, call it $\widehat{f}$, defined as

$$s \otimes \mu \in \text{dtrees}(A) \qquad \mapsto \qquad \widehat{f}(s \otimes \mu) = \bigsqcup_{X \text{ a class of } \mu} \sqcup f(s \otimes X) \quad \in \text{trees}(B).$$

This is a function that maps a data tree over A to a tree without data over B. We use the name *data aggregate transducer* for any such function. An automaton *representing* $\widehat{f}$ is any automaton representing f. Note that since $\sqcup f(s)$ is itself an aggregation, the output $\widehat{f}(s \otimes \mu)$ is

$$\bigsqcup_{X \text{ a class of } \mu} \quad \bigsqcup_{t \in f(s \otimes X)} t.$$

The main motivation behind data aggregate transducers is that they can be used to evaluate XPath queries. We show this in Section 3.

[2] In XML instead of small numbers we have arbitrary strings; however they can be sorted lexicographically and replaced by numbers in time linear in their total size. This is true even when the string value in an element node is not given explicitly, but is a concatenation of string values in its children, see [7].

Evaluation. The principal result on data aggregate transducers is that they can be evaluated in linear time. We have two variants of this result. The first variant works for the general case of data trees, but the constant in the linear time is exponential in the state space of the data aggregate transducer. The second variant has a polynomial constant, but it works only for data words, which are the special case of data trees where each node has at most one child.

Theorem 2.1. *Let $\widehat{f}$ be a data aggregate transducer represented by a nondeterministic tree automaton with states Q. The output of $\widehat{f}$ on a data tree $t \otimes \mu$ can be evaluated in time*

- $exp(Q) \cdot |t|$ *in the general tree case;*
- $\text{poly}(Q) \cdot |t|$ *if $t \otimes \mu$ is a data word, i.e. each node has one child.*

We do not know if the variant for the general tree case can be improved to run in time $\text{poly}(Q) \cdot |t|$.

2.3 Evaluating a data aggregate transducer on data words

In this section, we prove the word case of Theorem 2.1, which says that data aggregate transducers can be evaluated in linear time. The tree case is done in the appendix. Instead of writing a data word as a tree where each node has one child, we use the standard notation for words as sequences of letters $a_1 \cdots a_n$.

We fix a data aggregate transducer $\widehat{f}$, and a nondeterministic automaton $\mathcal{A}$ of states Q that recognizes the underlying transducer f. The input alphabet of $\mathcal{A}$ is $A \times \{0,1\} \times B$. Fix also an input data word $w \otimes \mu$ of length n. We want to compute the output $\widehat{f}(w \otimes \mu)$. When talking about factors, we mean factors in a word of length n.

Snippets. We write $\bot$ for the minimal letter in the output alphabet B of f. For a word v over alphabet B and a set of positions Y, we we write $s \sqcap Y$ for the word obtained from v by replacing the labels of positions outside Y by $\bot$. A *partial output* is any value $(\sqcup f(w \otimes X)) \sqcap Y$ for some class X. A *snippet* is a partial output in which Y is a factor that is either disjoint with X, or included in X.

The *type* of a word $v \in (A \times \{0,1\})^*$ is the set of state pairs (p,q) such that $\mathcal{A}$ has a run from p to q over $v \otimes u$ for some $u \in B^*$. The *internal type* of a factor Y in a word $w \otimes X$ is the type of the corresponding infix. Let Y_1 (respectively, Y_2) consist of all positions before (after) a factor Y. The *external type* of the factor Y in a word $w \otimes X$ is the set of state pairs (p,q) such that (q_I, p) is in the internal type of Y_1 and (q, q_F) is in the internal type of Y_2 for an initial state q_I and an accepting state q_F. The external type of Y can be deduced from the internal types of Y_1 and Y_2.

We will use a concise representation for a snippet $(\sqcup f(w \otimes X)) \sqcap Y$. The *snippet representation* consists of: the factor Y, its external type in $w \otimes X$ and a membership bit saying whether Y is contained in X or disjoint with X. Note that this information determines the value of $(\sqcup f(w \otimes X)) \sqcap Y$, as a word in B^*, even without knowing X.

We now have the necessary concepts to present our proof strategy for Theorem 2.1. Our goal is to produce the output $\widehat{f}(w \otimes \mu)$. Our algorithm will represent

this output as the aggregation of a set of snippets. Whenever a subroutine of the algorithm inputs or outputs a set of snippets, we assume that the snippets are given by their representations.

The algorithm works in three stages.

Stage 1. We compute two factorization forests. Consider two words

$$w_0 = w \otimes \emptyset, \qquad w_1 = w \otimes \{1, \ldots, n\} \quad \in (A \times \{0,1\})^*.$$

We will use factorization forests for these words, for the morphism

$$\alpha : (A \times \{0,1\})^* \to M_Q$$

which maps a word v to its type. Apply Theorem 1.2 to the words w_0, w_1 and the morphism α, yielding factorization forests $\mathcal{F}_0$, $\mathcal{F}_1$. These factorization forests will be used by the next two stages of the algorithm.

Stage 2. We show that for each class X, the output $\sqcup f(w \otimes X)$ can be represented by a small number of snippets. This is stated by the following lemma.

Lemma 2.2. *Let X be a set of positions. We can calculate a set S_X of snippets such that $\sqcup f(w \otimes X) = \sqcup S_X$. The cardinality of S_X and time to calculate it are* $\mathrm{poly}(Q) \cdot |X|$.

Proof. Let $Y_1, \ldots, Y_m$ be a partition of $\{1, \ldots, n\}$ into factors such that the odd numbered factors are the maximal factors contained in X, and the even numbered ones are disjoint with X. (The first and last factors might be empty.) The set S_X consisting of the snippets $(\sqcup f(w \otimes X)) \sqcap Y_i$ satisfies the thesis. The factors Y_i can be calculated in time linear in the number of positions in X.

We need to find the representation of the snippets, namely the external types of Y_i in $w \otimes X$. The calculation will take time linear in m, and therefore at most linear in the size of X. Let first compute their internal types. For even i, the internal type of Y_i is $\alpha(w \otimes X[Y_i]) = \alpha(w_0[Y_i])$, hence we can compute it using the factorization forest $\mathcal{F}_0$. Using $\mathcal{F}_1$, we can do the same for odd i. Using compositionality of types, we calculate internal types of $Y_1 \cup \cdots \cup Y_i$ for each i, going from left to right, and of $Y_i \cup \cdots \cup Y_m$, going from right to left. The external type of Y_i is found basing on the internal types of $Y_1 \cup \cdots \cup Y_{i-1}$ and $Y_{i+1} \cup \cdots \cup Y_m$. $\square$

Stage 3. In the third and final stage, we show that snippets can be efficiently aggregated. We apply Lemma 2.2 to each class X, yielding a set of snippets S_X. All we have to do is to aggregate them, i.e. aggregate all snippets that belong to some S_X for some class X. This can be done in linear time thanks to the following proposition.

Proposition 2.3. *Let $s_1, \ldots, s_m$ be snippets. Their aggregation $s_1 \sqcup \cdots \sqcup s_m$ can be calculated in time* $\mathrm{poly}(Q) \cdot (m + |w|)$.

3 An application to evaluating queries of Regular XPath

Regular XPath is a logic for data trees, which extends XPath 1.0 by adding a Kleene star. There are two kinds of formulas in Regular XPath: unary queries and binary queries. A unary query maps a data tree to a set of nodes, and a binary query maps a data tree to a set of node pairs. The formulas of Regular XPath and their semantics, as we use them here, are defined in [3].

Theorem 3.1. *Let φ be a unary query of Regular XPath. The set of nodes selected by φ in a data tree with n nodes can be computed in time:*

- $exp(\varphi) \cdot n$*; or*
- $\text{poly}(\varphi) \cdot n$*; if the input is a word.*

The proof, given in the appendix, is straightforward: describe a query using data aggregate transducers, and apply Theorem 2.1. We would like to point out that a query is not described by a single data aggregate transducer, but a sequential composition, where each new transducer reads the output of the previous one.

Corollary 3.2. *Let φ be a unary query of Regular XPath. The nodes selected by φ in a data tree of height k with n nodes can be computed in time* $\text{poly}(k, \varphi) \cdot n$.

Proof. A data tree $t \otimes \mu$ of height k over an alphabet A can be encoded, by writing the nodes in document order and decorating them with their depths, as a data word $enc_k(t \otimes \mu)$ over alphabet $A \times \{1, \ldots, k\}$. This encoding can be decoded by Regular XPath in the following sense: for each unary query φ we can compute in time $\text{poly}(k, \varphi)$ a query $enc_k(\varphi)$ such that the set of nodes selected by φ in $t \otimes \mu$ can be recovered in linear time from the set of nodes selected by $enc_k(\varphi)$ in the data word $enc_k(t \otimes \mu)$. The idea is to replace the axes: e.g. next sibling is replaced by a disjunction, over all $i \in \{1, \ldots, k\}$, of the binary query which connects a position x of depth i with the first position $y > x$ such that y has depth i and all positions between x and y have depth at least $i + 1$. The Kleene star is needed to talk about the positions between x and y. □

References

1. M. Benedikt and C. Koch. XPath leashed. *ACM Comput. Surv.*, 41(1), 2008.
2. M. Bojanczyk. Factorization forests. In *Developments in Language Theory*, pages 1–17, 2009.
3. M. Bojanczyk and P. Parys. XPath evaluation in linear time. In *PODS*, pages 241–250, 2008.
4. T. Colcombet. On factorisation forests. *CoRR*, abs/cs/0701113, 2007.
5. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.
6. M. Kufleitner. The height of factorization forests. In *MFCS*, pages 443–454, 2008.
7. P. Parys. XPath evaluation in linear time with polynomial combined complexity. In *PODS*, pages 55–64, 2009.
8. I. Simon. Factorization forests of finite height. *Theor. Comput. Sci.*, 72(1):65–94, 1990.

A Evaluating infix queries for nondeterministic automata

A.1 Accelerating pointers

We first show some additional lemma about factorization forests, which will be used in Section A.2, as well as in Section D.

When we say that a factor F is *decomposed* into some factors, we mean that the factors are disjoint and their union is F.

Let $\mathcal{F}$ be a factorization forest. Consider the family $\mathcal{P}(\mathcal{F})$ containing

- all factors from $\mathcal{F}$, and
- for each accelerating pointer from F to G, the factors $\mathit{left}(F, G)$ and $\mathit{right}(F, G)$, if are nonempty.

Note that the number of factors in $\mathcal{P}(\mathcal{F})$ is linear in $|w|$ and logarithmic in the height of $\mathcal{F}$, and that this family satisfies the following property $\sharp$: for each factor X in $\mathcal{P}(\mathcal{F})$,

- $X = \{x\}$ for some position x, or
- X is decomposed into factors $X_1, X_2 \in \mathcal{P}(\mathcal{F})$. Moreover, $\mathcal{P}(\mathcal{F})$ is organized so that X_1 and X_2 can be found in constant time.

Indeed, a non-singleton factor $F \in \mathcal{F}$ can be decomposed into its first child F_1 and the factor $\mathit{right}(F_1, F)$ (we have an accelerating pointer of length 1 from F_1 to F). A factor $\mathit{left}(F, G)$ for an accelerating pointer of length $2^k > 1$ from F to G can be decomposed into $\mathit{left}(F, F')$ and $\mathit{left}(F', G)$, where from F to F' and from F' to G we have pointers of length 2^{k-1}. When the length is 1, we decompose $\mathit{left}(F, G)$ into the previous sibling F' of F and $\mathit{left}(F', G)$. Similarly for $\mathit{right}(F, G)$.

The accelerating pointers are invented in such a way that the following lemma holds.

Lemma A.1. *Any factor X can be decomposed into several factors $X_1, \ldots, X_m$ from $\mathcal{P}(\mathcal{F})$ and at most one factor X' being a collation $F_1 \cup \cdots \cup F_k$, where $F_1, \ldots, F_k$ are siblings in $\mathcal{F}$. Both the number of factors and the time to compute it are logarithmic in the height of the forest.*

Proof. Let F be the smallest factor in $\mathcal{F}$ that contains X, and let $F_0, \ldots, F_{n+1}$ ($n \geq 0$) be the children of F that intersect X, written from left to right. The records of F, F_0 and F_{n+1} can be found by following the pointers in the forest, starting with the leftmost and rightmost positions in X. If we use the accelerating pointers, we only need time logarithmic in the height of the forest.

The factor X is decomposed as

$$X = \mathit{right}(X, F_0) \cup F_1 \cup \ldots \cup F_n \cup \mathit{left}(X, F_{n+1}).$$

Moreover, $\mathit{right}(X, F_0)$ can be decomposed into several factors of the form $\mathit{right}(F, G)$ with an accelerating pointer from F to G. Namely, we take such factor for each accelerating pointer used to find F_0. Their number is logarithmic in $\mathcal{F}$. Similarly for $\mathit{left}(X, F_n)$. □

A.2 Proof of Lemma 1.1

Let X be a factor whose image under α we want to calculate. We decompose X using Lemma A.1 into factors from $\mathcal{P}(\mathcal{F})$ and a collation. It is enough to find the image under α for each of them, and then compose. For the factors from $\mathcal{P}(\mathcal{F})$ the image is remembered in the data structure. Let now find the image of the factor $F_1 \cup \cdots \cup F_n$. If $n = 1$ or 2, we can simply read the image. Otherwise, $n > 2$ and therefore $F_1, \ldots, F_n$ are α-homogeneous, and the image of the collation $F_1 \cup \cdots \cup F_n$ is the same as the image of, say F_1, which is stored in its record.

A.3 Proof of Lemma 1.4

It is easy to compute the composition $r \circ s$ of two relations $r, s \in M_Q$ in time poly(Q). The rest of Section A.3 is devoted to showing how to test

$$r \circ s \overset{?}{\sim}_{\mathcal{J}} r, \qquad r \circ s \overset{?}{\sim}_{\mathcal{J}} s.$$

The following lemma shows that all we need to do is test $\mathcal{R}$-equivalence and $\mathcal{L}$-equivalence.

Lemma A.2. *For $r, s \in M_Q$ the two equivalences below hold:*

$$r \circ s \sim_{\mathcal{J}} r \quad \Leftrightarrow \quad r \circ s \sim_{\mathcal{R}} r \qquad \text{and} \qquad r \circ s \sim_{\mathcal{J}} s \quad \Leftrightarrow \quad r \circ s \sim_{\mathcal{L}} s.$$

Proof. This is a classic fact from the theory of Green's relations, but we prove it here for the sake of completeness. We only prove the part concerning $\mathcal{R}$-classes, namely

$$r \circ s \sim_{\mathcal{J}} r \quad \Leftrightarrow \quad r \circ s \sim_{\mathcal{R}} r.$$

The proof for $\mathcal{L}$-classes is the same. Only the implication from left to right is nontrivial. By the assumption $r \circ s \sim_{\mathcal{J}} r$ there must be some $t, u \in M$ such that

$$r = t \circ r \circ s \circ u$$

By substituting n times the right side instead of r, we get

$$r = t^n \circ r \circ (s \circ u)^n$$

If we choose n so that $(s \cdot u)^n$ is idempotent (this is always possible in a finite monoid), we get

$$r = t^n \circ r \circ (s \circ u)^n = t^n \circ r \circ (s \circ u)^n \circ (s \circ u)^n = r \circ (s \circ u)^n$$

which shows that $r \circ s$ is a prefix of r, and hence $r \circ s \sim_{\mathcal{R}} r$. □

To complete the proof of Lemma 1.4, it remains to efficiently test $\mathcal{R}$-equivalence and $\mathcal{L}$-equivalence. We show how to test $\mathcal{R}$-equivalence, $\mathcal{L}$-equivalence is done the same way. The key observation is stated below.

Lemma A.3. *Let Q be a finite set and r_1, r_2 two elements of M_Q. We define*

$$Q_1(q_2) = \{q_1 \colon r_1^{-1}(q_1) \subseteq r_2^{-1}(q_2)\}.$$

It holds $r_1 \geq_{\mathcal{R}} r_2$ iff $r_1^{-1}(Q_1(q_2)) = r_2^{-1}(q_2)$ for each state q_2.

Proof. First assume that $r_1 \geq_{\mathcal{R}} r_2$, i.e. $r_1 \cdot r = r_2$ for some $r \in M_Q$. Fix some state q_2. Of course $r_1^{-1}(Q_1(q_2)) \subseteq r_2^{-1}(q_2)$ because $q_1 \in Q_1(q_2)$ only if $r_1^{-1}(q_1) \subseteq r_2^{-1}(q_2)$. Now take $q \in r_2^{-1}(q_2)$, which means that $q \in r_1^{-1}(q_1)$ for some $q_1 \in r^{-1}(q_2)$. But then $r_1^{-1}(q_1) \subseteq r_2^{-1}(q_2)$, so $q_1 \in Q_1(q_2)$ and $q \in r_1^{-1}(Q_1(q_2))$.

For the other direction assume that $r_1^{-1}(Q_1(q_2)) = r_2^{-1}(q_2)$ for each state q_2; we need to find r such that $r_1 \cdot r = r_2$. Let r contain pairs (q_1, q_2) such that $q_1 \in Q_1(q_2)$. Then for any state q_2 it holds

$$(r_1 \cdot r)^{-1}(q_2) = r_1^{-1}(r^{-1}(q_2)) = r_1^{-1}(Q_1(q_2)) = r_2^{-1}(q_2),$$

which shows that $r_1 \cdot r = r_2$. □

The above lemma gives a criterion for deciding whether $r_1 \geq_{\mathcal{R}} r_2$, which may be checked in time $O(|Q|^3)$.

A.4 From a $\mathcal{J}$-homogeneous forest to an α-homogeneous one

In this part of the appendix we finish the proof of Theorem 1.2. Recall that thanks to Lemma 1.5, we have a $\mathcal{J}$-homogeneous factorization forest for our input word, which has height linear in $|M_Q|$.

Our proof is in two steps. First, we upgrade the $\mathcal{J}$-homogeneous factorization forest to an $\mathcal{H}$-homogeneous one. Then, we upgrade the $\mathcal{H}$-homogeneous factorization forest to an α-homogeneous one required by Theorem 1.2.

First we state a technical lemma, which will be used in both steps.

Left-zero monoids. As a tool in the proof, we show that factorization forests can be efficiently computed for monoids where the value of a factor is the value of its leftmost position.

For a set M, define a monoid *left*(M). Its elements are elements of M, plus an identity. The multiplication is defined by $mn = m$ for $m \in M$. This type of monoid is called a left-zero monoid.

Lemma A.4. *Let M be a set whose elements can be represented using k bits. Let $\beta : M^* \to$ left(M) be the morphism that extends the identity function. For each word $w \in M^*$, in time $k \cdot |w|$ we can compute a β-homogeneous factorization forest of height at most $|M|$.*

Proof. Using the bit representation, we create a dictionary, whose keys are elements of M, and whose values are subsets of positions in w. If the dictionary is based on a binary search tree, any key in the dictionary can be found in time k. For $m \in M$, consider the set X_m of positions in the word w which are labeled by m. In particular, each factor of w that begins in a position from X_m has value

m. In time linear in $k \cdot |w|$ we compute a dictionary which contains the nonempty sets X_m.

Define a linear order $\leq$ on elements of M, e.g. the lexicographic order on bit representations. We process the nonempty sets X_m according to the order. For each X_m, we define a family of factors $\mathcal{F}_m$ as follows. Consider two consecutive positions $j < l$ in

$$Y_m = \bigcup_{n \leq m} X_n \cup \{1, |w|\},$$

If at least one of j, l is in X_m, we add to $\mathcal{F}_m$ the factor that begins in position j and ends in position $l-1$. The size of $\mathcal{F}_m$ and the time to compute it is linear in X_m (as a byproduct we compute Y_m based on the previous Y_m). It is not hard to see that $\mathcal{F} = \bigcup_i \mathcal{F}_i$ is a β-homogeneous factorization forest. □

From a $\mathcal{J}$-homogeneous forest to an $\mathcal{H}$-homogeneous one. We first present an auxiliary lemma. A *partial factorization forest* is defined like a factorization forest, but its leaves need not be singletons. We also require that any factor is the union of its children, a requirement which is redundant when leaves are singletons.

Lemma A.5. *Let $F_1, \ldots, F_k$ be consecutive factors that are $\mathcal{J}$-homogeneous. In time $\mathrm{poly}(Q) \cdot k$, we can construct an $\mathcal{R}$-homogeneous partial factorization forest with leaves $F_1, \ldots, F_n$ and height at most $|M_Q|$.*

Proof. Let F be the union $F_1 \cup \cdots \cup F_k$. Recall the notation $\mathit{right}(F, G)$ defined in the part of Section 1 about logarithmic querying. We will treat the factors $F_1, \ldots, F_k$ as letters in a word

$$v = r_1 \cdots r_k \in (M_Q)^*,$$

where r_i is the image of $F_i \cup \mathit{right}(F_i, F)$. Apply Lemma A.4 to the word v with $M = M_Q$, yielding a factorization forest $\mathcal{G}'$. By expanding the i-th letter of v to the factor F_i, we can convert the factorization forest $\mathcal{G}'$ into a partial factorization forest $\mathcal{G}$ with leaves $F_1, \ldots, F_k$ and root F.

We claim that $\mathcal{G}$ satisfies the statement of the lemma. Its height is at most $|M|$ by Lemma A.4, so we only need to show that it is $\mathcal{R}$-homogeneous.

Consider a set of at least three siblings $G_1, \ldots, G_m$ in $\mathcal{G}$. From the way $\mathcal{G}$ was constructed, we know that for each $i \in \{1, \ldots, n\}$ the image of $G_i \cup \mathit{right}(G_i, F)$ is the same. Note that for each i, G_i is a prefix of $G_i \cup \mathit{right}(G_i, F)$, and both are $\mathcal{J}$-equivalent. Consequently, by Lemma A.2, they must be $\mathcal{R}$-equivalent. It follows that $G_1, \ldots, G_m$ are all $\mathcal{R}$-equivalent. □

We use the above lemma to upgrade a $\mathcal{J}$-homogeneous factorization forest $\mathcal{F}$ to an $\mathcal{R}$-homogeneous one, call it $\mathcal{G}$. We will simply add factors to $\mathcal{F}$. Initially, $\mathcal{G} = \mathcal{F}$. We process each maximal set of siblings $\mathcal{S} = \{F_1, \ldots, F_k\}$ from $\mathcal{F}$. (In most cases, $\mathcal{S}$ consists of all the children of a common parent, the exception is

when $\mathcal{S}$ is the roots of $\mathcal{F}$.) If $\mathcal{S}$ has at most two factors, we do not need to do anything. Otherwise, by assumption on $\mathcal{J}$-consistency of $\mathcal{F}$, we can apply the above lemma to the factors in $\mathcal{S}$, and add all factors of the resulting factorization forest $\mathcal{G}_{\mathcal{S}}$ to $\mathcal{G}$. Note that the added factors from $\mathcal{G}_{\mathcal{S}}$ are all included in $\bigcup \mathcal{S}$, so $\mathcal{G}$ is a factorization forest. The processing time needed to compute $\mathcal{G}$ is linear in the number of factors in all the sets $\mathcal{S}$, which is simply the number of factors in $\mathcal{F}$. Finally, if $\mathcal{G}$ contains a set of at least three siblings, then these siblings were added in some $\mathcal{G}_{\mathcal{S}}$, and hence they all have the same $\mathcal{R}$-class.

By a symmetric argument we upgrade the factorization forest $\mathcal{G}$ to an $\mathcal{L}$-homogeneous one, call it $\mathcal{H}$. But $\mathcal{H}$ is also $\mathcal{R}$-homogeneous, as already $\mathcal{G}$ was such. (If a factorization forest $\mathcal{G}$ is f-homogeneous for some function f, and a factorization forest $\mathcal{H}$ contains more factors than $\mathcal{G}$, then $\mathcal{H}$ is also f-homogeneous.) Thus $\mathcal{H}$ is $\mathcal{H}$-homogeneous.

From $\mathcal{H}$-consistency to α-consistency. In this section we show how to upgrade an $\mathcal{H}$-homogeneous factorization forest to an α-homogeneous one. The structure of the proof is the same as in the previous case, we only need a new version of Lemma A.5.

Lemma A.6. *Let $F_1, \ldots, F_k$ be consecutive factors that are $\mathcal{H}$-homogeneous. In time $\mathrm{poly}(Q) \cdot k$, we can construct an α-homogeneous partial factorization forest with leaves $F_1, \ldots, F_n$ and height at most $|M_Q|$.*

Proof. We use the same approach as in Lemma A.5. Let F be $F_1 \cup \cdots \cup F_k$. We treat each of the factors $F_1, \ldots, F_k$ as a letter in a word

$$v = r_1 \cdots r_k \in (M_Q)^*$$

where r_i is the image of $F_i \cup \mathit{right}(F_i, F)$. Apply Lemma A.4 to the word v with $M = M_Q$, yielding a factorization forest $\mathcal{G}'$. Replacing each letter r_i by the factor F_i, we convert $\mathcal{G}'$ into a partial factorization forest $\mathcal{G}''$ with leaves $F_1, \ldots, F_k$. Then for each maximal set of siblings $G_1, \ldots, G_m$ (for $m > 2$) we add the factor $G_1 \cup \cdots \cup G_{m-1}$, getting a partial factorization forest $\mathcal{G}$.

We claim that $\mathcal{G}$ is α-homogeneous. We argue as in Lemma A.5: consider a maximal set of at least three siblings. The only possibility is that these are $G_1, \ldots, G_{m-1}$ among some maximal set of siblings $G_1, \ldots, G_m$ from $\mathcal{G}''$. From the way $\mathcal{G}$ was constructed, we know that for each i the images of $G_i \cup \mathit{right}(G_i, F)$ is the same. Let us write $g_1, \ldots, g_m$ for the images of $G_1, \ldots, G_m$; these satisfy

$$g_i \circ g_{i+1} \circ \cdots \circ g_m = g_{i+1} \circ \cdots \circ g_m \qquad \text{for all } i < m.$$

The following well known lemma on Green's relations completes the proof of Lemma A.6, since it shows that all the elements $g_1, \ldots, g_{m-1}$ must be equal, as they all represent the group identity. (In a group, if $g \circ h = h$ holds, then g must be the group identity.) □

Fact A.7. *Let H be a $\mathcal{H}$-class in a finite monoid M. If there exist $s, t \in H$ such that $s \cdot t \in H$, then H is a group.*

Proof. Take $a, b \in H$ such that $a \cdot b \in H$ and take any $d \in H$. Since $b \sim_{\mathcal{R}} d$, then $a \cdot b \sim_{\mathcal{R}} a \cdot d$, so even more $d \sim_{\mathcal{J}} a \cdot b \sim_{\mathcal{J}} a \cdot d$. On the other hand $a \cdot b \sim_{\mathcal{L}} d$ (because both are in H) and $d \sim_{\mathcal{L}} a \cdot d$ (from Lemma A.2). Hence $a \cdot b \sim_{\mathcal{R}} a \cdot d$ and $a \cdot b \sim_{\mathcal{L}} a \cdot d$, so $a \cdot d \in H$. Symmetrically we may show that if $a \cdot d \in H$ for some $a, d \in H$, then also $c \cdot d \in H$ for any $c \in H$. This shows that $c \cdot d \in H$ for any $c, d \in H$.

Take any $a \in H$. Since M is finite it has to be $a^n = a^{2n}$ for some positive n. It is $a^n \in H$. Denote a^n as $\mathbf{1}_H$ (for some fixed a). We have $\mathbf{1}_H \cdot \mathbf{1}_H = \mathbf{1}_H$. This will be the neutral element in H. Indeed, take any $b \in H$. We may write $b = m \cdot (b \cdot \mathbf{1}_H)$ for some $m \in M$. Then $b = m \cdot b \cdot \mathbf{1}_H = m \cdot b \cdot \mathbf{1}_H \cdot \mathbf{1}_H = b \cdot \mathbf{1}_H$. Symmetrically $\mathbf{1}_H \cdot b = b$.

To conclude that H is a group it is enough to show that each element has an inverse. Take any $a \in H$. For some positive n there is $a^n = a^{2n}$. As above $b \cdot a^n = b$ for any $b \in H$. So $a^n = \mathbf{1}_H \cdot a^n = \mathbf{1}_H$, hence a^{n-1} is an inverse of a. □

B Binary trees

A binary (data) tree is a (data) tree where every node has at most two children. In the proofs, it will be convenient to deal with binary tress. In this part of the appendix, we show that binary trees can be considered without loss of generality.

A *nondeterministic binary tree automaton* is given by: a state space Q, an alphabet A, a set of accepting states $F \subseteq Q$, and a set of transitions

$$\Delta \subseteq \bigcup_{i \in \{0,1,2\}} Q \times A \times Q^i \ .$$

A *run* is a mapping of tree nodes to states that is consistent with Δ in the following sense: a) the root is mapped to an accepting state; and b) for every node x with $i \in \{0, 1, 2\}$ children, there is a transition in $Q \times A \times Q^i$ such that the state in x is the first coordinate, the label in x is the second coordinate, and the states in the children are the remaining coordinates. A *partial run* is like a run, but it does not need to be defined for every node; the consistency condition a) is checked only if the partial run is defined in the root; and the consistency condition b) is checked only for nodes x such that x and all of its children have defined values in the partial run.

A data aggregate transducer on binary trees is one where the underlying automaton is of the kind above.

If t is an unranked (data) tree, we write $enc(t)$ for the usual first-child/next-sibling encoding. This encoding, and its inverse, can be computed in linear time. The following lemma, which can be proved in a standard way, allows us to talk about binary trees from now on.

Lemma B.1. *Let $\hat{f}$ be a data aggregate transducer on unranked data tees. In polynomial time we can compute a data aggregate transducer on binary trees $enc(\hat{f})$ such that for any unranked data tree $t \otimes \mu$,*

$$enc(\hat{f}(t \otimes \mu)) = (enc(\hat{f}))(enc(t \otimes \mu))$$

C Aggregate transducers

In this part of the appendix, we sketch an argument that aggregate transducers (without data) are interesting in their own right. The general idea is to consider the data-free case of the whole approach to evaluating XPath on data trees by using data aggregate transducers. Many of the constructions are drastically simplified (in particular, there is no need for factorization forests); but the results are still interesting.

The first observation is that an aggregate transducer can be evaluated efficiently: linear data complexity, and polynomial combined complexity.

Lemma C.1. *Suppose that $\sqcup f$ is an aggregate transducer recognized by a nondeterministic automaton with states Q. The value $\sqcup f(s)$ can be calculated in time* $\text{poly}(Q) \cdot |s|$.

Proof. Thanks to the results from Section B, we can use binary trees. Let A, B be the input and output alphabets of f. For each node x of s, we calculate

- $down_x$ is the set of states q such that for some tree $t \in \text{trees}(B)$, there is a partial run on $s \otimes t$ that is defined on x and its desecendants, and uses state q in x.
- up_x is the set of states q such that for some tree $t \in \text{trees}(B)$, there is a partial run on $s \otimes t$ that is defined on nodes that are not proper descendants of x, and uses state q in x.
- B_x is the set of labels $b \in B$ such that for some tree $t \in \text{trees}(B)$, the automaton accepts $s \otimes t$ and t has label b in x.

We first calculate the sets $down_x$ in a bottom-up pass through s; next we calculate the sets up_x in a top-down pass; finally use both sets to calculate the sets B_x. The sets B_x determine the output $\sqcup f(s)$, since the label of a node x in $\sqcup f(s)$ is the maximal letter in B_x. □

We now show how compositions of aggregate transducers can be used to capture Regular Core XPath, which is the variant of Regular XPath for trees without data, see e.g."Navigational XPath: calculus and algebra" by Balder ten Cate and Maarten Marx in Sigmod Record Volume 36, Issue 2. It is well known that Regular Core Xpath can be translated into automata; the contribution here is that the translation is polynomial, and not exponential.

The characteristic function of a unary query of Regular Core Xpath with input alphabet A is the function

$$\mathit{char}(\varphi) : \text{trees}(A) \to \text{trees}(\{0, 1\})$$

which maps each tree s to a tree with the same nodes, where the label of each node x indicates if the node is selected by φ.

Lemma C.2. *Let φ be a unary query of Core XPath. There are aggregate transducers $\sqcup f_1, \ldots, \sqcup f_n$ such that the characteristic function of φ is $\sqcup f_1 \circ \cdots \circ \sqcup f_n$. The number n, the state spaces of the automata recognizing the aggregate transducers, and the time to compute them are all polynomial in φ.*

Proof. Using the same techniques as in Section G.
□

D Evaluation of data aggregate transducers on data words

In this section we prove Proposition 2.3, which is the only missing element of the algorithm evaluating data aggregate transducers on data words. As the input we have snippets $s_1, \ldots, s_m$. We want to output their aggregation $s_1 \sqcup \cdots \sqcup s_m$. It will be done separately for snippets $\sqcup f(w \otimes X) \sqcap Y$ in which Y is disjoint with X ("no" snippets), and separately for those in which Y is contained in X ("yes" snippets); then the two results will be aggregated together. Hence assume that for all snippets Y is disjoint with X (the other case is done in exactly the same way, with the only difference that we use the factorization forest $\mathcal{F}_1$ instead of $\mathcal{F}_0$)[3].

The snippets $\sqcup f(w \otimes X) \sqcap Y$ in which $Y \in \mathcal{P}(\mathcal{F}_0)$ (where $\mathcal{P}(\mathcal{F}_0)$ is the structure defined in Section A.1) will be called *structural* snippets and the snippets in which $Y = F_1 \cup \cdots \cup F_k$ for siblings from $\mathcal{F}_0$ will be called *neighbor* snippets.

Consider a snippet $\sqcup f(w \otimes X) \sqcap Y$. We decompose Y according to Lemma A.1 into $Y_1, \ldots, Y_n, Y'$. This gives a decomposition of the snippet into structural and neighbor snippets:

$$\sqcup f(w \otimes X) \sqcap Y = (\sqcup f(w \otimes X) \sqcap Y') \sqcup \bigsqcup_{1 \leq i \leq k} (\sqcup f(w \otimes X) \sqcap Y_i).$$

Hence we may replace the original snippet by the new ones. We do this for each of the snippets; the number of snippets increases by $\mathrm{poly}(Q)$.

A first observation is that for each factor we need only a constant number of snippets.

Lemma D.1. *Let Y be a factor and S a set of "no" snippets of the form $\sqcup f(w \otimes X) \sqcap Y$. Then a subset $S' \subseteq S$ of cardinality at most $|Q|^2$ can be chosen such that $\bigsqcup S = \bigsqcup S'$. Moreover, S' can be calculated in time $\mathrm{poly}(Q) \cdot |S|$.*

Proof. By Q_S denote the union of external types of the snippets from S (more precisely: external types of the factors Y in the word w_0, for each snippet $\sqcup f(w \otimes X) \sqcap Y$). Then for each pair $(p, q) \in Q_S$ we take to S' one (any) snippet from s which has (p, q) in its external type. Each run of the automaton in Y allowed by some snippet from S uses particular states: p just before Y and q just after Y, for $(p, q) \in Q_S$. Hence this run is allowed also by some snippet from S', the one taken for this pair of states. □

A second observation allows us to reduce the number of neighbor snippets.

[3] In fact the combined size of all "yes" snippets generated by Lemma 2.2 is $|w|$, hence we can evaluate them directly. The real problem is only with "no" snippets.

Lemma D.2. *Let $F_1, \dots, F_n$ be consecutive α-homogeneous siblings in $\mathcal{F}_0$ and S a set of "no" neighbor snippets of the form $\sigma = \sqcup f(w \otimes X^\sigma) \sqcap (F_{i(\sigma)} \cup \dots \cup F_k)$, i.e. they all end on the same F_k, but may begin on different $F_{i(\sigma)}$. Then there exists a set S' of neighbor snippets ending on F_k and a set S'' of structural snippets, such that $\bigsqcup S = \bigsqcup S' \sqcup \bigsqcup S''$, and $|S'| \leq |Q|^2$, $|S''| \leq |S|$. Moreover, the sets can be calculated in time* $\mathrm{poly}(Q) \cdot (|S| + |w|)$.

Proof. First we split each snippet σ into two snippets. The part $\sqcup f(w \otimes X^\sigma) \sqcap F_{i(\sigma)}$ is taken to S''; it is a structural snippet. The part from $F_{i(\sigma)+1}$ to F_k is taken to $\widetilde{S}'$; it is a neighbor snippet. The problem is that $\widetilde{S}'$ is too big. In a second step, for each pair of states (p, q) we take to S' the longest snippet from $\widetilde{S}'$ (i.e. this with $i(\sigma)$ as small as possible) among those containing (p, q) in its external type. If there is no such snippet, we do not take any; if there are many longest, we take any of them.

We have to proof that $\bigsqcup \widetilde{S}' = \bigsqcup S'$, as obviously $\bigsqcup S = \bigsqcup \widetilde{S}' \sqcup \bigsqcup S''$. Take any label generated by some snippet $\sigma' \in \widetilde{S}'$, which was created as a part of a snippet $\sigma \in S$. The label was generated by a run of $\mathcal{A}$ having some state p just before $F_{i(\sigma)+1}$ and some state q just after F_k. For the pair (p, q) some snippet $\tau' \in \widetilde{S}'$ was taken to S', which was created as a part of a snippet $\tau \in S$. It holds $i(\tau) \leq i(\sigma)$. Because (p, q) is contained in the external type of τ', there exists p' such that (p', q) is contained in the external type of τ and $(p', p) \in \alpha(w_0[F_{i(\tau)}])$. Because the split is consistent with α, it holds $\alpha(w_0[F_{i(\tau)}]) = \alpha(w_0[F_{i(\tau)} \cup \dots \cup F_{i(\sigma)}])$, hence τ' also allows a run which has p just before $F_{i(\sigma)+1}$ and q just after F_k. □

We process the neighbor snippets in a right-to-left pass through each sequence of siblings in $\mathcal{F}_0$. When we are in a factor F_k, we eliminate neighbor snippets ending in F_k. First we reduce their number using Lemma D.2, so that only $|Q|^2$ are left. Then we split each of them into F_k and the rest, which results in a structural snippet and a neighbor snippet ending in F_{k-1}. The snippets of the second kind are processed again later, when we are in F_{k-1}. Lemma D.2 ensures that the number of snippets is always small, hence the running time is linear in $|t| + |S|$ and polynomial in $|Q|$.

The only thing left is to simplify the structural snippets, which is possible thanks to the property ♯, given in Section A.1. We start from the longest snippets and we move towards shorter. For each Y in $\mathcal{P}(\mathcal{F}_0)$, we first reduce the number of snippets to $|Q|^2$ using Lemma D.1, and then we decompose them into Y_1 and Y_2 (such that $Y_1 \cup Y_2 = Y$), getting shorter snippets, which are processed again later.

E Factorization forests for trees

Here we prove the tree case of Theorem 2.1. We begin by defining the factorization forests for trees.

Thanks to the discussion in Section B, we may assume that all trees are binary. For two nodes x, y we write $x \leq y$ ($x < y$) to say that x is a (proper) ancestor of y.

The definition of a factorization forest is in a slightly different style than for words: we will use forward ramseyan splits [4]. A *split* is a function

$$split : \text{nodes}(t) \to \{1, \ldots, K\}.$$

The number K is called the *height* of the split.

Consider a function α which maps node pairs $x < y$ to elements of a finite monoid M. Such a function is called a *morphism* when for any three nodes $x \leq y \leq z$,

$$\alpha(x, z) = \alpha(x, y) \circ \alpha(y, z).$$

Given a split, two nodes $x \leq y$ are called *neighbors* if their split values are equal and no node between them has smaller split value. A split is α-*homogeneous*[4] if for any $x_1 < x_2$ and $y_1 < y_2$ which are all neighbors (in particular all are on one path from the root to a leaf) it holds

$$\alpha(x_1, x_2) = \alpha(x_1, x_2) \circ \alpha(y_1, y_2).$$

Note how this is a different requirement than in the word case. A theorem by Colcombet says that such a split can be constructed.

Theorem E.1 ([4]). *Let t be a tree, M a finite monoid, and α a morphism into M given by its values for pairs x, y in which x is the parent of y. Then an α-homogeneous split of height $K = O(|M_Q|)$ exists and can be computed by a deterministic transducer, hence in time linear in the tree size.*

The state space of the transducer is linear in M_Q, and therefore the constant in the linear time is also linear in M_Q. The key point is that the height K does not depend on the tree, only on the monoid. Additionally, we may assume that the value assigned to the root is 1.

Besides of the split we keep the following information, for each $1 \leq k \leq K$:

A) an unranked tree s_k consisting of nodes of t having split value at most k (a node is a child of an other node in s_k if it is its descendant in t and each node between them has a split value greater than k);
B) for each node x, a pointer to its closest proper ancestor y being in s_k (i.e. with the split value at most k).

Denote the set of pointers (pairs of nodes) from B as $\mathcal{P}(split)$. Observe that the information of both types can be constructed in time linear in the tree size (e.g. separately for each k). Moreover we have the following fact which we use for the trees s_k. This fact comes from "The LCA Problem Revisited", by M. Bender and M. Farach-Colton (LATIN 2000), or "Fast algorithms for finding nearest ancestors", by D. Harel and R. Tarjan (SICOMP 1984).

Fact E.2. *There is a data structure, which*

– *for a given tree t, can be constructed in time $O(|t|)$, and can be used to*

[4] Colcombet uses the name *forward ramseyan split* instead.

– *find for any nodes x, y their closest common ancestor in time $O(1)$.*

Corollary E.3. *There is a data structure, which*

– *for a given unranked tree t, can be constructed in time $O(|t|)$,*
– *then, for a given node x and its descendant y, one can read which child of x is an ancestor of y, in time $O(1)$.*

The additional information is prepared in such a way that the following lemma holds.

Lemma E.4. *For two nodes $x < y$ we can compute a sequence of nodes $x = x_0 < x_1 < \cdots < x_n = y$ such that every two consecutive nodes are either neighbors or are connected by a pointer from $\mathcal{P}(split)$; both n and the running time is $O(K)$ (i.e. constant in the tree size and in the distance between x and y).*

Proof. The proof is by induction on $split(x) + split(y)$. When x and y are neighbors, we are done (it is the case when $split(x) = split(y)$ and their pointers in B for $k = split(x) - 1$ point to the same node). Otherwise there are two cases. First assume $split(x) \leq split(y)$. Then from B we read a closest ancestor z of y with a split value smaller than $split(y)$. It has to be x or a descendant of x, as otherwise x and y would be neighbors. Now z can be the last element of the sequence and we can proceed inductively for x and z, which have a smaller sum of split values.

Otherwise $split(x) > split(y)$. Here we have to do something similar, but the problem is that pointers from B go only up. Let $k = split(x) - 1$. From x we go to its closest ancestor z with $split(z) \leq k$. Then we use Fact E.3 to find in s_k the child z' of z which is an ancestor of y (or is equal to y); if y is not in s_k we first move to its closest ancestor being in s_k. Now again from z' we go to its closest ancestor z'' being in s_{k+1}. Note that x and z'' are neighbors (possibly $x = z''$), as no node between them is in s_k. Hence we may use z'' and z' as first two elements of the sequence, and then proceed inductively for z' and y.

Note that each step is done in constant time, and the number of steps is limited by $split(x) + split(y)$, which is $O(K)$. □

Now observe that the set $\mathcal{P}(split)$ defined above satisfies the following property ♯♯: for each pair (x, y) in $\mathcal{P}(split)$,

– x is the parent of y, or
– there is z such that $(x, z) \in \mathcal{P}(split)$ and $(z, y) \in \mathcal{P}(split)$ (moreover, $\mathcal{P}(split)$ is organized such that z and these pairs can be found in constant time).

Indeed, as z we can take the parent of y. This is a tree replacement of property ♯ for words (from Section A.1). From this property follows that we can calculate $\alpha(x, y)$ for each pair in $\mathcal{P}(split)$ in time linear in $|\mathcal{P}(split)|$, hence linear in $|t|$. It is an easy dynamic algorithm: we calculate the values for longer pointers using the results for shorter ones.

Corollary E.5. *Given the α-homogeneous split for a tree t (together with the additional information), one can for any given nodes $x < y$ calculate $\alpha(x, y)$ in time linear in the height of the split.*

Proof. Thanks to Lemma E.4 it is enough to calculate α when x and y are neighbors or (x, y) is in $\mathcal{P}(split)$ (and then compose the values). In the second case $\alpha(x, y)$ is known, as noted above. Let now x and y be neighbors and let x' be the closest neighbor of x such that $x < x' \leq y$. Node x' can be read from $s_{split(x)}$ using Fact E.3. As the split is α-homogeneous, we know that

$$\alpha(x, y) = \alpha(x, x') \cdot \alpha(x', y) = \alpha(x, x').$$

However we have a pointer in $\mathcal{P}(split)$ from x' to x, hence the last value is known. □

F Evaluation of data aggregate transducers on data trees

In this section, we prove the tree case of Theorem 2.1, which says that split transducers can be evaluated in linear time. We fix a split transducer $\widehat{f}$ and a data tree $t \otimes \mu$. We want to compute the output $\widehat{f}(t \otimes \mu)$.

Thanks to the discussion in Section B, we are working on binary trees. Recall that a split transducer is given by a tree automaton (which we assume to be a nondeterministic automaton on binary trees), as well a linear order on the output alphabet. Suppose that for the split transducer $\widehat{f}$ this automaton is $\mathcal{A}$, with states Q. The input alphabet of f is $A \times \{0, 1\}$ and the output alphabet is B.

Unless otherwise stated, all the trees considered in this section will have the same nodes as t. They will either be t, or trees of the form $t \otimes X$ for some set of nodes X, or outputs in $f(t \otimes X)$. In particular, the alphabets will be either A (for t), or $A \times \{0, 1\}$ (for the trees $t \otimes X$), or B (for the outputs in $f(t \otimes X)$).

We present our proof strategy in Section F.2. First we introduce some terminology.

F.1 Zones, types and snippets

Zones. We use the name *zone* for a set of nodes in t. We write X, Y, Z for zones. The complement $\overline{X}$ of a zone X is with respect to the nodes of t. We will be mainly interested in *prime* zones, which are of three kinds: node, tree, or context. A *node zone* consists of a single node. A *tree zone* is given by a node, called its root; the zone contains the root and all of its descendants. A *context zone* is given by two nodes, called the root and the hole, one a proper descendant of the other; it contains the root node and its descendants, excluding the hole its descendants. Hence it is a difference of two tree zones: one rooted in the root and the other rooted in the hole.

Prime zones are to trees what factors are to words.

Fact F.1. *Any set X of nodes can be partitioned into at most $O(|\overline{X}|)$ prime zones.*

Proof. Let Y be all the nodes in $\overline{X}$ together with their closest common ancestors. The complement of Y can be partitioned into at most $2|Y| + 1$ prime zones: contexts and trees, which are given by nodes from Y as holes, and children of nodes from Y and the root of the tree as roots. Then, we add node zones for $Y - \overline{X}$. □

We will need to quickly compute this partition, given $\overline{X}$. For that we need a procedure calculating closest common ancestors (Fact E.2) and a little more.

Fact F.2. *The partition from Fact F.1 can be computed in time $O(|\overline{X}|)$, when the nodes of $\overline{X}$ are given in document order.*

Proof. A skeleton is a tree containing nodes from Y; a node $x \in Y$ is a child of other node $y \in Y$ if it is its descendant in t and no node between them is in Y. It is a binary tree (each node has at most two children). Calculating the skeleton is enough, we can easily read the partition from it. Note that it is also necessary: having just a list of roots and a list of holes is not enough, we need to know how they are paired.

We process the nodes of $\overline{X}$ in the document order (from left to right). At every moment we already have a skeleton for some subset of $\overline{X}$, and all other nodes from $\overline{X}$ are later in the document order. We want to add the next node $y \in \overline{X}$. We find the closest common ancestor z of this new node y and the rightmost already processed node $x \in \overline{X}$. We need to add z in the appropriate place in the skeleton. We compare z with the nodes on the rightmost path of the skeleton, starting from x and going up (note that Fact E.2 allows us also to check if a node is a descendant of other node). When z is between some node and its parent in the skeleton, we add it there, together with attached y. It is also possible that $z = x$ or that z is over the root of the current skeleton.

Why does it work in linear time? Potentially there are many nodes on the rightmost path of the current version of a skeleton. However always only one of the visited nodes is an ancestor of z. Other visited nodes, which are not ancestors of z no longer will be on the rightmost path, so every node can be visited only once in that role. □

Types of prime zones. Like for words, we define two notions of type for prime zones: an internal type, and an external type.

We first define the *internal type* of a prime zone Y inside a tree $t \otimes X$, where X is some class. The definition is by cases, depending on whether Y is a tree, node or context zone. If Y is a tree zone, then the internal type is the set of states q such that for some $s \in \text{trees}(B)$, there is a partial run of the tree automaton on $t \otimes X \otimes s$ that is defined on nodes from Y and has state q in the root of Y. (Note that only the labels of s in nodes from Y are relevant to this definition, a similar situation will hold for context and node zones.) If Y is a context zone, then its internal type is a set of state pairs: it contains a pair (p, q) if there is

some tree $s \in \text{trees}(B)$ and a partial run of the tree automaton on $t \otimes X \otimes s$ that is defined on nodes from Y (and the hole), and has state p in the root of Y and state q in the hole of Y. Likewise for a node zone, but this time we get a set of state triples: we take a state in the node and in its two children (if one or two of the children does not exists, the type consists of pairs or single states). The *external type* of a prime zone Y is defined in the same way, but now we consider mappings which are consistent in the complement of Y.

The internal types are compositional in the following sense: if a prime zone is partitioned into several smaller prime zones, only the internal types of the smaller prime zones are needed to determine the internal type of the larger prime zone. The idea behind the external type is that it describes the type of the zone's complement. The following lemma says external types can be deduced from internal types.

Lemma F.3. *Let $\mathcal{X}$ be a partition of the nodes of a tree s into prime zones, whose internal types in s are known. The external types of these zones in s can be calculated in time* $\text{poly}(Q) \cdot |\mathcal{X}|$.

Proof. Let X be the set of nodes from the node zones in $\mathcal{X}$ and the roots of the context zones in $\mathcal{X}$; it is linear in the size of $\mathcal{X}$. We use compositionality of internal types to calculate, in a leaves-root pass, for each node $x \in X$ the internal type of the tree zone given be x. Likewise, in a root-leaves pass, we calculate for each node $x \in X$ the internal type of the context whose root is the root of s and whose hole is x. The internal types calculated in these two passes are all sufficient to give all the external types of zones in $\mathcal{X}$. □

Snippets. Like for words, a *partial output* is any value $(\sqcup f(t \otimes X)) \sqcap Y$, where X is a class and Y is a set of nodes; a *snippet* is a partial output $(\sqcup f(t \otimes X)) \sqcap Y$ where Y is a prime zone that is either disjoint with X (a *"no" snippet*), or included in X (a *"yes" snippet*).

The point of snippets is that they can be represented by a constant number of pieces of information: at most two nodes to represent the prime zone Y, an external type of a prime zone, and a "yes"/"no" bit. This is described by the following observation.

Observation F.4. *Let Y be a prime zone that is disjoint with or included in a zone X. The value*

$$(\sqcup f(t \otimes X)) \sqcap Y$$

of the snippet (for a given t and f) depends only on Y, the external type of Y in $t \otimes X$, and whether the snippet is a "yes"/"no" snippet.

We use the name *snippet representation* for the above information. This representation has size $\text{poly}(Q)$. (We assume that nodes are stored using unit cost.)

F.2 Proof strategy

We now have the necessary concepts to present our proof strategy for Theorem 2.1. Our goal is to produce the output of the split transducer $\widehat{f}$ on a data

tree $t \otimes \mu$. This output is, by definition,

$$\widehat{f}(t \otimes \mu) = \sqcup_{X \text{a class of } \mu} \sqcup f(t \otimes X).$$

Our algorithm will be manipulating sets of snippets. Whenever a subroutine inputs or outputs a set of snippets, we assume that the snippets are given by their constant size representations, as described before Observation F.4.

The algorithm works in three stages. First, we construct factorization forests. Then, we show that for each class X, the output $\sqcup f(t \otimes X)$ can be represented by a small number of snippets. Finally, we show that these snippets can be efficiently aggregated.

Stage 1. In the first stage, we compute two factorization forests. Consider two trees

$$t_0 = t \otimes \emptyset, \qquad t_1 = t \otimes \overline{\emptyset}$$

over alphabet $A \times \{0, 1\}$. We want to compute factorization forests (splits) for these words. We use the morphism α which maps a pair of nodes $x < y$ into the internal type of the context having the root in x and the hole in y. Apply Theorem E.1 to the trees t_0, t_1 and the morphism α, yielding factorization forests $split_0$, $split_1$. These factorization forests will be used by the next two stages of the algorithm. As an input of Theorem E.1 we have to give the values of α for pairs (x, y) in which x is the parent of y. Note that these values can be computed in linear time, as a context zone with the root in the parent of the hole can be decomposed into a tree zone and a node zone. The internal types of all tree zones can be computed in one leaves-root pass.

Stage 2. As for words, we show that for each class X, the output $\sqcup f(w \otimes X)$ can be represented by a small number of snippets. This is stated by the following lemma.

Lemma F.5. *Let X be a zone. We can calculate a set S_X of snippets such that $\sqcup f(t \otimes X) = \sqcup S_X$. Both the cardinality of S_X and time to calculate it are linear in the number of nodes in X.*

Proof. Thanks to Fact F.2, we can calculate disjoint prime zones $Y_1, \ldots, Y_m$ that partition the complement of X. Moreover, we can assume that these are only node and context zones, as each tree zone can be partitioned into one node zone and one context zone (the node zone and simultaneously the hole is in any leaf). Both m and the time to calculate these zones are at most linear in $|X|$. Since $Y_1, \ldots, Y_m$ and X cover all the nodes of t, we have

$$\sqcup f(t \otimes X) = \quad \sqcup_{x \in X} (\sqcup f(t \otimes X) \sqcap \{x\}) \quad \sqcup \quad \sqcup_{i \in \{1,\ldots,m\}} ((\sqcup f(t \otimes X)) \sqcap Y_i).$$

The above gives a decomposition into snippets as required. The problem is that we need to calculate representations of these snippets. In order to do this, we need to know the external types, in $t \otimes X$, of the zones $\{x\}$ for $x \in X$ and the prime zones $Y_1, \ldots, Y_m$.

Thanks to Lemma F.3, all we need is the internal types of the zones. For the node zones this is easy: the internal type just depends on the label of the node. For the context zones Y_i, we remark that the internal type of Y_i in $t \otimes X$ is the same as the internal type of Y_i in $t \otimes \emptyset$, since X is disjoint with Y_i. Therefore, we may read the internal types of $Y_1, \ldots, Y_n$ in $t \otimes X$ from the factorization forest $\mathcal{F}_0$ (Corollary E.5). $\square$

Stage 3. Finally we aggregate the snippets. The following proposition is shown in the next subsection.

Proposition F.6. *Let $s_1, \ldots, s_m$ be snippets. Their aggregation $s_1 \sqcup \cdots \sqcup s_m$ can be calculated in time linear in $m + |t|$.*

F.3 Aggregating snippets

We now prove Proposition F.6, which says that for any set of snippets S, its aggregation $\sqcup S$ can be computed in time linear in the cardinality of S. We can treat the "yes" and "no" context snippets separately, and then aggregate the two partial outputs. Therefore, without loss of generality, we may assume that only one kind of snippets appears in S, say "no" snippets. The problem is that S might have quadratic *combined size* (the size of a snippet is the number of nodes with a defined value; the combined size of a set of snippets is the sum of the sizes of its snippets). If the combined size is small, then aggregation can be computed easily, as stated by the following lemma.

Lemma F.7. *For a set S of snippets, $\sqcup S$ can be calculated in time linear in the combined size of S, and polynomial in $|Q|$.*

A second observation is that for each prime zone we need only a constant number of snippets; this is a tree equivalent of Lemma D.1 and can be proved analogously.

Lemma F.8. *Let Y be a tree zone or a context zone and S a set of "no" snippets of the form $(\sqcup f(t \otimes X)) \sqcap Y$. Then a subset $S' \subseteq S$ of cardinality at most $|Q|^2$ can be chosen such that $\sqcup S = \sqcup S'$. Moreover, S' can be calculated in time linear in the cardinality of S, and polynomial in $|Q|$.*

The snippets that involve node zones can be aggregated in linear time using Lemma F.7. Furthermore, also the tree snippets can be quickly aggregated.

Lemma F.9. *For a set S of tree snippets, $\sqcup S$ can be calculated in time linear in $|t| + |S|$, and polynomial in $|Q|$.*

Proof. First in a leaves-root pass we may calculate the internal type of each tree zone in $t \otimes \emptyset$.

Then we process the tree in a root-leaves pass. For each tree zone Y we first reduce the number of snippets of the form $(\sqcup f(t \otimes X)) \sqcap Y$ using Lemma F.8. Then we split each of them into three snippets: a node snippet $\sqcup f(t \otimes X) \sqcap \{y\}$

and two tree snippets $(\sqcup f(t \otimes X)) \sqcap Y_L$, $(\sqcup f(t \otimes X)) \sqcap Y_R$, where y is the root of Y, and Y_L, Y_R are its two subtrees rooted in the children of y. The internal types of $Y_L, Y_R, \{y\}$ in $t \otimes X$ (needed to calculate their external types) are the same as in $t \otimes \emptyset$ (since X and Y are disjoint), hence known. The smaller tree snippets are then processed in the children of y.

Finally the node snippets are aggregated using Lemma F.7. The total running time is linear, because we were always eliminating redundant snippets, hence for each subtree we had a constant number of snippets. □

Our proof of Proposition F.6 for context snippets proceeds in two steps. In the first step, we split each context snippet from S into some number of context snippets in which the hole is a child of the root. However we can not calculate all of them, as there would be too many of them (quadratically many). We have to eliminate redundant ones, in the spirit of Lemma F.8. This way, we create a new set S' of context snippets with the hole in a child of the root, of cardinality linear in $|t|$, with the property $\sqcup S = \sqcup S'$. Each snippet in S' can be divided into a tree snippet and a node snippet, which can be aggregated using Lemmas F.7 and F.9. Thus it is enough to show the following proposition.

Proposition F.10. *For a set S of context snippets, we can calculate a set S' of snippets such that each context snippet from S' has the hole is a child of the root, and $\sqcup S = \sqcup S'$. The cardinality of S', as well as the running time, can be linear in $|t| + |S|$.*

Now we prove the above proposition. In a first step we use Lemma E.4 to partition each context snippet from S into context snippets in which the root and the hole are either neighbors (we call them *neighbor snippets*) or are connected by a pointer from $\mathcal{P}(split_0)$ (we call them *structural snippets*); we get a set $\widetilde{S}$ containing $O(|S|K)$ snippets. In a second step we will eliminate neighbor snippets; the following lemma will be useful. This is a tree equivalent of Lemma D.2, and can be proved analogously.

Lemma F.11. *Let x be a node and S a set of "no" neighbor snippets with the hole in x. Then there exists a set S' of neighbor snippets with the hole in x and a set S'' of structural snippets, such that $\sqcup S = (\sqcup S') \sqcup (\sqcup S'')$, and $|S'| \leq |Q|^2$, $|S''| \leq |S|$. Moreover, the sets can be calculated in time linear in $|S| + |t|$, and polynomial in $|Q|$.*

We process the neighbor snippets in a leaves-root pass. When we are in a node x, we eliminate neighbor snippets having the hole in x. First we reduce their number using Lemma F.11, so that only $|Q|^2$ are left. Then we decompose each of them in the closest neighbor $y < x$, which results in a structural snippet (from y to x) and a neighbor snippet with a hole in y. The snippets of the second kind are processed again later, when we are in y. Lemma F.11 ensures that the number of snippets is always small, hence the running time is linear in $|t| + |\widetilde{S}|$ and polynomial in $|Q|$.

The only thing left is to simplify the structural snippets. We start from the longest snippets and we move towards shorter. For each $x < y$ in $\mathcal{P}(split_0)$, we

first reduce the number of snippets to $|Q|^2$ using Lemma F.8, and then we split them at the node z (such that (x, z) and (z, y) are in $\mathcal{P}(split_0)$), getting shorter snippets, which are processed again later.

G XPath evaluation

G.1 Definition of Regular XPath

There are two types of expressions: programs and node tests. A *program* is a binary query. In each data tree, a program will select a set of pairs (x, y) of nodes. Intuitively a program will describe the path from x to y, although the path might not be the shortest one. A typical program is `next−sibling`, it selects a pair (x, y) if y is the right child of x (which corresponds to being the next sibling in the original XML document). A *node test*, on the other hand, is a unary query: it selects a set of nodes in a data tree. A typical node test is a, it selects nodes that are labeled by the tag name a. In general, the two types of expression are mutually recursive, as defined below:

- Every tag name a is a node test, which holds in nodes labeled by tag a.
- Node tests admit negation, conjunction and disjunction.
- There are two types of *atomic* programs. Every axis

  `first−child` `parent−of−first` `next−sibling` `prev−sibling`

  is an atomic program, which holds in pairs of nodes connected by this axis. Furthermore, a node test φ can be interpreted as an atomic program $[\varphi]$, which holds in pairs (x, x) such that φ holds in x.
- In general, a program is a regular expression over atomic programs. In other words, programs contain the atomic programs, the empty program ϵ, and are closed under union, composition and Kleene star. For instance, the program $\texttt{first−child} \cdot \texttt{next−sibling}^*$ select (x, y) if y is a child of x, i.e. it stands for `child` axis; and program ϵ selects all identity pairs (x, x), i.e. it stands for `self` axis.
- If α, β are programs then $\alpha \sim \beta$ is a node test. In a data tree $t \otimes \mu$, this query selects a node x if there exist nodes y, z in the same class such that (x, y) is selected by α and (x, z) is selected by β.
- Similar to the above, a node test $\alpha \not\sim \beta$ is also defined. Here, the requirement is that y and z are in different class.

Note that we allow the Kleene star in programs, while usually XPath does not (the extension is called Regular XPath). We do so because our techniques work even when the Kleene star is present. Also, the Kleene star allows us to use a smaller set of four axes and to encode trees of fixed height into words. When referring below to XPath or Regular XPath, we mean the fragment above. We have defined unary queries (node tests) and binary queries (programs). Boolean queries can be defined by taking a unary query, and choosing the matching trees where the root is selected.

G.2 Proof of Theorem 3.1

In the proof, we compose data aggregate transducers. We need to adjust the definitions a bit to make composition work, since the output of a data aggregate transducer is a tree without data. Consider alphabets A and $B_1, \ldots, B_n$. Let $\widehat{f_1}, \ldots, \widehat{f_n}$ be data aggregate transducers, where the input alphabet of $\widehat{f_i}$ is $A \times B_1 \times \cdots \times B_{i-1}$ and the output alphabet if B_i. Consider a data tree $t_0 \otimes \mu$ over alphabet A. We define t_i to be the output of $\widehat{f_i}$ on the data tree $t_0 \otimes \cdots \otimes t_{i-1} \otimes \mu$. The function, call it f, which maps $t_0 \otimes \mu$ to t_n is called a *composed data aggregate transducer*. The *state space* Q *of* f is defined to be the disjoint union of states spaces of the automata representing $\widehat{f_1}, \ldots, \widehat{f_n}$. By repeatedly applying Theorem 2.1 we can compute $t_n = f(t_0 \otimes \mu)$ in time $exp(Q) \cdot |t_0|$ in general, and in time $\text{poly}(Q) \cdot |t_0|$ when t_0 is a word.

To complete the proof, we will show that for every regular XPath query φ, there is a composed data aggregate transducer, whose state space is polynomial in φ, and which maps each data tree to the characteristic function of the set of nodes selected by φ.

Lemma G.1. *For every node test φ of Regular XPath, there is a composed data aggregate transducer, whose state space is polynomial in φ, and which maps each data tree to the characteristic function of the set of nodes selected by φ.*

Technically, the proof is simple; and no new techniques are used on top of what was done in, say [3] or [7]. The general idea is that composition of transducers is used to deal with composition of formulas; and the aggregate transducers are designed to capture the basic types of node test.

Product alphabets. In the construction, we consider alphabets of the form

$$A \times \{0,1\}^k$$

where k is polynomial in φ. Technically, such an alphabet is exponential (of course, each letter is represented in polynomial space). On the other hand, we want to use expressions and state spaces of automata that are polynomial in φ. To solve this problem, the automata and expressions will refer to the alphabet in a succinct way. For the automata, we assume that there is a polynomial algorithm, which given the representation of an input letter a computes the (polynomial size) set of transitions that involve this letter. For the expressions, we allow label tests (i.e. test for tag names) of the form: test if coordinate i has label a.

Reduction to unnested queries. A program α of Regular XPath is called *unnested* if all the node tests that appear in it are just label tests (possibly in the more general succinct form described above, where only some coordinate of the label is tested).

In the proof of Lemma G.1, we first reduce to the case when the node test φ only uses unnested programs. This reduction is standard; it uses the idea that a sub-transducer computes the sub-queries of φ.

Let $sub(\varphi)$ be all the node tests that appear as proper subformulas of φ. Let φ_{sub} be the query with input alphabet $A \times \{0,1\}^{sub(\varphi)}$ that is obtained from φ by replacing each node test $\psi \in sub(\varphi)$ by a label test which checks if the label has bit 1 on the coordinate corresponding to ψ. By construction, every program in φ_Γ is unnested. For a data tree $t \otimes \mu$, we define a tree t_{sub} without data over the alphabet $\{0,1\}^{sub(\varphi)}$. This tree has the same nodes as t; the label of a node says x which node tests from Γ are true in x.

The query φ_{sub} and the data tree t_{sub} are defined so that the nodes selected by φ_{sub} in the data tree $t \otimes t_{sub} \otimes \mu$ are the same as the nodes selected by φ in the data tree $t \otimes \mu$. By induction assumption, we have a composed data aggregate transducer for each $\psi \in sub(\varphi)$. Composing these transducers, we see that

$$t \otimes \mu \in \text{dtrees}(A) \qquad \mapsto \qquad t \otimes t_{sub} \otimes \mu \in \text{dtrees}(A \times \{0,1\}^{sub(\varphi)})$$

can be computed by a composed data aggregate transducer; which consists of a polynomial number of polynomial size data aggregate transducers (all polynomials in φ). All that remains to be done is finding a composed data aggregate transducer that evaluates the unnested query φ_{sub}. We do this below. The transducer for φ_{sub} that we will produce will also require composition; so the composition of transducers is required also to deal with unnested programs.

Computing loops. Let α be an unnested program with input alphabet A. An unnested program cannot refer to data, since data can only be tested via a subquery $\alpha \sim \beta$ or $\alpha \not\sim \beta$, which involves nesting. Therefore, it makes sense to talk the set $\alpha(t)$ of node pairs selected by α in a tree $t \in \text{trees}(A)$ without data.

When talking of a path here, we mean a sequence of nodes connected by parent/child edges in the binary encoding of the tree. Hence in the unranked tree it may go between a node and its first child (in any direction), and between consecutive siblings. The path may loop. Note that this corresponds to the set of axes in our definition of XPath[5]. In the natural way we say what it means for a path π to be consistent with a program α.

We now define the notion of a k-looping path. A 1-looping path is any path without repetition of nodes. A $(k+1)$-looping path is a k-looping path, or a path of the form

$$\sigma_0 x_1 \pi_1 x_1 \sigma_1 x_2 \pi_2 x_2 \cdots \sigma_{n-1} x_n \pi_n x_n \sigma_n$$

where $\sigma_0, \ldots, \sigma_n$ and $\pi_1, \ldots, \pi_n$ are paths, $x_1, \ldots, x_n$ are nodes (or paths of length 1), the path

$$\sigma_0 x_1 \sigma_1 \cdots \sigma_{n-1} x_n \sigma_n$$

[5] Note that in some sense this switching to binary tree is necessary. Consider the `next-sibling` axis. It does not correspond to any edge in the unranked tree. Moreover it can not be expressed as a combination of `parent` and `child` axes, as after going to the parent we loose the information from which child we have came.

is 1-looping and the paths $\pi_1, \ldots, \pi_n$ are k-looping. We define $\alpha_k(t)$ to be the set of node pairs (x, y) in t such that some k-looping path in t that is consistent with α can go from x to y. By a pumping argument, one sees that $\alpha_k(t)$ grows and then stabilizes after k exceeds some polynomial in α. This stabilized value is the set of node pairs $\alpha(t)$ that are selected by α in t.

Let $sub(\alpha)$ be the set of subprograms in α. We define $loops_{\alpha,k}(t)$ to be the tree where node x is labeled by the set of programs

$$\{\beta \in sub(\alpha) : (x, x) \in \beta_k(t)\}.$$

The following lemma, which can be proved in a standard way, shows that the mapping $loops_{\alpha,k}$ can be computed by a composition of aggregate transducers (without data).

Lemma G.2. *The function $loops_{\alpha,k} : \text{trees}(A) \to \text{trees}(P(sub(\alpha)))$ is a composition of k aggregate transducers, each with state space polynomial in α.*

Unnested node tests. We now show that any node test φ that only contains unnested programs can be computed by a composed data aggregate transducer. The interesting case is when φ is of the form $\alpha \sim \beta$ or $\alpha \not\sim \beta$, where α, β are unnested programs. We omit the simple proof the following lemma.

Lemma G.3. *We can compute a tree automaton $\mathcal{A}_{\alpha,\beta}$ with $\text{poly}(\alpha, \beta)$ states, input alphabet*

$$A \times P(sub(\alpha)) \times P(sub(\beta)) \times \{0, 1\} \times \{0, 1\}$$

and the following property. For any tree $t \in \text{trees}(A)$, any set of nodes X, and any node x, the automaton accepts the tree

$$t \otimes loops_\alpha(t) \otimes loops_\beta(t) \otimes X \otimes \{x\}$$

if and only if there are nodes $y, z \in X$ with $(x, y) \in \alpha(t)$ and $(x, z) \in \beta(t)$.

We use the above lemma to present the composed data transducers for the query $\alpha \sim \beta$. We can treat the automaton $\mathcal{A}_{\alpha,\beta}$ as a binary relation

$$f_{\alpha,\beta} \subseteq \text{trees}(A \times P(sub(\alpha)) \times P(sub(\beta)) \times \{0, 1\}) \times \text{trees}(\{0, 1\}).$$

Using the order $0 < 1$ on the output alphabet $\{0, 1\}$, we get an aggregate data transducer

$$\widehat{f_{\alpha,\beta}} : \text{dtrees}(A \times P(sub(\alpha)) \times P(sub(\beta))) \to \text{trees}(\{0, 1\}).$$

Thanks to the property stated by Lemma G.3, this transducer computes the set of nodes that satisfy the query $\alpha \sim \beta$; assuming that its input is of the form

$$t \otimes loops_\alpha(t) \otimes loops_\beta(t) \otimes \mu.$$

The coordinates describing the loops can then be filled in by transducers from Lemma G.2. Note that the transducers from Lemma G.2 did not use the data; they were not data aggregate transducers, but aggregate transducers. They can easily be lifted to data aggregate transducers by ignoring the data.

Queries $\alpha \not\sim \beta$ can be solved in a similar way.