Efficient evaluation for a temporal logic on changing XML documents

Mikolaj Bojańczyk, Diego Figueira

Efficient Evaluation for a Temporal Logic on Changing XML Documents*

Mikołaj Bojańczyk
University of Warsaw
bojan@mimuw.edu.pl

Diego Figueira
University of Warsaw
University of Edinburgh
dfigueir@inf.ed.ac.uk

ABSTRACT

We consider a sequence $t_1, \ldots, t_k$ of XML documents that is produced by a sequence of local edit operations. To describe properties of such a sequence, we use a temporal logic. The logic can navigate both in time and in the document, e.g. a formula can say that every node with label a eventually gets a descendant with label b. For every fixed formula, we provide an evaluation algorithm that works in time $O(k \cdot \log(n))$, where k is the number of edit operations and n is the maximal size of document that is produced. In the algorithm, we represent formulas of the logic by a kind of automaton, which works on sequences of documents. The algorithm works on XML documents of bounded depth.

Categories and Subject Descriptors

F.4.1 [**Mathematical logic and formal languages**]: Temporal logic; H.2.3 [**Database management**]: Languages—*Query languages*

General Terms

Theory, Algorithms, Languages

Keywords

Incremental evaluation, XML, temporal logic

1. INTRODUCTION

In this paper, we model an XML document as a finite unranked tree over a finite alphabet. Suppose that an XML document is modified by a sequence of local update operations. Each update adds, removes or relabels a node of the document. We want to know if these updates preserve some correctness constraint. This kind of problem is called incremental evaluation. Suppose, for instance, that the correctness constraint is given by a DTD, or more generally a regular language of unranked trees. This problem is solved by an algorithm of Balmin, Papakonstantinou and Vianu [4]. The algorithm maintains a data structure, and every update on the tree requires an update to the data structure that is done in time

$$poly(Q) \cdot \log^2(n)$$

where Q is the state space of the automaton for the correctness constraint and n is the size of the current document. If the correctness constraint is given by an XPath query, the translation from the query into a tree automaton can result in a state space Q that is exponential in the size of the XPath query. The problem has been addressed by [6], which provides algorithms that work directly with XPath queries.

A limitation of incremental evaluation, as described above, is that the property talks only about the current version of the XML document. In this paper, we study properties that compare versions of the document in different times. Here are some examples:

1. Every *order* node eventually gets an *approved* child. That is, if at some time i a node x has tag name *order*, then at some time $j \geq i$ node x has a child with tag name *approved*.

2. Every node with one child gets a second child in at most two steps. That is, if at some time i, a node x has one child, then node x has at least two children in any time $j \geq i + 2$.

To express these examples, we need a logic or formalism that is evaluated not in a single document, but in a sequence of documents

$$t_1, \ldots, t_k.$$

We use the name *document evolution* for such a sequence. In this paper we define a logic for document evolutions, which captures the two examples given above. We then study the evaluation problem for formulas φ of the logic:

- Input. A document evolution $t_1, \ldots, t_k$.
- Output. Does φ hold in the document evolution?

*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.

As in incremental evaluation, the document evolution on the input is not an arbitrary sequence of documents, but a sequence where t_1 is an empty document and t_{i+1} is obtained from t_i by doing a local update that consists of adding, removing or relabeling a single document node. In this case, the size of document t_i is bounded by i. If the updates consist exclusively of adding nodes, then after time k the combined size of the documents $t_1, \ldots, t_k$ is quadratic in k.

We want an algorithm that works well for large XML documents, with millions of nodes or more. For documents of this size an algorithm that is linear in the combined size of the documents $t_1, \ldots, t_k$, and therefore quadratic in k, may be impractical. However, when the document evolution is a result of local updates, there is no reason to read the whole document t_i for every $i \in \{1, \ldots, k\}$, because this document is almost identical to t_{i-1}. The principal contribution of this paper is an algorithm for the evaluation problem that runs in time

$$O(k \cdot \log(n)),$$

where k is the duration of the document evolution and n is the maximal document size (of course $n \leq k$). We present the algorithm only for XML documents of bounded depth, although we believe it should work for documents of unbounded depth.

To the best of our knowledge, the problem of efficiently evaluating logics on document evolutions has not been studied before. In this paper, we try to optimize the data complexity of the problem, and we assume that the formula of the logic is fixed. Unfortunately, the query complexity of our algorithm is very bad: the constant hidden in the expression $O(k \cdot \log(n))$ is nonelementary in the size of the formula of the logic. That is why this paper is more like a proof of concept: even when the correctness property compares different versions of the document, it is still possible to have an algorithm with very good data complexity.

In this paper, we only talk about the tag names and tree structure of XML documents. That is, we model an XML document as an unranked tree over a finite and fixed alphabet (the tag names). Any additional data in the document, such as attribute values, is ignored. Logics and automata that talk about such additional data are a lively research topic, see the survey [13], and in the future we would like to extend our results to the richer logics. Note that for logics that depend on attribute values, such as XPath, the incremental evaluation problem seems to be difficult already for formulas that talk about only the current state of the document. In particular, we are not aware of any algorithm that would be significantly faster than simply reevaluating the XPath query on every new version of the document. By significantly faster we mean an algorithm that would require polylogarithmic time per update operation. Observe that in the presence of additional data, an evalution algorithm for XPath (without any time aspects, just evaluating a query in a given document) with linear data complexity has appeared only recently [8].

Related work. As already mentioned, [6] studies the problem of incrementally maintaining the result set of a XPath query on a changing XML document. Algorithms are provided for several fragments of XPath. The queries from [6] talk only about the current document, and cannot study the change of the structure of the document in time.

There have been efforts to extend the XPath query language to make it time-aware [9]. In this work we choose to work with a temporal logic to talk about different instants of the evolution of a document.

In a similar setup, there have been works on the incremental maintenance of the validity of regular languages [4, 5], where the validity of an XML Schema, a DTD or a specialized DTD is checked to be preserved throughout the document evolution.

Abiteboul, Herr and Van den Bussche [3] study the expressive power of some of the logics we work with.

The model of evolution of documents is also related to regular model checking [2]. In this field, the evolution of the document is dictated by a regular transducer, whereas here updates can occur anywhere and in any order, in an incremental way. In this context, [1] treats a logic similar to ours, LTL(MSO), where first and second order variables are interpreted over positions of the document, and tempral operatores are used to navigate the word in time.

2. TWO LOGICS

In this section, we define the logics that we use to describe document evolutions. As presented in the introduction, the update operations are: adding and removing nodes, as well as relabeling. In order to simplify the complicated evaluation algorithm, and the definitions of the logics, for the most part of the paper we only use relabeling, and do not add or remove nodes. In particular, all documents in a document evolution will have the same nodes. In Section 7, we come back to the implementation of adding and removing nodes.

Fix an input alphabet A. We deal with finite, unranked, sibling-ordered trees with nodes labeled by A. The *domain* of a tree is the tree without the labels. A *document evolution* is a sequence of trees $t_1, \ldots, t_k$. The number k is the *duration* of the document evolution, and the number n of nodes in the domain is the *document size* of the evolution. We work with the assumption that all documents have the same domain. This assumption is made for technical reasons to simplify the algorithm, but this assumption can be lifted, as explained in Section 7. Throughout the paper, we will preserve the convention that n refers to the document size and that k refers to the duration. We use x, y, z to refer to nodes in the domain and variables i, j to refer to moments in time.

In this section we define two logics to describe document evolutions. The first logic is a variant of first-order logic. The second logic is a variant of temporal logic. The evaluation algorithms of this paper are for the temporal logic, and the first-order logic is described mainly to provide some background.

First-order logic. A document evolution $t_1, \ldots, t_k$ with domain X can be treated as a relational structure. The elements of the structure are pairs

$$(x, i) \in X \times \{1, \ldots, k\}.$$

The relational structure is equipped with the following predicates. For each letter $a \in A$ in the input alphabet, we have a unary predicate a, which holds for an element (x, i) if node x has label a in the document t_i. There are three binary relations:

- The *descendant order* $\leq_{desc}$ which ignores the time

coordinate of its arguments and compares the nodes for descendant relationship.

- The *document order* $\leq_{doc}$ which ignores the time coordinate of its arguments and compares the nodes for document order relationship. This corresponds to the total order induced by the preorder visit of the tree.
- The *time order* $\leq_t$ which ignores the node coordinate of its arguments and compares the times.

Because the predicates above ignore one of the coordinates, they are not antisymmetric when $k \geq 2$ or $n \geq 2$. Using these predicates, we can write formulas of first-order logic to express properties of document evolutions. For instance, the following formula says that once a document node gets label a, none of its descendants ever have label b afterwards. (We use p, q for the variable names, because x, y, z are reserved for positions, and not space/time pairs.)

$$\forall p \forall q.\ \big(a(p) \wedge p \leq_t q \wedge p \leq_{desc} q\big) \Rightarrow \neg b(q) \qquad (1)$$

Two-dimensional temporal logic. We now define a logic with temporal syntax, with operators that travel in the time dimension and the space dimension. In particular, formulas of this logic will have no variables. We use the name *two-dimensional temporal logic* for this logic. This is the principal logic studied in the paper.

In the two-dimensional temporal logic, a formula is evaluated in a pair

$$(x, i) \in X \times \{1, \ldots, k\}$$

of a document evolution of document domain X and duration k. Instead of quantifiers, formulas use modal operators such G for Globally or U for Until. For instance the property from example (1) is stated as

$$\mathsf{G}_{desc}\mathsf{G}_t(a \Rightarrow \mathsf{G}_{desc}\mathsf{G}_t \neg b).$$

In the formula above, we begin in the root node in the first document. We then use the $\mathsf{G}_{desc}\mathsf{G}_t$ operators to say that $a \Rightarrow \mathsf{G}_{desc}\mathsf{G}_t \neg b$ occurs in all descendants in all later times, and likewise for the smaller formula.

We believe that when describing document evolutions, it is important to allow past operators in the space dimension (i.e. descendants and preceding nodes in document order). For example, consider an XPath unary query α, which selects a position x in a document if it has an ancestor with label a. In XPath syntax this query looks like this:

$$//\mathrm{a}//*$$

One might want to say that for every document node x and time i, if node x has label b in time i, then at all later times the node x satisfies the XPath query α

$$\mathsf{G}_{desc}\mathsf{G}_t(b \Rightarrow \mathsf{G}_t \alpha).$$

In the particular example of α, we can write $\mathsf{F}^{-1}_{desc} a$ to search for an ancestor of the current node in the current document.

We now provide the formal syntax and semantics of the temporal logic. As remarked above, a formula of the logic is evaluated in a pair (x, i) of a document evolution $t_1, \ldots, t_k$. We fix this document evolution in the definition below.

- For every letter of the alphabet, a is a formula. This formula is true in (x, i) if node x has label a in document t_i.
- Boolean connectives are allowed, with the usual semantics.
- If φ, ψ are a formulas, then $\varphi \mathsf{U}_{desc} \psi$ is a formula, which is true in (x, i) if there exists a node $y >_{desc} x$ such that φ is true in all the pairs (z, i) with $x <_{desc} z <_{desc} y$ and ψ is true in the pair (y, i). Observe that we use the *strict until* operator, which does not use the current position. Often, one uses the *non-strict until*, which is defined by

  $$\psi \vee (\varphi \wedge (\varphi \mathsf{U}_{desc} \psi)).$$

  The strict until allows one to define all other operators as derived constructs:

  $$\mathsf{X}_{desc}\varphi \stackrel{def}{=} \bot \mathsf{U}_{desc}\varphi \qquad \mathsf{F}_{desc}\varphi \stackrel{def}{=} (\top \mathsf{U}_{desc}\varphi) \vee \varphi$$
  $$\mathsf{G}_{desc}\varphi \stackrel{def}{=} \neg \mathsf{F}_{desc} \neg \varphi$$

- If φ, ψ are formulas, then $\varphi \mathsf{U}^{-1}_{desc} \psi$ is a formula, which is the inverse of the U_{desc} operator. This operator is often called *since*. As in the previous item, we can define all sorts of derived operators.
- Let $\leq_{sib}$ be sibling order, where $x \leq_{sib} y$ if x and y are siblings, and x is to the left of y. If φ, ψ are formulas, then $\varphi \mathsf{U}_{sib} \psi$ and $\varphi \mathsf{U}^{-1}_{sib} \psi$ are formulas, defined in the same way as before, for the sibling order $\leq_{sib}$.
- Finally, we allow an until operator U_t that works in the time dimension, i.e. $\varphi \mathsf{U}_t \psi$ holds in (x, i) if there is a time moment $j > i$ such that φ is true in the pairs

  $$(x, i+1), \ldots, (x, j-1)$$

  and ψ is true in the pair (x, j).

Observe that we do not allow a since operator in the time dimension. Such a logic would make perfect sense, but we do not know how to extend our evaluation algorithm to cover a since operator in the time dimension. In fact, it seems that the two dimensional temporal logic with since and until is strictly more expressive than with only until, the witness property being "there is a time moment i such that $t_1 = t_i$".

As far as the space dimension is concerned, our logic behaves like two-way CTL. Two-way CTL is first-order complete, see [12]. We could even allow more powerful operations in the logic. This is because our evaluation algorithm represents the formula as an alternating automaton. Then, our algorithm works even if we make tests for any regular properties at fixed time moments.

3. EVALUATION

The subject of this paper is evaluation of logical formulas on document evolutions. We are not interested in the satisfiability problem, because it is undecidable for all the logics described here, by using a document evolution to describe a run of a Turing machine.[1]

The *evaluation problem* for a logical formula φ is stated as follows.

- Input. A document evolution $\bar{t}$.

[1]In fact, it has been shown that the two-dimensional temporal logic is undecidable even if we only allow future modalities in both dimensions [10].

- Output. Does the formula hold in $\bar{t}$?

When measuring the complexity of the problem, we use two parameters of the document evolution $\bar{t}$: the document size n and the duration k.

First-order logic. In this paper, we are mainly interested in the logics that are temporal in at least one of the two dimensions of space and time. Why not first-order logic where the variables range over space/time pairs? A document evolution of document size n and duration $n = k$ can encode any graph with n vertices, since it is essentially an $n \times k$ matrix. Therefore, evaluating first-order logic on document evolutions is the same as evaluating first-order logic on graphs, or arbitrary relational structures for that matter. The latter is a fascinating and widely studied topic, but it is not the topic of this paper.

An optimal algorithm for the two-dimensional temporal logic. There is a simple optimal algorithm for the two-dimensional logic.

THEOREM 1. *The evaluation problem for two-dimensional temporal logic can be solved in time $O(k \cdot n)$.*

PROOF. Suppose that X is the domain of the document evolution. By induction on the size of a formula φ, we show that the set of pairs

$$(x, i) \in X \times \{1, \ldots, k\}$$

in a document evolution $t_1, \ldots, t_k$ that satisfies φ can be computed in time

$$O(|\varphi| \cdot n \cdot k).$$

Suppose that we want to compute the pairs where $\varphi \mathsf{U}_t \psi$ is satisfied. For document node $x \in X$, we scan the document evolution from time k to time 1. Using the precomputed information on φ and ψ we can compute the bigger formula. A similar argument works for the other operators. For the space operators, one needs to do a bottom-up or top-down pass through the trees. □

Observe that the algorithm above would also work if we allowed a since operator in the time dimension. The since operator will be a problem in our later algorithm.

If the document evolutions are arbitrary sequences of documents, with no restriction on small differences between consecutive documents, then this naive algorithm is optimal. All $n \cdot k$ positions of the document evolution must be read for some queries, e.g. the query "all nodes in all times have label a."

Incremental evaluation. The *incremental evaluation problem* is a variant of the evaluation problem, where we assume that the document evolution is the result of applying a sequence of local updates to an initially empty document. Here empty means that all nodes have the same blank label. Formally speaking, the input of the incremental evaluation problem consists of: a tree domain X, a duration $k \in \mathbb{N}$, an initial letter $a \in A$ and a sequence of pairs

$$(a_1, x_1), \ldots, (a_{k-1}, x_{k-1}) \in A \times X.$$

Given this input, we define a document evolution $t_1, \ldots, t_k$ as follows. The first document t_1 has all nodes labeled by a. Once t_i has been defined, t_{i+1} is defined as t_i with position x_i changing label to a_i. A document evolution obtained from such a sequence of relabeling operations is called an *incremental document evolution.*

In the (general) evaluation problem, reading the input requires time $n \cdot k$. In the incremental evaluation problem, reading the input requires time k, so the simple lower bound of $O(n \cdot k)$ does not hold any more. The principal contribution of this paper is that the incremental evaluation problem can be solved more efficiently, as stated in the following theorem.

THEOREM 2 (MAIN THEOREM). *Fix a formula φ of the two-dimensional temporal logic. The incremental evaluation problem, on a document evolution of document size n and duration k can be solved in time $O(k \cdot \log(n))$.*

In the theorem above, the constant in the O notation depends on the formula φ. As a function of φ, the constant grows faster than any tower of exponentials. That is why this paper is intended more as a proof of concept, the concept being that the incremental evaluation problem can be done in time smaller than the obvious $O(n \cdot k)$.

4. AN AUTOMATON MODEL

In this section we define an automaton model that accepts or rejects document evolutions. This automaton can capture the two-dimensional temporal logic, and even some extensions. Our evaluation algorithm works with the automaton.

Regular queries. A regular tree language is like a boolean query for trees: it says yes or no to each tree. In this paper we also use unary queries, binary queries and so on. An m-ary query over trees, with input alphabet A, is a function f which maps every tree over alphabet A to a set of m-tuples of nodes in the tree. An example of a binary query is one that maps a tree to the set of all pairs of nodes (x, y) such that there is an even number of a's on the shortest path from x to y.

We will be using *regular queries*, which are queries that can be defined by formulas of MSO logic with free individual variables, see e.g. [11]. Regular queries can also be described in other ways, different than MSO formulas, e.g. by automata or monoids. The way we describe regular queries is not very important for our evaluation algorithm (actually, the algorithm uses monoids as its internal representation). In particular, the bad query complexity of our algorithm is not due to the use of MSO in the automaton, the algorithm would have nonelementary complexity even with regular queries represented in some less succint way, e.g. by monoids.

Document update processing automata. We now introduce the automaton model, which we call a *document update processing automaton.* Such an automaton, call it $\mathcal{A}$, is given by the following ingredients.

- An input alphabet A.
- A set of states Q, an initial state q_0 and a set of accepting states $F \subseteq Q$. The states are partitioned into $Q_\exists$ and $Q_\forall$. (This is a kind of alternating automaton.)

- A finite set Δ of transitions of the form (q, φ, p) where $q, p \in Q$ and φ is a binary regular query over input alphabet A.

An input for the automaton is a document evolution

$$\bar{t} = t_1, \ldots, t_k$$

with domain X, where the trees are over alphabet A. The automaton accepts the document evolution if player $\exists$ wins the following perfect-information, finite duration, two-player game, call it $G_{\bar{t}}^{\mathcal{A}}$. Positions of the game are triples of the form (q, x, i) where $q \in Q$ is a state of the automaton, $x \in X$ is a node, and $i \in \{0, \ldots, k\}$ is a moment in time. The interpretation of i is that we are about to read the tree t_{i+1}. The initial position is $(q_0, root, 0)$, i.e. the game starts in the initial state, in the root, and before reading the first tree in the document evolution. When in a game position (q, x, i) with $i < k$, the player who owns state q chooses a new game position $(p, y, i + 1)$ such that

$$(q, \varphi, p)$$

is a transition of the automaton and φ selects the pair (x, y) in the document t_{i+1}. When a game position of the form (q, x, k) is reached, the game is finished, and player $\exists$ wins if q is accepting, otherwise $\forall$ wins. A position in the game is called winning for $\exists$ if he can win the game starting in that position.

From two-dimensional logic to dupa automata

PROPOSITION 3. *For every formula φ of the two-dimensional temporal logic there is a document update processing automaton that accepts the same document evolutions.*

PROOF. States of the automaton are subformulas, so the size of the automaton is quadratic in the formula. The translation is as usual when going from temporal logic to an alternating automaton. □

Ordered automata. We say a transition (p, φ, q) in a document update processing automaton is a *local transition* if the formula φ only selects pairs of the form (x, x) in every tree (in wich case, φ is essentially a unary query). An automaton is called *ordered* if there is a ranking function $\Omega : Q \to \mathbb{N}$ on its states such that performing any transition preserves or decreases the rank, and performing a non-local transition decreases the rank. It is possible that several states have the same rank, in which case the states can be connected only by local transitions. Observe that the automaton produced in Proposition 3 is an ordered automaton, the ranking function refers to the size of the subformula corresponding to the current state.

The principal technical result of this paper is that ordered document update processing automata can be evaluated efficiently.

THEOREM 4. *Fix an ordered document update processing automaton. Whether or not the automaton accepts a document evolution of duration k and tree length n can be tested in time $O(k \cdot \log(n))$.*

Thanks to Proposition 3, the theorem above also yields an evaluation algorithm for the two-dimensional temporal logic, of same complexity, as stated in Theorem 2.

The evaluation algorithm is described in Sections 5 and 6. Due to the complicated structure of the algorithm, we describe it for the case of words (which can be seen as the special case of trees with only one path). In Section 7 we comment on how to extend the algorithm for trees.

5. LOCAL ORDERED AUTOMATA

In this section we show how to evaluate a *local automaton*, which is a document update processing automaton where only local transitions are allowed. In the next section, the ideas on local automata are extended to the general case of ordered automata with non-local transitions.

As mentioned before, we describe the algorithm for words and not trees. In particular, the tree domain is now given by just specifying the *word length* n, in which case the domain is $\{1, \ldots, n\}$. We will prove the following theorem in this section.

PROPOSITION 5. *Fix a local automaton. The automaton can be evaluated on incremental document evolutions of word length n and duration k in time $O(k \cdot \log(n))$.*

Actually, we prove a slightly stronger result. We will create a data structure such that for every node $x \in \{1, \ldots, n\}$, one can tell in time $O(\log(n))$ if the automaton accepts the word if it begins in the initial state in node x. So, in a sense, we evaluate the local automaton for all nodes in the word simultaneously.

We fix for the rest of this section a local automaton. We also fix an input to the automaton, i.e. an incremental document evolution

$$w_1, \ldots, w_k \in A^n.$$

Our goal is to determine if the automaton accepts this word. In the algorithm, we will refer to the differences between successive words w_i and w_{i+1}, so we assume that the incremental document evolution is given by a sequence of relabeling operations.

Monoids for queries. In this section, we assume that all of the transitions are local, i.e. every transition is of the form

$$(p, \varphi, q)$$

where φ only selects tuples of the form (x, x) in every word. Therefore, we will simply treat φ as a unary query. Instead of MSO logic, we use monoids to describe regular queries. We describe the monoid approach to unary queries below.

Suppose that

$$\alpha : A^* \to M$$

is a monoid homomorphism. The unary α-type of a node x in a word $a_1 \cdots a_n$ is the triple

$$(\alpha(a_1 \cdots a_{x-1}), a_x, \alpha(a_{x+1} \cdots a_n)) \in M \times A \times M.$$

We say the homomorphism α recognizes a unary query $\varphi(x)$ if there is a set of triples

$$H \subseteq M \times A \times M$$

such that a node x is selected by the query φ in a word w if and only if the unary α-type of x in w belongs to H. For every unary MSO query there exists a monoid homomorphism that recognizes it. The monoid M might be nonelementary in the size of the query. (This translation of MSO into

monoids is not the only reason for bad query complexity, our algorithm has nonelementary query complexity even when the monoids M are small.)

We fix a single monoid homomorphism $\alpha : A^* \to M$ that recognizes all the queries that appear in the transitions of the automaton. This can be done by using the direct product of all the monoids for all the queries. When referring to unary types, we omit the name of α, because it is the only homomorphism we use.

Zones. We begin by introducing some notation. We write x, y, z for positions in the words, i.e. for elements of $\{1, \ldots, n\}$. A zone is defined to be any connected set of positions, i.e. a zone is a set $\{x, x+1, \ldots, y\}$ for $1 \leq x \leq y \leq n$. We write X, Y for zones, and we also allow an empty zone. If X is a zone and w is a word with n positions, then $w|X$ is the word obtained from w by only keeping positions from X. The *type* of a zone X in a word w is defined to be $\alpha(w|X)$. Zones $X_1, \ldots, X_m$ are called *consecutive* if the last node in zone X_1 is the predecessor of the first node in X_2, and so on until X_{m-1} and X_m.

The *environment* of a zone X inside a zone $Y \supseteq X$ is the difference $Y - X$. Such an environment is the union of two disjoint zones, the one to the left of X and the one to the right of X. The type of such an environment is the pair of types of these two disjoint zones. We write

$$\mathit{Externals} \stackrel{\mathit{def}}{=} M^2$$

for the set of possible types of such environments. The *external type* of a zone X is the type of its environment with respect to the set $\{1, \ldots, n\}$ of all word positions. We write τ, σ for external types. For $i = 1, 2$, we write $\tau(i)$ for the i-th coordinate of an external type τ. We can also insert one external type inside another

$$\mathit{in}_\sigma(\tau) \stackrel{\mathit{def}}{=} (\tau(1) \cdot \sigma(1), \sigma(2) \cdot \tau(2)) \in \mathit{Externals}.$$

The idea is that if $X \subseteq Y \subseteq Z$ and σ is the type of the environment $Y - X$, and τ is the type of the environment $Z - Y$, then $\mathit{in}_\sigma(\tau)$ is the type of the environment $Z - X$.

Histories. We write i, j for times, which are elements of $\{1, \ldots, k\}$. A time interval is defined like a zone, but for times. For a time $i \in \{1, \ldots, k\}$ we write

$$\mathit{ext}(X, i) \in \mathit{Externals}$$

for the external type of a zone X in the word w_i. Consider a zone X and a time interval $I = \{i, \ldots, j\}$. We define the *external history* of the interval X in time interval I, denoted $\mathit{history}(X, I)$, to be the sequence

$$\mathit{ext}(X, i), \ldots, \mathit{ext}(X, j)$$

of external types of X in the times from the interval I. Since we only talk about external histories, we simply write history from now on. We only use the value $\mathit{history}(X, I)$ when the zone X does not change labels in the time interval I. To underline this requirement, we assume that $\mathit{history}(X, I)$ is undefined otherwise. We write *Histories* for the set of possible histories, this set includes the undefined history $\perp$.

5.1 The history toolkit

We define some operations on histories. These operations will be used by the algorithm as a black box.

Monoid operations on histories. There is an empty history:

$$\mathit{empty} \in \mathit{Histories} \tag{2}$$

The simplest operation we can do on histories is concatenate them. When histories are represented as sequences of external types, this operation is simply sequence concatenation. Later, we represent histories in a more concise way, and the operation becomes less trivial, so we give it a name:

$$\mathit{concat} : \mathit{Histories} \times \mathit{Histories} \to \mathit{Histories}. \tag{3}$$

Applying a history to a node. For a node x and a time i, we define $P_{xi} \subseteq Q$ to be the set of states q such that the game position (q, x, i) is winning in the game $G_w^{\mathcal{A}}$. Recall that (q, x, i) describes the situation between words w_i and w_{i+1}. The principal goal of our algorithm is to design an algorithm and data structure, so that after the algorithm terminates, each set P_{x0} can be computed in time $O(\log(n))$ by using the data structure.

Generally speaking, our algorithm will be processing the words in the time evolution in reverse order, beginning with w_k and ending with w_1. All the information gathered by the algorithm will be propagated from later times to earlier times. We say that a node x does not change label in time i if the label of x is the same in words w_i and w_{i+1}.

LEMMA 6. *Suppose that in the time interval $\{i, \ldots, j\}$, the label of a position x is constantly $a \in A$. Then the set $P_{x(i-1)}$ is uniquely determined by: the label a, the set P_{xj} and $\mathit{history}(\{x\}, \{i, \ldots, j\})$.*

PROOF. Consider first a history of length one, i.e. $j = i$. Let $\mathit{history}(\{x\}, \{i\})$ be (σ, τ). Then the α-type of x in w_i is (σ, a, τ). The set $P_{x(i-1)}$ contains the states q such that either

- q is owned by player $\forall$ and for every transition (q, φ, p), if (σ, a, τ) satisfies φ then $p \in P_{xi}$, or
- q is owned by player $\exists$ and there is a transition (q, φ, p) such that (σ, a, τ) satisfies φ, and $p \in P_{xi}$.

These conditions are necessary and sufficient for $(q, x, i-1)$ being an accepting position in the automaton's game.

For longer histories, the lemma is obtained by iterating the statement for histories of length one. □

We denote the function that realizes the dependency from the above lemma by

$$\mathit{apply} : \mathit{Histories} \to A \times P(Q) \to P(Q). \tag{4}$$

That is, this function is defined so that, under the assumptions of the lemma, we have

$$P_{x(i-1)} = \mathit{apply}(\mathit{history}(\{x\}, \{i, \ldots, j\}))(a, P_{xj}).$$

Inheriting the history of a parent. Let Y be a zone that contains X. Suppose that in the time interval I, the input labels of $Y - X$ did not change, and therefore constantly had the same type $\sigma \in \mathit{Externals}$. In this case,

$$\mathit{history}(X, I) = \mathit{in}_\sigma(\mathit{history}(Y, I))$$

where the operation

$$\mathit{in}_\sigma : \mathit{Histories} \to \mathit{Histories} \tag{5}$$

is defined by $\mathit{in}_\sigma(\tau_1, \ldots, \tau_m) = \mathit{in}_\sigma(\tau_1), \ldots, \mathit{in}_\sigma(\tau_m)$.

5.2 The algorithm

We now present our algorithm. It uses a fairly natural divide-and-conquer approach (not for nothing there is $\log(n)$). The crux is an efficient implementation of operations on histories, which is described in later sections.

We define *logarithmic* zones as follows: the set of all positions of $\{1,\ldots,n\}$ is a logarithmic zone; and if X is a logarithmic zone of w and $x \in X$ is in the middle of X then $X \cap \{1,\ldots,x\}$ and $X \setminus \{1,\ldots,x\}$ are both logarithmic zones. The properties of these zones are: they are either disjoint or one is included in the other, each position is included in a logarithmic number of them, and there are linearly many of them. We write $\mathit{parent}(X)$ for the smallest logarithmic zone $Y \supsetneq X$ and *Logs* for the set of all logarithmic zones.

The history tree. We first define the data structure used by our algorithm. An instance D of the data structure consists of four labelings:

$$\begin{aligned} D.\mathit{lastupdate} &: \mathit{Logs} \to \{0,\ldots,k\} \\ D.\mathit{lastlabel} &: \{1,\ldots,n\} \to A \times P(Q) \\ D.\mathit{history} &: \mathit{Logs} \to \mathit{Histories} \\ D.\mathit{monoid} &: \mathit{Logs} \to M \end{aligned}$$

Because the logarithmic zones have a tree structure, and because the component $D.\mathit{history}$ is the most important one, we use the name *history tree* for such an instance D.

We will first compute a history tree for time k, and then will compute it for times $k-1, k-2, \ldots, 0$. A history tree D is called *correct* at time moment $i \in \{0,\ldots,k\}$ if the following conditions hold. (The idea is that D describes the situation between words w_i and w_{i+1}.)

1. For every logarithmic zone X and its parent Y,

   $$D.\mathit{lastupdate}(X) \geq D.\mathit{lastupdate}(Y).$$

   In other words, zones intervals are more up to date than smaller zones. (Because the algorithm starts in k and progresses towards 0, information about a smaller time is more up to date.) We say a zone X is *up to date* (at moment i) if

   $$D.\mathit{lastupdate}(X) = i.$$

   We require that the root logarithmic zone $\{1,\ldots,n\}$ is up to date.

2. Labels cannot change between the current time i and the last update. That is, for every logarithmic zone, all of its input labels have stayed the same in the time interval

   $$\{i,\ldots,D.\mathit{lastupdate}(X)\}$$

3. For every logarithmic interval X, the value $D.\mathit{monoid}(X)$ stores the monoid element corresponding to X at the time of its last update:

   $$D.\mathit{monoid}(X) = \alpha(w_{D.\mathit{lastupdate}(X)}|X)$$

   In fact we have that $D.\mathit{monoid}(X) = \alpha(w_j|X)$ for every $i \leq j \leq D.\mathit{lastupdate}(X)$ since $w_j|X$ remains unchanged.

4. Suppose that X is a logarithmic interval, and Y is its parent. Then $D.\mathit{history}(X)$ stores the value

   $$\mathit{history}(X, \{D.\mathit{lastupdate}(Y),\ldots,D.\mathit{lastupdate}(X)-1\}).$$

   In the case when $D.\mathit{lastupdate}(Y) = D.\mathit{lastupdate}(X)$ the above history is empty.

5. Suppose that x is a node and let

   $$j = D.\mathit{lastupdate}(\{x\}) \quad \text{and} \quad (a,P) = D.\mathit{lastlabel}(x).$$

   Then the label of x in word w_j is a and $P_{xj} = P$. Together with condition 2 this means that the input label of x is a during the interval $\{i,\ldots,j\}$.

The algorithm. The algorithm begins with a history tree D_k that is correct and up to date for time moment $i = k$. Then for each time moment $i \in \{k-1,\ldots,0\}$, the algorithm calculates a history tree D_i that is correct for time moment i. The algorithm that maintains this invariant will manipulate histories. In our complexity analysis, we will use the history operations

$$\mathit{empty} \quad \mathit{concat} \quad \mathit{apply} \quad \mathit{in}_\sigma$$

described in Section 5.1 as black boxes, with unit cost in the algorithm. Later, we will show how the operations can be implemented in constant time.

LEMMA 7. *D_k can be computed in time $O(n)$.*

It is safe to assume $k > n$, because otherwise our algorithm can ignore the positions where no labels are updated. Therefore, the $O(n)$ precomputation cost of D_k will be amortized by the algorithm's $O(k \cdot \log(n))$ complexity.

LEMMA 8. *Let D_i be a history tree that is correct for time i. Let x be a position. We can compute a history tree D'_i that is also correct for time i and which is up to date at the zone $\{x\}$. This computation can be done in time $O(\log(n))$, assuming the history operations have unit cost.*

PROOF. Consider the logarithmic zones that contain x:

$$\{x\} = X_0 \subsetneq X_1 \subsetneq \cdots \subsetneq X_m.$$

We will compute history trees

$$D^{(m)}, D^{(m-1)}, \ldots, D^{(0)} = D'$$

which are all correct for time i, and such that

$$D^{(r)}.\mathit{lastupdate}(X_r) = i \qquad \text{for } r \in \{0,\ldots,m\}.$$

Furthermore, a constant number of operations on the history tree $D^{(r)}$ are sufficient to compute $D^{(r-1)}$. The statement of the lemma then follows, because m is logarithmic in n.

For the induction base, we simply use $D^{(m)} = D_i$, because item 1 of the correctness conditions says that the root logarithmic zone is always up to date.

The rest of the proof is devoted to the induction step. Suppose we computed $D^{(r+1)}$ and we want to compute $D^{(r)}$. We simply make the zone X_r up to date in $D^{(r)}$ by setting

$$D^{(r)}.\mathit{lastupdate}(X_r) := i$$

and keeping the value of *lastupdate* unchanged for other logarithmic zones. What else do we need to change in $D^{(r)}$ to make this history tree correct for time i? We examine the conditions of correctness item by item.

1. Item 1 holds because the zones $X_{r+1},\ldots,X_m$ are already up to date in $D^{(r+1)}$, and the root zone is unchanged.

2. We have decreased the value of $D^{(r)}.lastupdate(X_r)$, which makes item 2 even more true.

3. Let use j for the previously used time in X_r, i.e.

$$j = D^{(r+1)}.lastupdate(X_r).$$

We know from item 2 applied to $D^{(r+1)}$ that no positions have been changed in the zone X_r between the current time i and time j, so we can keep

$$D^{(r)}.monoid(X_r) := D^{(r+1)}.monoid(X_r).$$

4. For this item, we change the histories in at most three zones. For the history of X_r, we use the empty history, because X_r is now up to date:

$$D^{(r)}.history(X_r) := empty.$$

Suppose that $r > 0$. In this case, X_r has two children zones, call them Y_0 and Y_1 from left to right. (One of the children is X_{r-1}, but this plays no role in our argument.) The condition in item 4 could now be invalid for the two zones Y_0, Y_1, because we have changed the update time of their parent X_r. For $l \in \{0, 1\}$ let j_l be

$$D^{(r)}.lastupdate(Y_l) = D^{(r+1)}.lastupdate(Y_l).$$

To make the history tree $D^{(r)}$ correct, we should have

$$D^{(r)}.history(Y_l) = history(Y_l, \{i, \ldots, j_l - 1\}),$$

so we need to compute the history on the right of the above equation. What we have from the history tree $D^{(r+1)}$ is two histories:

$$history(Y_l, \{j, \ldots, j_l - 1\}) \quad history(X_r, \{i, \ldots, j - 1\}).$$

However, we know that from time i to j the external type of Y_l inside X_r was the same, call this external type τ_l. The value of this external type is taken from the *monoid* part of the history tree:

$$\tau_0 = (1, D^{(r)}.monoid(Y_1)) \quad \tau_1 = (D^{(r)}.monoid(Y_0), 1).$$

By applying the in_{τ_l} operation, we can compute the history of Y_l between times j and i as follows:

$$history(Y_l, \{i, \ldots, j - 1\}) = in_{\tau_l}(history(X_r, \{i, \ldots, j - 1\})).$$

Then, the value of $D^{(r)}.history(Y_l)$ should be set to

$$concat(D^{(r+1)}.history(Y_l), in_{\tau_l}(D^{(r+1)}.history(X_r))).$$

5. Let y be any position. If we have the equality

$$D^{(r)}.lastupdate(\{y\}) = D^{(r+1)}.lastupdate(\{y\}),$$

then no change in $D^{(r)}.lastlabel(y)$ is required preserve item 5, because of correctness of $D^{(r+1)}$. The only case when the equality above can fail is when $y = x$ and $r = 0$. We study this case below.

Let $j = D^{(1)}.lastupdate(\{x\})$. By correctness of $D^{(1)}$ we know that

$$D^{(1)}.lastlabel(x) = (a, P_{xj})$$

where a is the label of x in time j. To make item 5 true, we need to have

$$D^{(0)}.lastlabel(x) = (a, P_{xi}),$$

which requires knowing the set of states P_{xi}. By the previous item, we have the history

$$D^{(0)}.history(\{x\}) = history(\{x\}, \{i, \ldots, j - 1\}).$$

Therefore, we can apply the history above to the pair (a, P_{xj}) and get the set P_{xi} from

$$apply(D^{(0)}.history(\{x\}))(D^{(1)}.lastlabel(x)). \quad \square$$

A similar proof yields the following lemma.

LEMMA 9. *Let x be the position where w_{i+1} differs from w_i. Let D'_{i+1} be a history tree that is correct for time $i + 1$ and up to date at x. We can compute a data structure D_i that is correct for time w_i. This computation can be done in time $O(\log(n))$, assuming the history operations have unit cost.*

By applying the two lemmas in alternation, we can compute successively data structures

$$D_k, D'_k, D_{k-1}, D'_{k-1}, \ldots, D'_1, D_0.$$

This takes $O(k \cdot \log(n))$ time, and $O(k \cdot \log(n))$ history operations. If we implement histories in a naive way, by writing down a history as a list of pairs, each of the operations will take time linear in the length of the list. This will be a problem for the algorithm, which would run in time $O(k \cdot n \cdot \log(n))$.

The essence of the algorithm is to implement histories so that the operations can be done in constant time. This will be done in Section 5.3.

5.3 A constant time implementation of histories

History congruence. Recall the operations on histories that were used as black box operations in the algorithm from the previous section.

$$\begin{aligned} empty &\in Histories \\ concat &: Histories \times Histories \to Histories. \\ apply &: Histories \to A \times P(Q) \to P(Q). \\ in_\sigma &: Histories \to Histories \end{aligned}$$

A *history congruence* is an equivalence relation $\sim$ on *Histories* that is a congruence with respect to all the above operations. Formally speaking, a history congruence must satisfy the following conditions whenever $h \sim h'$

$$\begin{aligned} concat(g, h) &\sim concat(g, h') && \text{for all } g \in Histories \\ concat(h, g) &\sim concat(h', g) && \text{for all } g \in Histories \\ apply(h) &= apply(h') \\ in_\sigma(h) &\sim in_\sigma(h') && \text{for all } \sigma \in Externals \end{aligned}$$

If $\sim$ is a history congruence, then the history operations can be applied to equivalence classes of $\sim$. The family of equivalence classes together with the history operations on equivalence classes is denoted by $Histories/\sim$.

LEMMA 10. *Suppose that $\sim$ is a history congruence. Then Histories can be replaced by Histories$/_\sim$ in the algorithm, and the computed values for $D_i.lastlabel(\{x\})$ will be the same.*

PROOF. The algorithm accesses the histories only using the operations. □

The constant time implementation of histories is based on the following lemma.

LEMMA 11. *There is a history congruence where the number of equivalence classes is finite and depends only on the automaton, and not its input (not on n or k).*

The idea of the proof is that two histories are considered equivalent if they induce the same mapping:

$$\tau \in \mathit{Externals} \quad \mapsto \quad \mathit{apply}(\mathit{in}_\tau(h)) : A \times P(Q) \to P(Q).$$

6. NON-LOCAL ORDERED AUTOMATA

In this section we generalize the algorithm from the previous section to deal with non-local transitions in an ordered automaton.

The difficulty in this section is a new notion of history. A history is still used to store information about environments of zones, but this information is now more complex.

6.1 Preliminaries

Fix an ordered document update processing automaton. Assume that the maximum rank given by the function Ω of the automaton is t. We define the sets of states

$$Q^1 \subseteq Q^2 \subseteq \cdots \subseteq Q^t \qquad \text{where } Q^i \stackrel{def}{=} \{q \in Q \mid \Omega(q) \leq i\}.$$

Binary types. The automaton now includes non-local transitions, which are of the form (q, φ, q') where φ is a binary regular query that may select nodes (x, y) of the word with $x \neq y$. As in the previous section, we use monoids to recognize queries. For unary queries, we use the notion of type of a node x as in the previous section. We write $\mathit{Unarytypes}_\alpha$ to denote the set of unary α-types, we omit the index α when the homomorphism α is clear from the context.

We extend the notion of type to pairs of nodes. Suppose

$$\alpha : A^* \to M$$

is a monoid homomorphism. We define the binary α-type of two nodes x and y in a word w as the element of

$$\mathit{Binarytypes}_\alpha \stackrel{def}{=} M \times A \times M \times A \times M \times \{<, >, =\}$$

containing the three types of the three infixes of the word $\{1, \ldots, i-1\}$, $\{i+1, \ldots, j-1\}$, $\{j+1, \ldots, n\}$, where $i = \min(x, y)$, $j = \max(x, y)$, and the labels of positions i and j. The last component says whether $x = y$, $x < y$ or $x > y$. We write $\mathit{type}_\alpha(x, y, w) \in \mathit{Binarytypes}_\alpha$ to denote the binary type of x and y on w. For every regular binary query φ there is a homomorphism α and a set $F \subseteq \mathit{Binarytypes}_\alpha$ that recognizes φ, that is, that (x, y) is selected on w by φ if and only if $\mathit{type}_\alpha(x, y, w) \in F$.

We fix a homomorphism α that recognizes all the binary queries in the transitions of the automaton. We omit the index α from the notation.

Node profiles. As in Section 5, the goal of the algorithm is to compute the sets P_{xi} for positions x and times i.

Suppose that a node x does not change its label between in times $I = \{i, \ldots, j\}$. Due to non-local transitions, it is not enough to know the sequence of external types of x in order to update it from P_{xj} to $P_{x(i-1)}$. There may be a state q in $P_{x(i-1)}$ that is triggered by some $q' \in P_{yi}$ via a non-local transition from some other node y. Due to these issues, we store more information in our data structures. We describe this information below, starting with the simplest brick: profiles of nodes.

Consider a node $x \in \{1, \ldots, n\}$, a time $i \in \{1, \ldots, k\}$ and a rank $l \in \{1, \ldots, t\}$. By induction on the rank l, we define *the rank l profile of node x in time i* to be the following information, which is denoted by $\pi_x^{l,i}$:

- The type of node x in the word w_i;
- the set of states in $P_{xi} \cap Q^l$;
- for every binary type $\tau \in \mathit{Binarytypes}$, the set of rank $l-1$ profiles of nodes y in time i such that the binary type of x and y in w_i is τ.

When the rank l is not specified, it assumed to be the maximal rank $l = t$. Observe that the information in π_x^i also describes other nodes than x, due to the third item of the definition. Since the definition is recursive, a rank l profile can be seen a tree of depth t where edges correspond to *Binarytypes* elements, and nodes with a subtree of depth l are described by the accepting states from Q^l. The data type where rank l profiles live is denoted by $\mathit{Nodeprofiles}_l$, and it is described below:

$$\mathit{Nodeprofiles} \stackrel{def}{=} \mathit{Nodeprofiles}_t$$

$$\mathit{Nodeprofiles}_1 \stackrel{def}{=} \mathit{Unarytypes} \times P(Q^1)$$

$$\mathit{Nodeprofiles}_{l+1} \stackrel{def}{=} \mathit{Unarytypes} \times P(Q^{l+1}) \times \mathit{Neighbours}_l$$

$$\mathit{Neighbours}_l \stackrel{def}{=} \mathit{Binarytypes} \to P(\mathit{Nodeprofiles}_l)$$

Note that from $\pi_x^{l,i}$ we can deduce $\pi_x^{l',i}$ for all $l' < l$, because $Q^{l'} \subseteq Q^l$. The size of the data type $\mathit{Nodeprofiles}_{l+1}$ is exponential in the size of $\mathit{Nodeprofiles}_l$, hence the nonelementary query complexity of our algorithm.

Zone profiles. We define the profile of a node x relative to a zone $X \ni x$ by taking $w_i|X$ as the whole word. All the elements from *Unarytypes* and *Binarytypes* are relative, and the neighbours are limited to the zone X. This profile is denoted by $\pi_{x \in X}^i$.

The profile of a zone X in time moment i is just the set of node profiles (relative to X) of nodes in X. We use the symbol Π to denote zone profiles. Following the same notation introduced before, $\Pi_X^{l,i}$ is the zone profile of the zone X at time moment i containing the information of the states from Q^l.

$$\Pi_X^{l,i} \stackrel{def}{=} \{\pi_{x \in X}^{l,i} : x \in X\} \in \mathit{Zoneprofiles}_l \stackrel{def}{=} P(\mathit{Nodeprofiles}_l)$$

$$\Pi_X^i \stackrel{def}{=} \Pi_X^{t,i} \in \mathit{Zoneprofiles} \stackrel{def}{=} \mathit{Zoneprofiles}_t$$

The following lemma shows that zone profiles are compositional with respect to union of consecutive zones.

LEMMA 12. *There is an associative concatenation operation on profiles* $(\Pi_1, \Pi_2) \mapsto \Pi_1 \cdot \Pi_2$ *such that for two consecutive zones* X_1 *and* X_2*, we have*

$$\Pi_{X_1 \cup X_2} = \Pi_{X_1} \cdot \Pi_{X_2}.$$

One step computation.

LEMMA 13. *Suppose that the nodes of the word are partitioned into three consecutive zones* X_1, X, X_2*. Suppose also that in time* $i-1$ *the labels of zone* X *do not change. For every node* $x \in X$*, the node profile* $\pi^{i-1}_{x \in X}$ *depends only on:*

$$\Pi^i_{X_1} \qquad \Pi^i_{X_2} \qquad \textit{and} \qquad \pi^i_{x \in X}$$

Actually, the dependency is even a bit more fine grained. If we want to know the rank l profile $\pi^{l,i-1}_{x \in X}$, we only need the profile of the same rank in the current node, but profiles of smaller rank in the surrounding zones:

$$\Pi^{l-1,i}_{X_1} \qquad \Pi^{l-1,i}_{X_2} \qquad \text{and} \qquad \pi^{l,i}_{x \in X}.$$

The idea is that any dependency on nodes that are different from x requires using a non-local transition, which decreases the rank. However, since we do not need the more fine grained dependency, we use the dependency as stated in Lemma 13, which does not indicate ranks.

Observe that we do not need the profiles $\pi^i_{x' \in X}$ of other elements $x' \in X$ since all the information relevant to X is inside $\pi^i_{x \in X}$. Let us fix the term *Externalzoneprofiles* to name zone profiles of the environment of zones.

$$\textit{Externalzoneprofiles} \stackrel{def}{=} \textit{Externalzoneprofiles}_t$$

$$\textit{Externalzoneprofiles}_l \stackrel{def}{=} \textit{Zoneprofiles}_l \times \textit{Zoneprofiles}_l$$

We define the *ext* function to work with *Externalzoneprofiles*.

$$\textit{ext}(X, i) \stackrel{def}{=} (\Pi^i_{X_1}, \Pi^i_{X_2}) \in \textit{Externalzoneprofiles}$$

for (X_1, X_2) the environment of X in w_i. By Lemma 13, there exists a function

$$\textit{update} : \textit{Externalzoneprofiles} \to \textit{Nodeprofiles} \to \textit{Nodeprofiles}$$

such that the following holds.

$$\pi^{i-1}_{x \in X} = \textit{update}(\textit{ext}(X, i), \pi^i_{x \in X})$$

This function can be extended to zones

$$\textit{update} : \textit{Externalzoneprofiles} \to \textit{Zoneprofiles} \to \textit{Zoneprofiles}$$

by lifting the previous definition to sets of profiles, and we then have

$$\Pi^{i-1}_X = \textit{update}(\textit{ext}(X, i), \Pi^i_X)\ .$$

6.2 Histories

We now generalize the notion of histories, as they were defined in Section 5, to the non-local case. Let $i \leq j$ be times and let X be a zone, with (X_1, X_2) its environment. Suppose that the labels of X do not change in times $i, \ldots, j$ (and therefore they are the same as in the word w_{j+1}). We define the history of X between i and j, denoted by

$$\textit{history}(X, \{i, \ldots, j\}),$$

to be the following sequence:

$$\textit{ext}(X, i), \textit{ext}(X, i+1), \ldots, \textit{ext}(X, j) \in \textit{Externalzoneprofiles}.$$

We write *Histories* for the set of possible histories. We next define the *apply* and *in* functions on *Histories*.

LEMMA 14. *Suppose that in time interval* $\{i, \ldots, j\}$*, the labels of a zone* X *do not change. Then the zone profile* Π^{i-1}_X *is uniquely determined by* Π^j_X *and* $\textit{history}(X, \{i, \ldots, j\})$.

PROOF. By iterating the function

$$\textit{update} : \textit{Externalzoneprofiles} \to \textit{Zoneprofiles} \to \textit{Zoneprofiles}.$$

for each item on the list in the history. □

Let the dependency in the above lemma be realized by the function:

$$\textit{apply} : \textit{Histories} \to \textit{Zoneprofiles} \to \textit{Zoneprofiles}$$

That is, under the assumptions of the lemma we have

$$\Pi^{i-1}_X \quad = \quad \textit{apply} \quad (\textit{history}(X, \{i, \ldots, j\})) \quad (\Pi^j_X).$$

In the algorithm for local transitions from Section 5, one important part has to do with the possibility to compute, given the history h of a zone X in an interval $\{i, \ldots, j\}$ (and assuming that there were no changes on X between times i and j), the history h' of a smaller zone $X_0 \subseteq X$ having as environment inside X the subzones X_1 and X_2. In the previous section, this was obtained simply with $in_\sigma(h)$ where $\sigma \in \textit{Externals}$ corresponds to the types of (X_1, X_2). But here the operation is somewhat more complicated, since to obtain the history h' we need at the same time to update the zone profiles of X_1 and X_2 to the time $j-1$ (remember that a history element is from *Externalzoneprofiles*), then to time $j-2$ until we update it to time i. Thus, in some sense the *apply*() and $in_\sigma()$ functions of the previous section are done simultaneously for the non-local automata. Another issue to solve is that in order to update the zone profile of, say X_1, we need the updated profiles of X_0, X_2. But these need at the same time the updated profile of X_1. However, this will not be a problem because of the *ordered* condition we impose to the automaton.

LEMMA 15. *Let* Y *be a zone partitioned into three consecutive zones* X_1, X, X_2*. Suppose that in time interval* $\{i, \ldots, j\}$ *the labels of zone* Y *do not change. Then*

$$\textit{history}(X, \{i, \ldots, j\})$$

is uniquely determined by

$$\textit{history}(Y, \{i, \ldots, j\}) \qquad \Pi^j_{X_1} \qquad \Pi^j_{X_2} \qquad \Pi^j_X.$$

Let the operation that realizes the dependency of the above lemma be denoted by

$$\textit{in} : \textit{Histories} \to \textit{Externalzoneprofiles} \to \textit{Zoneprofiles} \to \textit{Histories}.$$

That is, under the assumptions of the lemma we have

$$\textit{history}(X, \{i, \ldots, j\}) = \textit{in} \quad (\textit{history}(Y, \{i, \ldots, j\})) \quad (\Pi^j_{X_1}) \quad (\Pi^j_{X_2}) \quad (\Pi^j_X).$$

6.3 The algorithm

The high level structure of the algorithm is the same as in Section 5. We keep a tree of histories and update one branch with each iteration, performing $O(\log(n))$ operations. The main difference is that we also retain the profiles of each of the logarithmic zones, which become necessary to transfer a history of a zone to a history of a smaller zone.

The history tree. We also maintain a structure as the one showed previously. An instance D of the data structure consists of the following labeling.

$$D.lastupdate : Logs \to \{0, \ldots, k\}$$
$$D.lastprofile : Logs \to Zoneprofiles$$
$$D.history : Logs \to Histories$$

Note that instead of using $D.lastlabel$ and $D.monoid$ we have a $D.lastprofile$ which contains the profile of the logarithmic zone. In particular it contains the label and external value for each node.

As before, we first compute a history tree for time k, then for times $k-1$ and so on, down to 0. We call a history tree *correct* at time moment $i \in \{0, \ldots, k\}$ if conditions 1, 2, and 4 of the previous definition of correctness hold (with the new notions of history) and

6. $D.lastprofile(X)$ is the updated zone profile of X, i.e., after applying $D.history(X)$ to it. Suppose that $X \in Logs$ and let

$$j = D.lastupdate(parent(X)) \quad \text{and}$$
$$\Pi = D.lastprofile(X)\ ,$$

then Π is equal to the zone profile Π_X^j of X in the time moment j. If X is the root, then it is the zone profile at time moment i: $D.lastprofile(X) = \Pi_X^i$.

Like with the algorithm of previous section we begin with a history tree D_0 that is correct for time moment $i = k$ and then for each time moment $i = k-1, \ldots, 0$ the algorithm calculates the history tree D_i that is correct for time moment i. The algorithm manipulates histories and zone profiles. Here we use as black boxes the following operations on histories that have unit cost in the algorithm.

$$empty \quad concat \quad apply \quad in$$

emtpy and *concat* have the exact same definition as in the previous section: the empty history and the concatenation of histories. *apply* and *in* follow the definitions seen in Section 6.2.

One states analogues of Lemmas 8 and 9 for the new setup, with the exact same statements, only with the new notion of history tree. Then, one proves that the history operations can be implemented in constant time, also using a history congruence.

7. IMPROVEMENTS

7.1 Offline vs online

There are two ways of looking at the incremental evaluation problem, when the input is a document evolution

$$w_1, \ldots, w_k.$$

In the offline view, the algorithm begins its work once all of the document evolution is known.

In the online view, the words of the document evolution come one at a time, and the algorithm should do logarithmic processing for each word, without knowing the words that are going to come in the future.

Is our algorithm offline or online?

The answer depends on the order in which the words come, or equivalently stated, the direction of time in the temporal logic. As we had defined the temporal logic, time flows to the right, with the first time moment being 1 and the last time moment being k. As the reader will recall, our algorithm begins by analyzing time k, then $k-1$ and so on down to 1. An inspection of the algorithm reveals that operations of the algorithm when doing step i do not depend on the part of the document evolution $w_1, \ldots, w_{i-1}$. Therefore, our algorithm actually is an online algorithm, assuming that the flow of time in the logic or automaton works in the opposite direction as the development of the document evolution.

Summing up, our algorithm is online (i.e. it does a logarithmic computation per each new word of the document evolution, and the computation does not depend of the words that are about to come in the future) if the temporal logic uses past operators on the time dimension, and not future operators. Although future operators are more common in temporal logics (which is why we defined the logic with future operators), it seems that past operators might actually be a better idea for the application at hand.

7.2 Insertion and deletion

In our evaluation algorithm we dealt with only one kind of update operation: relabeling. In this section we show that the algorithm still works if we allow insertions and deletions.

In the presence of insertions and deletions, one has to take care to distinguish between a node and its distance from the beginning of the word. For instance, suppose that we start with a word a and then apply the operation "insert a node with label b before the first position". The resulting document evolution is a, ba. However, the first and only node in the word a corresponds to the second node in the word ba, and not the first one.

To solve the issue presented above, we redefine a word as a connected graph where every node has outdegree and indegree at most one, and the nodes are labeled by letters from the alphabet. This way, we can assume that every node has a unique identifier, and the identifier does not change even if the distance of the node from the beginning of the word changes as a result of an insertion or deletion. A document evolution is a sequence of such words, which share nodes with the same identifiers.

Using identifiers, we adapt the semantics of the temporal and first order logic to document evolutions that include insertions and deletions. In the first-order logic, the quantifiers range over pairs (identifier/time), and not (distance from beginning/time) pairs. In the temporal logic, a modality that stays in a node stays in an identifier. For instance, the formula $\neg \mathsf{X}_t \top$ says that the current node will be deleted in the next time moment. Another example is the formula

$$\mathsf{G}_t\big((\neg \mathsf{X}_t \top) \Rightarrow a\big),$$

which says that the label of the current node is a just before it gets deleted (if it gets deleted).

Thanks to the insertion and deletion operations, we may assume that an incremental document evolution begins in an empty word.

THEOREM 16. *Suppose that, apart from label changes, we also allow insertions and deletions as update operations. For any fixed formula φ of the temporal logic, the incremental evaluation problem can be solved in time $O(k \cdot \log(k))$, where k is the number of updates.*

The same online/offline remarks as in the previous section

apply to the algorithm above. That is, it works online if the words are given in reverse order, or the temporal logic uses past instead of future operators in the time dimension. (But not both changes simultaneously.)

7.3 Trees

In the previous sections, we have described the evaluation algorithm for ordered document update processing automata, assuming input document evolution was a sequence of words. What about the trees?

Bounded depth trees. One solution is to reduce trees to words. For a tree t, we define its word representation $word(t)$, which is like the text representation of an XML tree. If the labels of t are A, then the labels of $word(t)$ are $\{open, close\} \times A$. Every node of t corresponds to two nodes in w, one with an opening tag and one with a closing tag. If the depth of the tree is known in advance, and can be encoded in the states of the automaton, then this representation can be decoded by document update processing automata, as stated in the following lemma.

LEMMA 17. *Fix $d \in \mathbb{N}$. For any document update processing automaton on trees $\mathcal{A}$, one can compute a document update processing automaton on words $\mathcal{A}_d$ such that $\mathcal{A}$ accepts a document evolution (consisting of trees)*

$$\bar{t} = t_1, \ldots, t_k$$

if and only if $\mathcal{A}_d$ accepts the document evolution

$$word(\bar{t}) = word(t_1), \ldots, word(t_k),$$

provided all the trees in $\bar{t}$ have depth at most d.

From the lemma above, it follows that all results on evaluation can be transferred from the bounded depth words to bounded depth trees.

Unbounded depth trees. We believe that, after some modifications, our algorithm can actually work directly on trees, without the need for the reduction $t \mapsto word(t)$. That is, we believe that Theorem 16 holds for trees, without any restriction on the depth. The idea is to use forest algebra [7], instead of monoids. Observe that this would improve the algorithm of Balmin, Papakonstantinou and Vianu [4] in two ways: first, a more general problem is considered, and second the data complexity per update is improved from $O(\log^2(n))$ to $O(\log(n))$. On the other hand, the query complexity of our algorithm is very bad.

8. CONCLUSIONS

We have designed an algorithm for evaluating a logic that inspects documents evolutions. The logic has operators that travel in the time dimension, and operators that travel in the space dimension. The algorithm works in time $O(k \cdot \log(k))$ where k is the length of the document evolution.

Below we list some topics for future work. We would like to investigate a logic with past and future operators on the time axis. We would also like to investigate a hybrid logic, where quantifiers are used for the space dimension and temporal operators are used for time dimension. Finally, we would like to improve the query complexity of the algorithm, possibly at the cost of using weaker logics.

9. REFERENCES

[1] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, and Mayank Saksena. Regular model checking for LTL(MSO). In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 348–360. Springer, 2004.

[2] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena. A survey of regular model checking. In *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer, 2004.

[3] Serge Abiteboul, Laurent Herr, and Jan Van den Bussche. Temporal connectives versus explicit timestamps to query temporal databases. *J. Comput. Syst. Sci.*, 58(1):54–68, 1999.

[4] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[5] Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. Efficient incremental validation of XML documents. In *ICDE*, pages 671–682, 2004.

[6] Henrik Björklund, Wouter Gelade, Marcel Marquardt, and Wim Martens. Incremental XPath evaluation. In *ICDT*, pages 162–173, 2009.

[7] M. Bojańczyk and I. Walukiewicz. Forest algebras. In *Automata and Logic: History and Perspectives*, pages 107–132. Amsterdam University Press, 2007.

[8] Mikołaj Bojańczyk and Paweł Parys. XPath evaluation in linear time. In *PODS*, pages 241–250, 2008.

[9] Ghislain Fourny, Daniela Florescu, Donald Kossmann, and Markos Zacharioudakis. A time machine for XML: PUL composition. In *XML Prague*, 2010.

[10] David Gabelaia, Agi Kurucz, Frank Wolter, and Michael Zakharyaschev. Products of 'transitive' modal logics. *J. Symb. Log.*, 70(3):993–1021, 2005.

[11] Leonid Libkin. Logics for unranked trees: An overview. *Logical Methods in Computer Science*, 2(3), 2006.

[12] Bernd-Holger Schlingloff. Expressive completeness of temporal logic of trees. *Journal of Applied Non-Classical Logics*, 2(2), 1992.

[13] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, pages 41–57, 2006.

APPENDIX

A. IMPORTANCE OF THE ORDER

In this section we provide some evidence that the order, in ordered automata, is important. The evidence is the following implication: If we could evaluate document update processing automata in time $O(k \cdot \log(n))$ without the order assumption, then we would get an implausibly fast algorithm for evaluating cellular automata. Consider a one-dimensional cellular automaton, with cell colors C, given by a function

$$f : (C \cup \{\vdash\}) \times C \times (C \cup \{\dashv\}) \to C \ .$$

We can extend this function in the natural way to configurations

$$f : C^* \to C^*$$

Consider an input word $w \in C^*$ and a duration $k \in \mathbb{N}$. Suppose that we want to compute the value of the first cell in the word $f^k(w)$. It seems unlikely that there is a better algorithm than computing all the cells on which this cell depends, which takes time $O(k \cdot n)$. The following lemma shows that evaluating cellular automata can be reduced to evaluating document update processing automata, and therefore it is unlikely that the $O(k \cdot \log(n))$ works without the assumption on order.

LEMMA 18. *Let f be a one-dimensional cellular automaton f with cell colors C and let $c \in C$. One can compute a document update processing automaton $\mathcal{A}_{f,c}$ such that any word $w \in C^*$, the automaton $\mathcal{A}_C$ accepts the document evolution*

$$\bar{w} = \overbrace{w, \ldots, w}^{k\text{-times}}$$

if and only if $f^k(w)$ has c on the first cell position. In the above, the word w is treated as a tree with a single path.

B. APPENDIX TO SECTION 5

B.1 Proof of Lemma 7

The history tree D_k is built bottom-up. We first define D_k for all the leaves, and we continue by levels up to the root.

We start with the singleton zones $\{x\}$ for every node x of w_k. We define $D_k.lastlabel(\{x\}) = (a, F)$, where a is the label of node x in w_k and F is the set of accepting states of the automaton. We define $D_k.monoid(\{x\}) = \alpha(a)$, $D_k.lastupdate(\{x\}) = k$ and $D_k.history(\{x\}) = empty$.

Once we have computed D_k for all the tree nodes of depth at least $l + 1 > 0$, we compute the logarithmic zones of depth l by combining pairs of logarithmic zones of depth $l + 1$ in order, from left to right. For each such logarithmic zone X we define $D_k.lastupdate(X) = k$, $D_k.history(X) = empty$ and $D_k.monoid(X)$ to be the product of the monoid elements of its children logarithmic zones.

B.2 Proof of Lemma 9

Suppose that the label of x in w_i is a. Consider all the logarithmic zones that contain x,

$$\{x\} = X_1 \subsetneq X_2 \subsetneq \cdots \subsetneq X_m$$

We first precompute the following information. For every X_j we compute the external type σ_j of X_j in w_i. This can be done in linear time in m.

We iterate from X_1 up to X_m making updates to the history tree. We first start with $X_1 = \{x\}$. We update $D'.lastlabel(x)$ to

$$apply(D'.history(\{x\}))(a, R)$$

where $D'.lastlabel(x) = (a', R)$. We also update $D'.monoid(\{x\})$ to $\alpha(a)$ and $D'.lastupdate(\{x\})$ to i.

For any other X_j with $j > 1$ we proceed as follows. Suppose Y is the right sibling of X_{j-1}, i.e. such that $parent(X_{j-1}) = parent(Y) = X_j$ and Y is to the left of X_{j-1}. We then update $D'.monoid(X_j)$ to

$$D'.monoid(X_{j-1}) \cdot D'.monoid(Y)$$

and $D'.lastupdate(X_j)$ to i. Finally, we need to propagate this changes by using the history on Y. We then update $D'.history(Y)$ to $conc(D'.history(Y), \tau)$ with $\tau = in_{(D'.monoid(X_{j-1}),1)}(\sigma_j)$, where σ_j is the already precomputed the external type of X_j in w_i. A similar update occurrs if Y is a left sibling of X_{j-1}.

B.3 Proof of Lemma 11

To a history h we associate the mapping

$$\tau \in Externals \quad \mapsto \quad apply(in_\tau(h)) : A \times P(Q) \to P(Q)$$

which we call its signature of h and denote by sig_h.

Consider the equivalence relation on histories which considers two histories equivalent if they have the same signature. There are finitely many equivalence classes because there are finitely many signatures. We will prove that this equivalence is a history congruence. We need to show the following items.

- The signature of the $concat(g, h)$ depends only of the signatures of g and h.

  $$\tau \in Externals \qquad \mapsto \qquad (a \in A \mapsto f_{a,\tau})$$

  where $f_{a,\tau} : P(Q) \to P(Q)$ is the composition of the functions

  $$sig_h(\tau)(a) : P(Q) \to P(Q) \quad sig_g(\tau)(a) : P(Q) \to P(Q).$$

- The value of $apply(h)$ depends only on the signature of h. Indeed,

  $$apply(h) = sig_h(\sigma)$$

  where σ is the empty external type $(1, 1)$.

- For any external type σ, the value of $in_\sigma(h)$ depends only on the signature of h. Indeed, the signature of $in_\sigma(h)$ is the function

  $$\tau \in Externals \qquad \mapsto \qquad sig_h(in_\sigma(\tau)).$$

C. APPENDIX TO THE SECTION ON NON-LOCAL TRANSITIONS

Proof of Lemma 12.

Given two zone profiles Π_{X_1} and Π_{X_2} with X_1 and X_2 consecutive zones, we write $\Pi_{X_1} \cdot \Pi_{X_2}$ for the zone profile $\Pi_{X_1 \cup X_2}$. This is an operation that depends solely on the information contained in Π_{X_1} and Π_{X_2}.

$$\Pi_{X_1} \cdot \Pi_{X_2} \stackrel{def}{=} out_{(1,\tau_2)}(\Pi_{X_1}) \cup out_{(\tau_1,1)}(\Pi_{X_2})$$

where τ_1 is the zone type of Π_{X_1}, and τ_2 that of Π_{X_2} (these are the monoid's identity if the zones are empty), and $out_{(\tau_1,\tau_2)}(\Pi)$ is defined as follows by induction on l of the type $Zoneprofiles_l$ of Π. Assume $\Pi \in Zoneprofiles, \Pi^l \in Zoneprofiles_l$.

$$out_{(\tau_1,\tau_2)}(\Pi) \stackrel{def}{=} out_{(\tau_1,\tau_2)}(\Pi^t)$$

$$out_{(\tau_1,\tau_2)}(\Pi^1) \stackrel{def}{=} \{((\tau_1 \cdot \sigma_1, \sigma_2 \cdot \tau_2), \rho) \mid ((\sigma_1, \sigma_2), \rho) \in \Pi^1\}$$

$$out_{(\tau_1,\tau_2)}(\Pi^{l+1}) \stackrel{def}{=} \{((\tau_1 \cdot \sigma_1, \sigma_2 \cdot \tau_2), \rho, f') \mid ((\sigma_1, \sigma_2), \rho, f) \in \Pi^{l+1}, f' = [(\sigma_1', \sigma_2') \mapsto out_{(\tau_1,\tau_2)}(f(\tau_1 \cdot \sigma_1', \sigma_2' \cdot \tau_2))]\}$$

Proof of Lemma 15.

We give a rough idea of how *in* can be defined, which realizes the dependency stated in the Lemma. Given a one-element history $h = (\Pi_{X_1}^{t,i+1}, \Pi_{X_2}^{t,i+1})$ and $\Pi_{Y_1}^{t,i}, \Pi_{Y_0}^{t,i}, \Pi_{Y_2}^{t,i}$ three consecutive zone profiles, we can calculate $in(h)(\Pi_{Y_1}^{t,i}, \Pi_{Y_2}^{t,i})(\Pi_{Y_0}^{t,i})$ defined as

$$in(h)(\Pi_{Y_1}^{t,i}, \Pi_{Y_2}^{t,i})(\Pi_{Y_0}^{t,i}) = (\Pi_{X_1}^{t,i+1} \cdot \Pi_{Y_1}^{t,i+1}, \Pi_{Y_2}^{t,i+1} \cdot \Pi_{X_2}^{t,i+1})$$

with

$$\Pi_{Y_0}^{0,i+1} = \Pi_{Y_0}^{0,i} \qquad \Pi_{Y_1}^{0,i+1} = \Pi_{Y_1}^{0,i} \qquad \Pi_{Y_2}^{0,i+1} = \Pi_{Y_2}^{0,i}$$

$$\begin{aligned}
\tau_0^{l-1} &= (\Pi_{X_1}^{l-1,i+1} \cdot \Pi_{Y_1}^{l-1,i+1,} \Pi_{Y_2}^{l-1,i+1,} \cdot \Pi_{X_2}^{l-1,i+1}) \\
\Pi_{Y_0}^{l,i+1} &= update_l(\tau_0^{l-1})(\Pi_{Y_0}^{l,i}) \\
\tau_1^{l-1} &= (\Pi_{X_1}^{l-1,i+1}, \Pi_{Y_0}^{l-1,i+1,} \cdot \Pi_{Y_2}^{l-1,i+1,} \cdot \Pi_{X_2}^{l-1,i+1}) \\
\Pi_{Y_1}^{l,i+1} &= update_l(\tau_1^{l-1})(\Pi_{Y_1}^{l,i}) \\
\tau_2^{l-1} &= (\Pi_{X_1}^{l-1,i+1} \cdot \Pi_{Y_1}^{l-1,i+1} \cdot \Pi_{Y_0}^{l-1,i+1,}, \Pi_{X_2}^{l-1,i+1}) \\
\Pi_{Y_2}^{l,i+1} &= update_l(\tau_2^{l-1})(\Pi_{Y_2}^{l,i})
\end{aligned}$$

for every $l \in \{1, \ldots, t\}$. This definition can be extended to any number of history items by iterating the above definition.

C.1 The algorithm

As stated in Section 6.3, we show that the analogues of Lemmas 8 and 9 hold for the new notion of history tree.

LEMMA 19. *Let D_i be a history tree that is correct for time i. Let x be a position. We can compute a history tree D_i' that is also crrect for time i and which is up to date at x. This computation can be done in time* $\log(n)$*, assuming the history operations have unit cost.*

PROOF. In our structure, in order to make a position x at time i up to date, we first find the biggest logarithmic zone $X \ni x$ such that it is not up to date, i.e., such that $D_i.lastupdate(X) > i$. Suppose Y and Y' are the children logarithmic zones of X. By

$$in(D.history(X), (\emptyset, D.lastprofile(Y')), D.lastprofile(Y))$$

we obtain the history h_Y and we define

$$D.history(Y) = conc(D.history(Y), h_Y)$$

and by

$$in(D.history(X), (D.lastprofile(Y), \emptyset), D.lastprofile(Y'))$$

we obtain the history $h_{Y'}$ and we define

$$D.history(Y') = conc(D.history(Y'), h_{Y'}).$$

Now we define

$$D.lastprofile(Y) = apply(h_Y)(D.lastprofile(Y))$$

and

$$D.lastprofile(Y') = apply(h_{Y'})(D.lastprofile(Y')) \ .$$

If $X = \{x\}$, we apply the history to the label of x, and define

$$D.lastlabel(X) = apply(D.history(X))(D.lastlabel(X)) \ .$$

Now X is up to date, and we set $D.lastupdate(X) = i$ and $D.history = empty$. We iterate at most $O(\log(n))$ times (i.e., the height of the logarithmic tree) to arrive to the desired state of our structure, where x is up to date. □

LEMMA 20. *Let x be the position where w_{i+1} differs from w_i. Let D_{i+1}' be a history tree that is correct for time i and up to date at x. We can compute a data structure D_i that is correct for time w_i. This computation can be done in time* $\log(n)$*, assuming the history operations have unit cost.*

PROOF. Consider $X_1, \ldots, X_m$ the logarithmic zones such that

- $X_1 \cup \cdots \cup X_m = \{1, \ldots, n\}$,
- X_j and X_{j+1} are consecutive zones for every $j < m$,
- there exists some $X_r = \{x\}$,
- m is minimal.

Then $m \leq \log(n)$. Note that from D_{i+1}' we have that $D_{i+1}'.lastprofile(X_j) = \Pi_{X_j}^{i+1}$ for every $j \in \{1, \ldots, m\}$, this is granted by D_{i+1}' being updated for all logarithmic zones containing x at time $i+1$. Given $\Pi_{X_1}^{i+1}, \ldots, \Pi_{X_m}^{i+1}$ and $\Pi_{X_r}^{i}$ we obtain $\Pi_{X_1}^{i}, \ldots, \Pi_{X_m}^{i}$ using the *update* operation. This is done in $O(m)$. To obtain D_i we modify each $D_{i+1}'.lastprofile(X_j)$ to $\Pi_{X_j}^{i}$ and $D_{i+1}'.history(X_j)$ to $conc(D_{i+1}'.history(X_j), \tau)$, with

$$\tau = (\Pi_{X_1}^{i+1} \cdot \cdots \cdot \Pi_{X_{j-1}}^{i+1}, \Pi_{X_{j+1}}^{i+1} \cdot \cdots \cdot \Pi_{X_m}^{i+1})$$

These external zone profiles can be computed in $O(m)$. We first compute $\Pi_{X_1}^{i+1} \cdot \cdots \cdot \Pi_{X_j}^{i+1}$ for all j in one pass, starting from $j = 1$ up to $j = m$, and then all $\Pi_{X_j}^{i+1} \cdot \cdots \cdot \Pi_{X_m}^{i+1}$ from $j = m$ to $j = 1$ in another pass. Finally, we set $D_{i+1}'.lastupdate(Y) = i$ for all $Y \ni x$. □

C.2 Constant time implementation of non-local histories

In a very similar way as for the section on local transitions, we give a congruence of finite index over *Histories*.

$$\begin{aligned}
&empty \in Histories \\
&concat : Histories \times Histories \to Histories \\
&apply : Histories \to Zoneprofiles^2 \to Zoneprofiles^2 \\
&in : Histories \to Externalzoneprofiles \\
&\qquad \to Zoneprofiles \to Histories
\end{aligned}$$

We define a history congruence such that whenever $h \sim h'$

$$\begin{array}{rcll} concat(g,h) & \sim & concat(g,h') & \text{for all } g \in Histories \\ concat(h,g) & \sim & concat(h',g) & \text{for all } g \in Histories \\ apply(h) & = & apply(h') & \\ in(h)(\tau)(\Pi) & \sim & in(h')(\tau)(\Pi) & \\ & & \text{for all } \tau \in Externalzoneprofiles, & \\ & & \Pi \in Zoneprofiles & \end{array}$$

LEMMA 21. *There is a history congruence where the number of equivalence classes is finite and depends only on the automaton, and not its input (not on n or k).*

PROOF. To a history h we associate the signature of h

$$sig_h : Zoneprofiles^2 \to Zoneprofiles \to Zoneprofiles$$

The idea is that a history signature sig_h describes, for a given zone X, how to split the history into two smaller zones $Y \subseteq X$. In order to do this, we need the zone profiles of the environment of Y in X. Let Y_1 and of Y_2 be the left and right environment, such that $Y_1 \cup Y \cup Y_2 = X$, and let Π_1, Π_2 be the two respective zone profiles of Y_1 and Y_2. Then $sig_h(\Pi_1, \Pi_2)$ is defined as the result of applying the history that corresponds to Y on Π.

$$sig_h(\Pi_1,\Pi_2)(\Pi) = apply(in(h)(\Pi_1,\Pi_2)(\Pi))(\Pi)$$

Consider the equivalence relation on histories which considers two histories equivalent if they have the same signature. There are finitely many equivalence classes because there are finitely many signatures. We will prove that this equivalence is a history congruence. We need to show the following items.

- The signature of the $concat(g,h)$ depends only of the signatures of g and h.

$$\begin{array}{rl} & \Pi_1, \Pi_2 \mapsto \Pi \mapsto sig_g(\Pi_1', \Pi_2')(\Pi) \\ \text{where} & \Pi' = sig_h(\Pi_1,\Pi_2)(\Pi)\ , \\ & \Pi_1' = sig_h(\emptyset, \Pi \cdot \Pi_2)(\Pi_1) \text{ and} \\ & \Pi_2' = sig_h(\Pi_1 \cdot \Pi, \emptyset)(\Pi_2)\ . \end{array}$$

- The value of $apply(h)$ depends only on the signature of h. Indeed,

$$apply(h) = sig_h(\emptyset,\emptyset)$$

since $in(h)(\emptyset,\emptyset)(\Pi) = h$.

- The value of $in(h)$ depends only on the signature of h. Indeed, the signature of $in(h)(\Pi_1,\Pi_2)(\Pi)$ is the function

$$(\Pi_1', \Pi_2') \mapsto \Pi' \mapsto sig_h(\Pi_1 \cdot \Pi_1', \Pi_2' \cdot \Pi_2)(\Pi')$$

□

D. APPENDIX TO SECTION 7

D.1 Proof of Theorem 16

The basic idea is to rebalance the history trees.

Since the set of positions changes dynamically, our notion of logarithmic zone will change. A decomposition into zones, as in the logarithmic zones, will be called a zone tree. A *zone tree* for a word w is a family $\mathcal{X}$ of zones in w subject to the following conditions.

- Every two zones in $\mathcal{X}$ are disjoint or one is included in the other.
- The zone with all nodes of the word belongs to $\mathcal{X}$.
- Every singleton zone belongs to $\mathcal{X}$.
- Every non-singleton zone in $\mathcal{X}$ is a union of two zones in $\mathcal{X}$.

There is a natural tree structure on a zone tree, which we use to talk about things like children zones or the height of a zone tree. The family of logarithmic zones is a special case of a zone tree, which has logarithmic height.

We now show how to adapt the evaluation algorithm to take into account insertion and deletion operations. Suppose that we have an incremental document evolution

$$w_1, \ldots, w_k$$

where the update operations are insertions, deletions and relabelings. Without loss of generality, we assume that w_k is empty. Our algorithm will process the evolution from last word to first word. We will compute:

- Zone trees $\mathcal{X}_1, \ldots, \mathcal{X}_k$ for the words $w_1, \ldots, w_k$ of height at most $O(\log(k))$. These zone trees are balanced in the sense of AVL trees: for every two siblings, the difference in depths of their subtrees is in $\{-1, 0, 1\}$. Because the last word w_k is empty, the zone tree $\mathcal{X}_k$ is empty, too.
- History trees $D_0, \ldots, D_{k-1}$, such that D_i is correct in time i, with $\mathcal{X}_{i+1}$ playing the role of the logarithmic intervals.

Maintaining the zone trees. We begin with describing the zone trees $\mathcal{X}_1, \ldots, \mathcal{X}_k$. Suppose that we have the zone tree $\mathcal{X}_{i+1}$ and we want to compute the zone tree $\mathcal{X}_i$. If the words w_{i+1} and w_i have the same positions, then we do not need to do anything. Suppose that w_i has a position x, that does not appear in w_{i+1} (i.e. there is a deletion between w_i and w_{i+1}). The insertion case is done the same way. Let y be a position adjacent to x (to the left or right, does not matter). To all the zones in $\mathcal{X}_{i+1}$ that contain y we add position x, and then we add two singletons $\{x\}$ and $\{y\}$. Now we need to rebalance the tree.

As is known from AVL trees, rebalancing requires a logarithmic number of rotations. What is a rotation in the case of zone trees? We say a zone tree $\mathcal{X}'$ is obtained from a zone tree $\mathcal{X}$ by a *right rotation*, if there are three consecutive zones X_1, X_2, X_3 in the zone tree $\mathcal{X}$ such that

$$\mathcal{X}' = \mathcal{X} - (X_1 \cup X_2) \cup \{X_2 \cup X_3\}.$$

In particular, a right rotation removes one zone and adds another. A left rotation is the inverse operation.

The key property of the zone trees is as follows: the zone tree $\mathcal{X}_i$ is obtained from the zone tree $\mathcal{X}_{i+1}$ by first removing $O(\log(k))$ zones, and then adding $O(\log(k))$ zones.

Maintaining the history trees. We now give a very rough sketch of how the history trees are maintained. Suppose that we have computed D_i and we want to compute D_{i-1}. The history tree D_{i+1} consists of mappings defined on the zones from $\mathcal{X}_{i+1}$, and the history tree D_i consists of mappings defined on the zones from $\mathcal{X}_i$.

Consider the case when w_i has a position x, that does not appear in w_{i+1}. The case of an insertion from w_i to w_{i+1} is

done the same way, and the case of relabeling was already studied.

Using the non-local version of Lemma 8, we prepare D_i so that all of the zones from $\mathcal{X}_{i+1} - \mathcal{X}_i$ are up to date. This requires time logarithmic in the depth of the history tree. We now need to define the zone profile of the zone $\{x\}$ in D_{i-1}. Since the node is new (i.e. it does not exist in w_{i+1}) and because the zone has one element, this zone profile is easy to compute (actually, the zone profile a new singleton zone is always the same). We are still missing the zone profiles assigned to the zones of $\mathcal{X}_i - \mathcal{X}_{i+1}$. These can be built using the concatenation operation from Lemma 12.