

# Streamable Fragments of Forward XPath

Olivier Gauwin<sup>3</sup> and Joachim Niehren<sup>1,2</sup>

<sup>1</sup> Mostrare project, INRIA & LIFL (CNRS UMR8022)

<sup>2</sup> INRIA, Lille

<sup>3</sup> University of Mons

**Abstract.** We present a query answering algorithm for a fragment of Forward XPath on XML streams that we obtain by compilation to deterministic nested word automata. Our algorithm is earliest and in polynomial time. This proves the finite streamability of the fragment of Forward XPath with child steps, outermost-descendant steps, label tests, negation, and conjunction (aka filters), under the reasonable assumption that the number of conjunctions is bounded. We also prove that finite streamability fails without this assumption except if  $P=NP$ .

**Keywords :** tree automata, pushdown automata, query answering, XML streams, XPath, temporal logics for unranked trees.

## 1 Introduction

Query answering algorithms for XPath on XML streams received much interest in the database and document processing communities [2,24,3,22,5,12,29,6,21] and are currently in the focus of the W3C working groups on XSLT and XPROC [14]. A little surprisingly, the topic is far from being settled given the large remaining gap between known streamable and non-streamable fragments. The objective of this paper is to narrow this gap by providing new positive and negative results for fragments of Forward XPath. Our approach relies on the relationship between temporal logics for unranked trees [19], which abstracts from the concrete syntax of XPath, and tree automata for XML streams [1,20,16].

Streaming is particularly relevant for data collections that are too large to be stored in main memory. Instead, incremental processing is needed in order to buffer only small parts of the data collection at every time point. In the easiest case, a stream is a word over some finite alphabet and a query selects some elements of this word, for instance all  $a$ -positions with two subsequent  $b$ 's. Usually, a query is considered streamable if there exists a one pass algorithm (see e.g. [26]) that computes the set of query answers with constant memory, independently of the input stream [28,27]. Note however, that streaming algorithms for element selection queries need to buffer all *alive* elements, i.e. those positions which might be selected in some continuations of the stream but not in others. In the above example, there exists at most one alive  $a$ -element at every time point, so this query can indeed be answered with bounded memory for all possible input streams.

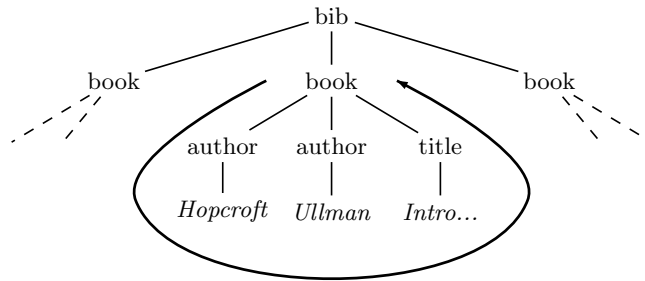


Fig. 1: Sample XML document describing a bibliography.

The case of XML streams is similar except that they contain linearizations of unranked data trees and that queries select nodes in such trees. Consider for instance the XPath query  $/bib/book[author="Ullman"]/author$  that selects all co-authors of Ullman (including himself) in all books of some bibliography (as illustrated in Fig. 1), or more precisely, all *author*-children of *book*-nodes that have at least one *author*-child with data value “Ullman”. An *author*-child of a *book*-node is alive, once the corresponding opening tag was seen on the stream, and as long as the closing *book* tag was not met and no *author*-node with data “Ullman” has been read. For bibliographies, in which all books have a bounded number of authors, the maximal number of alive nodes is bounded, so that the above query can be answered with bounded memory. For unusual bibliographies, however, the number of alive candidates may grow without any bound. As a consequence, the above query is not streamable in the usual sense even though it should be intuitively.

We propose the more liberal notion of *finite streamability* for languages of node selection queries on unranked trees. Finite streamability allows the memory to grow polynomially with the number of alive candidates, the size of the query, and the depth of the tree. In order to enable negative results, we assume in addition that the computation time per step is polynomial in the above parameters, and that the memory grows at least linearly with the number of alive candidates. The latter assumptions hold for all streaming algorithms without compression tricks for representing sets of alive candidates, an assumption that is satisfied by all streaming XPath algorithms in the literature so far.

An overview on finite streamability results for XPath fragments is given in Fig. 2. Despite of the intended weakness of this notion, only few positive results exist so far. Backward XPath (BXP) was proved finitely streamable based on transducers networks [5]. BXP queries never have any alive candidate since node selection is always determined at opening time. The second positive result [3] applies to FXP  $(ch, o-ch_a^*, \wedge)^{thin}$ , a thin fragment of positive Forward XPath on non-recursive documents, with star-restricted child steps, label-guarded (and thus outermost) descendants steps, and conjunctions (and thus filters in official XPath syntax). The only negative result so far got established for FXP  $(ch, ns^*, \wedge, \vee)$ , the fragment of positive Forward XPath with child and following-sibling axes,

	bounded number of $\wedge$	unbounded number of $\wedge$
BXP	yes [5]	yes [5]
FXP $(ch, o-ch_a^*, \wedge)^{thin}$	yes [3]	yes [3]
FXP $(ch, \wedge, \neg)$	yes	no
FXP $(ch, o-ch_a^*, \wedge, \neg)$	yes	no
FXP $(ch, o-ch_a^*, ns, \wedge, \neg)$	?	no
FXP $(ch, ns^*, \wedge, \vee)$	?	no [5]
FXP $(ch, ch^*, \wedge, \neg)$	?	no

Colored results derive from the present paper. We assume here that  $P \neq NP$ .

Fig. 2: Finite streamability of fragments of XPath.

conjunction, and disjunction [5]. There, a counter example from online verification [18] was adapted in order to show for a family of queries in this fragment, that every streaming algorithm answering them must produce a doubly exponential number of states, and thus be of exponential size at least. This result applies even to Boolean queries (without node selection).

In this paper we study  $\text{FXP}(ch, o-ch_a^*, \wedge, \neg)$ , the fragment of Forward XPath with child axis, outermost descendant axis, conjunction, and negation. An outermost descendant axis  $o-ch_a^*$  selects all  $a$ -descendants reachable via non- $a$ -descendants. Outermost constraints on descendant steps are a natural restriction for streaming algorithms as noticed for instance in the XSLT 2.1 definition [15]. Our first main result is a streaming algorithm for  $\text{FXP}(ch, o-ch_a^*, \wedge, \neg)$  that shows that this query language becomes finitely streamable if its queries are restricted to a bounded number of conjunctions. This result is relevant for the W3C pipeline language XPROC, for instance, where Forward XPath queries with at most 3 filters (and thus conjunctions) appear to be enough. Our second main result is the failure of finite streamability for  $\text{FXP}(ch, \wedge, \neg)$  except if  $P=NP$ . It shows the necessity to bound the number of conjunctions theoretically.

We obtain our streaming algorithm by compiling  $\text{FXP}(ch, o-ch_a^*, \wedge, \neg)$  to deterministic nested word automata (dNWAs) [1]. These are tree automata processing linearizations of unranked trees in preorder in a single pass, while mixing top-down and bottom-up determinism. For queries with a fixed number of conjunctions, our compiler is in polynomial time. Otherwise it is in exponential time, while still avoiding the usual doubly-exponential blow-up for translating XPath to deterministic automata [7]. Since the query language defined by dNWAs is finitely streamable [11], the finite streamability follows for all fragments of  $\text{FXP}(ch, o-ch_a^*, \wedge, \neg)$  with a bounded number of conjunctions.

*Outline.* Section 2 introduces FXP and Section 3 recalls dNWAs. In Section 4 we present our compiler from FXP to dNWAs. Section 5 introduces the notion of finite streamability and states our main results, positive and negative. Further related work is discussed in Section 6. The short CIAA version contains only sketches or ideas of proofs. Complete proofs are available in the long version [9].

$$\begin{array}{ll}
\llbracket F_1 \wedge F_2 \rrbracket_{t,\mu} = \llbracket F_1 \rrbracket_{t,\mu} \cap \llbracket F_2 \rrbracket_{t,\mu} & \llbracket d(F) \rrbracket_{t,\mu} = \{\pi \mid \exists \pi' \in \llbracket F \rrbracket_{t,\mu}. (\pi, \pi') \in d^t\} \\
\llbracket \neg F \rrbracket_{t,\mu} = \text{nod}(t) - \llbracket F \rrbracket_{t,\mu} & \llbracket a(F) \rrbracket_{t,\mu} = \{\pi \mid a = \text{lab}^t(\pi)\} \cap \llbracket F \rrbracket_{t,\mu} \\
\llbracket \text{true} \rrbracket_{t,\mu} = \text{nod}(t) & \llbracket x \rrbracket_{t,\mu} = \{\mu(x)\}
\end{array}$$

Fig. 3: Semantics of FXP( $ch, ch^*, \wedge, \neg$ ) formulas.

## 2 FXP

We present FXP temporal logics for unranked trees, which abstract from various aspects of the Forward XPath concrete syntax. More general temporal logics are reviewed by Libkin in [19] for instance (except for variables that we use for node selection here such as in hybrid logic).

For a finite label set  $\Sigma$ , we define the set of unranked trees  $\mathcal{T}_\Sigma$  to be the least set such that  $a(t_1, \dots, t_k) \in \mathcal{T}_\Sigma$  if  $a \in \Sigma$ ,  $k \geq 0$  and  $t_i \in \mathcal{T}_\Sigma$  for all  $1 \leq i \leq k$ . We write  $\text{nod}(t)$  for the set of nodes of the tree  $t$ ,  $\epsilon$  for its root node, and  $\text{lab}^t(\pi)$  for the label of node  $\pi$  of  $t$ . By  $ch^t$  and  $ch^{*t}$  we denote the child and descendant relations of  $t$  respectively. We will also use the outermost descendant relation  $(o\text{-}ch_a^*)^t$  which navigates to all  $a$ -descendants reachable over non- $a$ -descendants. A monadic node selection query  $\Phi$  over  $\Sigma$  is a total function that maps trees  $t \in \mathcal{T}_\Sigma$  to set of tuples of nodes  $\Phi(t) \subseteq \text{nod}(t)$ .

The temporal logic FXP( $ch, o\text{-}ch_a^*, \wedge, \neg$ ) is a query language for node selection in unranked trees, in which one can talk about outermost  $a$ -descendants and children while using negation and conjunction. The expressions of this logic are terms with a single fixed free variable  $x$  (for the selecting position) over the ranked signature  $\Delta = \{\wedge, \neg, \text{true}, x\} \cup \mathcal{D} \cup \Sigma$  where  $\mathcal{D} = \{ch\} \cup \{o\text{-}ch_a^* \mid a \in \Sigma\}$ . These terms have the following form where  $d \in \mathcal{D}$  and  $a \in \Sigma$ .

$$F ::= F_1 \wedge F_2 \mid \neg F \mid \text{true} \mid d(F) \mid a(F) \mid x$$

FXP( $ch, o\text{-}ch_a^*, \wedge, \neg$ ) corresponds to a natural class of Forward XPath expressions in the official XPath syntax modulo linear time transformations. The XPath expression  $/ch^*::a[ch::b]/ch::*$  for instance becomes  $ch^*(a(ch(x) \wedge ch(b(\text{true}))))$ . Note that XPath filters are mapped to conjunctions in FXP.

Given a tree  $t$  and a variable assignment  $\mu : \{x\} \rightarrow \text{nod}(t)$ , we define a set valued semantics  $\llbracket F \rrbracket_{t,\mu} \subseteq \text{nod}(t)$  for all formulas in Fig. 3. Path expression  $F$  defines the monadic query  $\llbracket F \rrbracket$  that selects the following nodes for  $t \in \mathcal{T}_\Sigma$ :

$$\llbracket F \rrbracket(t) = \{\mu(x) \mid \epsilon \in \llbracket F \rrbracket_{t,\mu}, \mu : \{x\} \rightarrow \text{nod}(t)\}$$

The size  $|F|$  is the usual size of term  $F$  and its (conjunction) *width* is the number of leaves in  $F$ .

Smaller fragments of FXP( $ch, o\text{-}ch_a^*, \wedge, \neg$ ) can be obtained by removing some of the operators. For instance, we will write FXP( $ch, \wedge, \neg$ ) for the fragment using only the  $ch$  axis, conjunction and negation. The dialect of FXP( $ch, ch^*, \wedge, \neg$ ) is obtained by allowing for arbitrary descendant axis instead of only outermost  $a$ -descendants.

### 3 Deterministic Automata for XML Streams

We recall the notion of deterministic nested word automata (dNWAs) [1] following their presentation as streaming tree automata [8], and illustrate how to run them on XML streams. Similar kinds of tree automata were proposed for processing XML streams already in [20,16,17]. Note that these tree automata provide an explicit “visual” stack in contrast to standard tree automata.

XML streams are linearizations of unranked trees. The unranked tree  $a(b, c)$  for instance becomes the XML stream  $\langle \mathbf{a} \rangle \langle \mathbf{b} \rangle \langle / \mathbf{b} \rangle \langle \mathbf{c} \rangle \langle / \mathbf{c} \rangle \langle / \mathbf{a} \rangle$  where  $\langle \mathbf{a} \rangle$  is an opening tag and  $\langle / \mathbf{a} \rangle$  a closing tag. The events of the preorder traversal of a tree  $t$  are defined as follows (where **op** marks opening and **cl** closing events):

$$eve(t) = \{\mathbf{start}\} \cup (\{\mathbf{op}, \mathbf{cl}\} \times nod(t))$$

Hence,  $eve(a(b, c)) = \{\mathbf{start}, (\mathbf{op}, \epsilon), (\mathbf{op}, \pi_1), (\mathbf{cl}, \pi_1), (\mathbf{op}, \pi_2), (\mathbf{cl}, \pi_2), (\mathbf{cl}, \epsilon)\}$ , where  $\pi_i$  denotes here the  $i$ th child of the root. All events in  $eve(t)$  except for **start** can be identified with a precise position in the XML stream for  $t$ . The event set is totally ordered with **start** as least element. We denote this order by  $\preceq$  and for an event  $\eta \neq \mathbf{start}$  we write  $pr_{\preceq}(\eta)$  for the immediately preceding event wrt.  $\preceq$ .

**Definition 1.** A dNWA is a tuple  $(\Sigma, Q, \Gamma, i, F, \delta)$  where  $\Sigma$  is a finite alphabet,  $Q$  a finite set of states with a distinguished initial state  $i \in Q$  and final states  $F \subseteq Q$ ,  $\Gamma$  a finite set of stack symbols, and  $\delta$  a set of rules. For each state  $q_0 \in \text{stat}$  and letter  $a \in \Sigma$ , there is at most one rule  $q_0 \xrightarrow{\mathbf{op} \ a: \gamma} q_1$  in  $\delta$ , and for each  $q_0 \in Q$ ,  $a \in \Sigma$ , and  $\gamma \in \Gamma$ , it contains at most one rule  $q_0 \xrightarrow{\mathbf{cl} \ a: \gamma} q_1$ .

A configuration of a dNWA  $A$  on a tree  $t$  consists of an event of  $t$ , a state of  $Q$ , and a stack of elements in  $\Gamma$ . An opening rule  $q_0 \xrightarrow{\mathbf{op} \ a: \gamma} q_1$  can be applied to a configuration that opens some  $a$ -node in state  $q_0$ . In this case, the subsequent configuration is reached by pushing  $\gamma$  to the current stack, changing the state to  $q_1$ , and advancing to the next event. A closing rule  $q_0 \xrightarrow{\mathbf{cl} \ a: \gamma} q_1$  can be applied to a configuration that closes some  $a$ -node in state  $q_0$ . The symbol  $\gamma$  is then popped from the stack, the current state is changed to  $q_1$ , and the current event is advanced by one. It should be noticed that transitions on configurations are always deterministic.

There is exactly one initial configuration: its event is **start**, its state  $i$ , and its stack is empty. Furthermore, note that the current stack is always the sequence of symbols that were pushed to the stack by the ancestors of the current node and itself. A configuration is accepting if the current event is the closing event of the root, the current state is final, and the current stack is empty.

More formally, a run  $r$  of an dNWA  $A$  on a tree  $t$  is a pair of functions  $r_e : eve(t) \rightarrow Q$  and  $r_n : nod(t) \rightarrow \Gamma$ , such that  $r_e(\mathbf{start}) = i$  and that  $\delta$  contains the following rules for all  $\pi \in nod(t)$  with  $a = lab^t(\pi)$ ,  $\alpha \in \{\mathbf{op}, \mathbf{cl}\}$  and  $\eta = (\alpha, \pi)$ :

$$r_e(pr_{\preceq}(\eta)) \xrightarrow{\alpha \ a: r_n(\pi)} r_e(\eta)$$

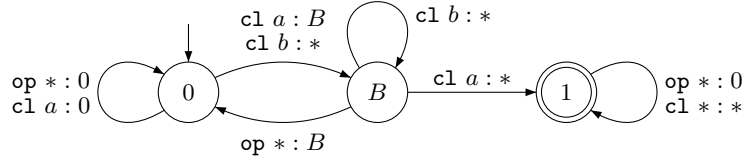
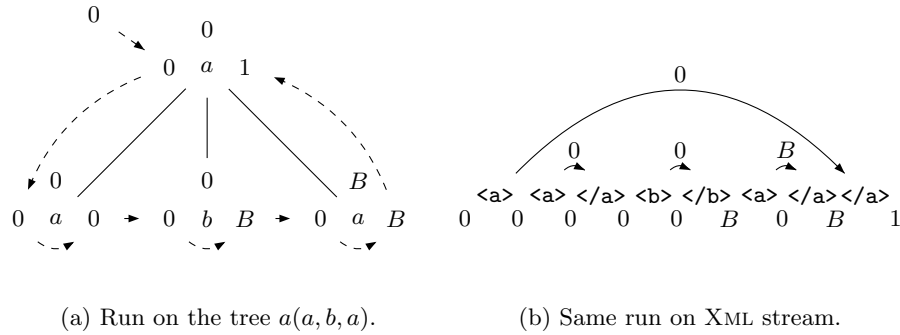


Fig. 4: A dNWA over  $\Sigma = \{a, b\}$  with  $Q = \{0, B, 1\}$  and  $\Gamma = \{0, B\}$ .



(a) Run on the tree  $a(a, b, a)$ .

(b) Same run on XML stream.

Fig. 5: Run of the dNWA of Fig. 4 on an input XML document.

A run  $r$  is successful if  $r_e((c1, \epsilon)) \in F$ . The recognized language  $L(A)$  is the set of trees on which  $A$  has a successful run. We call an dNWA *pseudo-complete* if there is a run on every tree  $t \in \mathcal{T}_\Sigma$ .

For illustration, consider the dNWA in Fig. 4, which recognizes all trees containing some  $a$ -node with some  $b$ -child. This Boolean query is  $ch^*(a(ch(b(true))))$  in FXP or  $[/a/b]$  in XPath syntax. We will freely use the symbol  $*$  to stand either for an arbitrary letter or an arbitrary stack symbol. The idea of this automaton is to move to state  $B$  when ever closing some  $b$ -node and to propagate this state by passing  $B$  to all closing events of following-siblings (except if some of them contains some  $a$ -descendant with some  $b$ -child, so that the automaton can safely go into the successful state 1). The automaton can move to the successful state 1 when closing some  $a$ -node from state  $B$ , since state  $B$  can only be assigned to closing events of children with a previous  $b$ -sibling. The run of this dNWA on tree  $a(a, b, a)$  is illustrated in Fig. 5. Stack symbols can be either annotated to nodes of trees or to edges from opening to corresponding closing events on XML streams. The horizontal propagation of  $B$  works as follows: at opening time  $B$  is pushed onto the stack and at closing time it is popped from there.

In order to compute the run of a dNWA  $A$  on an XML stream with tree  $t$ , the current configuration of  $A$  needs to be stored at each event of  $t$ . This configuration contains the state of the current event and the sequence of states annotated to the ancestors of the current node, i.e., the current stack. Note that the size of the stack is at most  $depth(t)$ , so that membership to  $L(A)$  can be decided by a streaming algorithm with a memory of size  $O(|A| + depth(t))$ .

Evaluation of dNWAs encoding DTDs or other XML schemas performs streaming schema validation. A weakness of naive evaluation for testing membership  $t \in L(A)$  is the laziness of  $A$  in streaming mode: it only detects  $a$ -nodes with  $b$ -children when closing the  $a$ -node, but could already do so when opening the  $b$ -child. For tree  $a(a, b, a)$  for instance, the earliest event is  $(\text{op}, \pi_2)$  when reading the first tag  $\langle \mathbf{b} \rangle$ . The streaming algorithm from [11] improves on this situation: it decides membership  $t \in L(A)$  for dNWAs  $A$  at the earliest possible event of tree  $t$  while remaining in PTIME. In order to find this earliest event, this algorithm needs to inspect the whole configuration at every event, not only the state.

Automata can also be used to define monadic queries. As before, we fix a variable  $x$ . For every tree  $t \in \mathcal{T}_\Sigma$  and node  $\pi \in \text{nod}(t)$ , we define the *canonical tree*  $t * \pi \in \mathcal{T}_{\Sigma \times 2^{\{x\}}}$  obtained from  $t$  by relabeling  $\pi$  with  $(\text{lab}^t(\pi), \{x\})$  and all other nodes  $\pi'$  with  $(\text{lab}^t(\pi'), \emptyset)$ . More generally, a tree  $t \in \mathcal{T}_{\Sigma \times 2^{\{x\}}}$  is *canonical* if exactly one of its nodes has a label in  $\Sigma \times \{x\}$ . A dNWA  $A$  with signature  $\Sigma \times 2^{\{x\}}$  defines the query  $\llbracket A \rrbracket$  on trees over  $\Sigma$  with  $\llbracket A \rrbracket(t) = \{\pi \in \text{nod}(t) \mid t * \pi \in L(A)\}$ .

## 4 FXP to Deterministic Automata

In this section, we propose a translation of  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge, \neg)$  to dNWAs. It runs in polynomial time if we assume a bound on the number of conjunctions. Our translation works by induction on the structure of formulas.

In order to avoid exponential blowups, our dNWAs will evaluate at most one subformula at every time point. Consider for instance the formula  $ch(F')$ . As all axes in  $F'$  are downwards (this would fail with the next-sibling axis), the algorithm can always know when closing a child, whether  $F'$  holds there or not. Thus, when opening the next child, the test for the previous child is finished. Therefore  $F'$  is tested for at most one child at a time. Note that an unbounded number of overlapping tests would end up in an exponential blowup. The same invariant also holds for  $o\text{-}ch_a^*(F')$  formulas: no nested  $a$ -descendants need to be tested simultaneously for  $F'$ ; considering outermost  $a$ -descendants is enough.

**Proposition 1.** *For every formula  $F$  of  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge, \neg)$ , we can build a dNWA  $A$  such that  $\llbracket A \rrbracket = \llbracket F \rrbracket$  in time  $O(|F|^{2 \cdot \text{width}(F)} \cdot |\Sigma|^{\text{width}(F)+1} \cdot 45^{\text{width}(F)})$ .*

The automaton construction is by induction on the structure of formulas. Here we only highlight the main trick necessary that makes the construction polynomial when fixing  $\text{width}(F)$ . Conjunctions are mapped to automata intersection and negations to automata complementation, by swapping final states while assuming pseudo-complete dNWAs. Determinism is essential here. Note

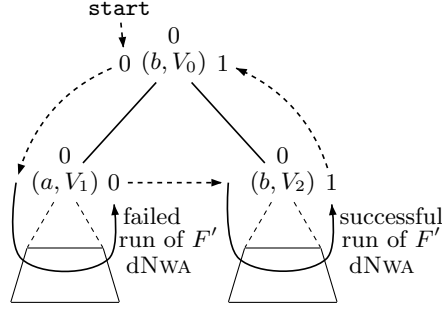


Fig. 6: Successful run of the dNWA recognizing  $F = ch(F')$ .

that the compilation of conjunctions might produce dNWA of size exponential in  $width(F)$ . The translations of label tests and variables is straightforward.

The main point, where we avoid an important blow up, appears already in the construction of the automaton for  $ch(F')$  and similarly for  $o-ch_a^*(F')$ . The idea of the dNWA for  $ch(F')$  is to run the dNWA  $A'$  testing  $F'$  on every child of the root until finding one that satisfies  $F'$ . When running on the subtree rooted by some child, the algorithm must know when the child will be closed. In order to do so, it must push a special symbol to the stack when opening the child. It could do so by pushing a tagged version of the stack symbol  $\gamma$  pushed by  $A'$ . However, this would double the number of node states at each  $ch$  operator (as we also have to use  $\gamma$  below), leading to a global size increase of  $2^n$  for formula  $ch^n(true)$ . The trick here, is to push a single new symbol 0, and to recompute node state  $\gamma$  corresponding to the current run due to determinism: knowing the initial state of  $A'$  and the label of the child, we can infer the rule of  $A'$  applied to open this child, and thus  $\gamma$ .

Let  $A' = (\Sigma \times 2^{\{x\}}, Q', \Gamma', i', F', \delta')$  be the automaton built for  $F'$ . Automaton  $A = (\Sigma \times 2^{\{x\}}, Q, \Gamma, i, F, \delta)$  for  $F$  will produce runs of the form in Fig. 8. It has three new states  $Q = Q' \uplus \{\mathbf{start}, 0, 1\}$  and one additional stack symbol  $\Gamma = \Gamma' \uplus \{0\}$ .

1. State **start** is only used as initial state, to open the root node:  $i = \{\mathbf{start}\}$  and a rule **start**  $\xrightarrow{\text{op } (a,V):0}$  0 is added to  $\delta$  for all possible  $(a, V) \in \Sigma \times 2^{\{x\}}$ .
2. State 0 is used when closing a child of the root, if no matching for  $F'$  has been found so far. When a child is opened from 0, we start testing  $F'$  and assign node state 0 to this child. We have to add new rules, from rules starting from the initial state of  $A'$  (note that stack symbol  $\gamma$  are lost):

$$\frac{q_1 \in i' \quad q_1 \xrightarrow{\text{op } (a,V):\gamma} q_2 \in \delta'}{0 \xrightarrow{\text{op } (a,V):0} q_2 \in \delta}$$

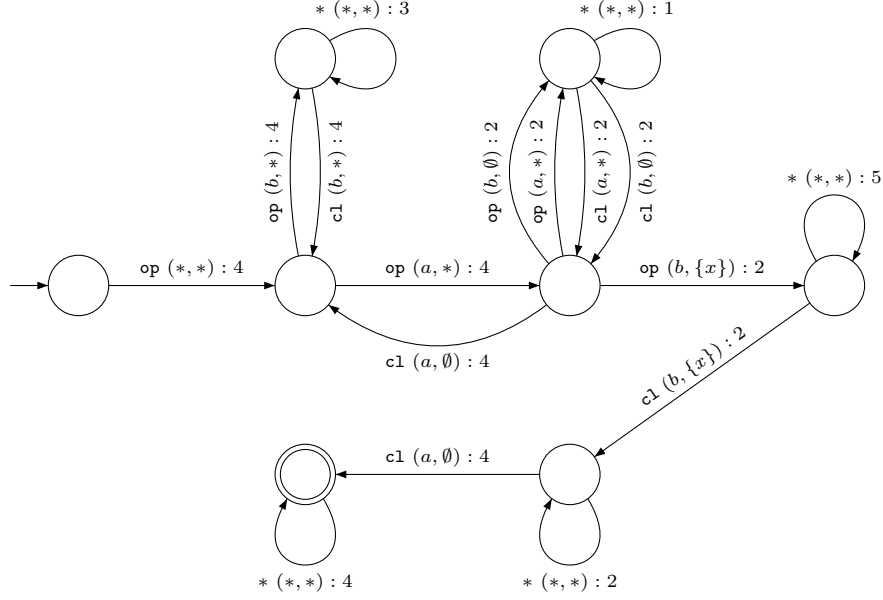


Fig. 7: dNWA constructed for  $ch(a(ch(b(x))))$  with  $\Sigma = \{a, b\}$ .

3. State 1 is universally accepting, so we always stay there once a matching has been found:  $1 \xrightarrow{\alpha(a,V):0} 1 \in \delta$  for all  $(\alpha, a, V) \in \{\text{op}, \text{c1}\} \times \Sigma \times 2^{\{x\}}$ , and  $F = \{1\}$ .
4. Then a test of  $F'$  is launched: the set of new rules  $\delta$  subsumes  $\delta'$ .
5. When closing a child of the root, we have to check whether the test of  $F'$  succeeded or not. As argued before,  $A$  pushes state 0 when oping a child, so that stack symbol  $\gamma$  pushed by  $A'$  is lost temporarily. But  $A$  can recompute this symbol when closing the child. In case of success,  $A$  closes in state 1, otherwise in state 0.

$$\frac{q'_1 \in i' \quad q'_1 \xrightarrow{\text{op}(a,V):\gamma} q'_2 \in \delta' \quad q_1 \xrightarrow{\text{c1}(a,V):\gamma} q_2 \in \delta' \quad q_2 \in F'}{q_1 \xrightarrow{\text{c1}(a,V):0} 1 \in \delta}$$

$$\frac{q'_1 \in i' \quad q'_1 \xrightarrow{\text{op}(a,V):\gamma} q'_2 \in \delta' \quad q_1 \xrightarrow{\text{c1}(a,V):\gamma} q_2 \in \delta' \quad q_2 \notin F'}{q_1 \xrightarrow{\text{c1}(a,V):0} 0 \in \delta}$$

6. Finally, to remain pseudo-complete, we have to propagate state 0 when closing the root node:  $0 \xrightarrow{\text{c1}(a,V):0} 0 \in \delta$  for all  $(a, V) \in \Sigma \times 2^{\{x\}}$ .

Even though the ideas of the constructions are rather simple, it should be noticed that dNWAs obtained by this construction are often hard to understand. This is mainly due to the recomputation trick. See Fig. 7 for an example.

## 5 Streamability of Query Languages

We present the notion of finite streamability of query languages, and apply it to the query languages defined by dNWA's and fragments of Forward XPath.

**Definition 2.** A monadic query language for unranked trees in  $\mathcal{T}_\Sigma$  is a triple  $(E, \llbracket \cdot \rrbracket, |\cdot|)$  that consists of a set  $E$  whose elements are called query definitions, a function from definitions  $e \in E$  to monadic query  $\llbracket e \rrbracket$ , that we call the query defined by  $e$ , and a mapping of query definitions  $e \in E$  to natural numbers  $|e| \in \mathbb{N}$ , that we call the size of  $e$ .

How many candidates must be buffered when answering a query  $\Phi$  on a tree  $t$ ? Intuitively, at least all alive candidates need to be stored, where a candidate  $\pi \in \text{nod}(t)$  is called *alive* at an event  $\eta \in \text{eve}(t)$  if it can be selected in some continuation of the stream and rejected in other ones. The concurrency  $\text{concur}_\Phi(t)$  of  $\Phi$  on  $t$  is the maximal number of alive candidates at all events.

The main idea of finite streamability is to require that the number of buffered candidates must be polynomially bounded in the concurrency. In order to do so, aliveness of some candidates must be decided at some point. Doing this in PTIME in the size of query definitions imposes a serious restriction, that all finitely streamable query languages must satisfy. In order to obtain lower bounds we assume that candidate sets are always stored without compression. This property is satisfied by all streaming algorithms in the literature.

**Definition 3.** We call a query language  $(E, \llbracket \cdot \rrbracket, |\cdot|)$  finitely streamable if there exists polynomials  $p_0, p_1, p_2$  such that for all query definitions  $e \in E$  one can compute in time  $p_0(|e|)$  a RAM machine  $\mathcal{M}_e$  computing  $\llbracket e \rrbracket$ , such that

- the space used by  $\mathcal{M}_e$  per step on  $t \in \mathcal{T}_\Sigma$  is at most  $p_1(|e|, \text{concur}_{\llbracket e \rrbracket}(t), \text{depth}(t))$  and at least  $\text{concur}_{\llbracket e \rrbracket}(t)$ , and
- the time used by  $\mathcal{M}_e$  per step on  $t \in \mathcal{T}_\Sigma$  is at most  $p_2(|e|, \text{concur}_{\llbracket e \rrbracket}(t), \text{depth}(t))$ .

Prior work on earliest query answering provides our first positive result on streamability for dNWA's.

**Theorem 1 ([11]).** The language of monadic queries defined by dNWA's over  $\Sigma \times 2^{\{x\}}$  is finitely streamable.

*Proof.* For monadic queries, the streaming algorithm in [11] has the following costs per step:  $O(c \cdot |A|^2)$  in time and  $O(c \cdot d \cdot |A|)$  in space, where  $c = \text{concur}_{\llbracket A \rrbracket}(t)$  and  $d = \text{depth}(t)$ . This algorithm requires the dNWA  $A$  to accept only canonical trees, which can be obtained by intersecting it with a dNWA checking canonicity (this can be done in polynomial time). A RAM machine implementing this algorithm can be built in PTIME.

We define the query language  $\text{FXP}(ch, o-ch_a^*, \wedge^{(k)}, \neg)$  which expressions are formulas  $F$  of  $\text{FXP}(ch, o-ch_a^*, \wedge, \neg)$  with less than  $k$  conjunctions, i.e. such that  $\text{width}(F) \leq k$ . For this fragment, the translation provided in Section 4 is in polynomial time, and thus avoids more general doubly exponential compilation schemas of XPath expressions into deterministic tree automata [7].

**Theorem 2.** *For every fixed  $k \geq 0$  and alphabet  $\Sigma$ ,  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge^{(k)}, \neg)$  is finitely streamable.*

*Proof.* Let  $k$  be fixed. For every formula  $F$  in  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge^{(k)}, \neg)$ ,  $\text{width}(F) \leq k$ , so, according to the translation proposed in Section 4 (Proposition 1), there exists a polynomial  $p$  such that for all formulas  $F$  of  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge^{(k)}, \neg)$  we can build in time  $O(p(|F|))$  a dNWA  $A$  such that  $\llbracket A \rrbracket = \llbracket F \rrbracket$ . Hence, finite streamability of queries by dNWA (Theorem 1) can be lifted to  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge^{(k)}, \neg)$ .

The restriction on the width of formulas is necessary to remain in PTIME.

**Theorem 3.**  *$\text{FXP}(ch, \wedge, \neg)$  is not finitely streamable, and remains non finitely streamable when restricted to non-recursive trees, unless  $P = NP$ .*

Here, we only give a brief sketch of the proof. We first show for all languages of descending queries that finite streamability implies that query satisfiability is in polynomial time. This can be shown by proving that aliveness of candidates must be decided for obtaining finite streamability, so that previous hardness results for earliest query answering carry over [6,11]. This works under the realistic assumption that the number of alive candidates is a space lower bound for streaming algorithms. We then show that satisfiability of  $\text{FXP}(ch, \wedge, \neg)$  is NP-hard by strengthening results from [4]. Hence, without assuming  $P=NP$  or a bound on the number of conjunctions,  $\text{FXP}(ch, \wedge, \neg)$  cannot be finitely streamable, nor any larger query language.

## 6 Related Work

Our compiler from  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge^{(k)}, \neg)$  must avoid the usual doubly exponential blow-up when translating XPath expressions into deterministic tree automata [7]. One exponential goes away by bounding the number of conjunctions and all kinds of overlapping tests, for instance when adding  $ns$  or  $ns^*$  steps. The other exponential is circumvented by the restriction to outermost descendants steps since these can be checked deterministically.

As proved in the current paper, finite streamability of  $\text{FXP}(ch, \wedge, \neg)$  continues to fail even if restricted to non-recursive documents. This shows that the memory consumption of the two algorithms of [2] and [12] cannot be polynomial in the number of alive candidates, in contrast to what is stated there<sup>4</sup> except if  $P=NP$ . We also note that streaming algorithms for Forward XPath in [22] and [23,24] do not claim finite streamability. The complexity results stated there count the maximal number of candidates stored simultaneously by their algorithms, rather than the maximal number of alive candidates with respect to the query.

Space lower bounds for multi-pass streaming algorithms were shown in [13]. Previous space lower bounds for one-pass streaming algorithms for XPath were obtained by communication complexity arguments without any assumptions on

<sup>4</sup> Authors of [2] and [12] have been notified. The journal version of [2] will take this remark into account.

compression tricks. Therefore, they remained limited to very specific fragments. In [2], wildcard-free queries in  $\text{FXP}(ch, ch^*, \wedge, \neg)$  are considered under the assumption of an infinite signature. It is shown that the maximal number of closed simultaneously alive answer candidates is a lower bound for “mostly all” non-recursive trees in the sense of instance complexity. In [25], it is shown that for some queries in  $\text{FXP}(ch, ch^*, \wedge)$  with independent  $ch$  predicates, the lower bound becomes  $n \cdot c$  where  $n$  is the length of the selecting branch of the XPath expression, and  $c$  is maximal number of concurrently alive candidates. This shows that even compression tricks do not help for these query languages.

In [10] it was shown that it is decidable in polynomial time for queries defined by deterministic nested word automata, whether the maximal number of concurrently alive candidates is bounded. This result can be lifted to  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge^{(k)}, \neg)$  by using our P-time compiler to dNWAs.

## 7 Conclusion

We have shown that  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge, \neg)$  becomes finite streamability when fixing the number of conjunctions. Without such a bound, even  $\text{FXP}(ch, \wedge, \neg)$  is not finitely streamable. Our results reveal some errors in previous work. This illustrates that they are nontrivial even though proofs are straightforward (once the translation is set up properly). It should also be noticed that our algorithm can be extended to support schemas (defined by DTDs or dNWAs) as well as for queries selecting tuples of nodes instead of nodes.

In QuiXProc (see [www.quixproc.com](http://www.quixproc.com)), a transfer project of INRIA and INNOVIMAX, we are currently working on highly efficient streaming algorithms for  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge, \neg)$  based on similar dNWA constructions, which enable early node selection (not necessarily always earliest). First tests with our implementation, whose source code is freely available at [fxp.lille.inria.fr](http://fxp.lille.inria.fr), confirm this expectation. We are working on improving the integration of these algorithms into XPROC to industrial quality. We are thus confident to prove the practical relevance of the methods presented here in the near future.

## References

1. R Alur and P Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009.
2. Z Bar-Yossef, M Fontoura, and V Josifovski. Buffering in query evaluation over XML streams. In *ACM PODS*, pages 216–227, 2005.
3. Z Bar-Yossef, M Fontoura, and V Josifovski. On the memory requirements of XPath evaluation over XML streams. *J. Comp. Syst. Sci.*, 73(3):391–441, 2007.
4. M Benedikt, W Fan, and F Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM*, 55(2):1–79, 2008.
5. M Benedikt and A Jeffrey. Efficient and expressive tree filters. In *FST-TCS*, pages 461–472, 2007.
6. M Benedikt, A Jeffrey, and R Ley-Wild. Stream Firewalling of XML Constraints. In *ACM SIGMOD*, pages 487–498, 2008.

7. N Francis, C David, and L Libkin. A direct translation from XPath to nondeterministic automata. In *5th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2011.
8. O Gauwin. *Streaming Tree Automata and XPath*. PhD thesis, Université Lille 1, 2009.
9. O Gauwin and J Niehren. Streamable fragments of Forward XPath. Long version. <http://hal.inria.fr/inria-00442250/en>. 2011.
10. O Gauwin, J Niehren, and S Tison. Bounded delay and concurrency for earliest query answering. In *LATA*, volume 5457 of *LNCS*, pages 350–361, 2009.
11. O Gauwin, J Niehren, and S Tison. Earliest query answering for deterministic nested word automata. In *FCT*, volume 5699 of *LNCS*, pages 121–132, 2009.
12. G Gou and R Chirkova. Efficient algorithms for evaluating XPath over streams. In *ACM SIGMOD*, pages 269–280, 2007.
13. M Grohe, C Koch, and N Schweikardt. Tight lower bounds for query processing on streaming and external memory data. In *ICALP, LNCS 3580*, 1076–1088, 2005.
14. M Kay. Saxon diaries, 2009. <http://saxonica.blogharbor.com/blog>.
15. M Kay. XSLT 2.1 – W3C working draft, May 2010.
16. L Libkin. Logics over unranked trees: an overview. *Logical Methods in Computer Science*, 3(2):1–31, 2006.
17. V Kumar, P Madhusudan, and M Viswanathan. Visibly pushdown automata for streaming XML. In *WWW*, pages 1053–1062, 2007.
18. C Koch, S Scherzinger, N Schweikardt, and B Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *30th VLDB*, pages 228–239. Morgan Kaufmann, 2004.
19. O Kupferman and M Y. Vardi. Model checking of safety properties. *Form. Meth. in Syst. Design*, 19(3):291–314, 2001.
20. A Neumann and H Seidl. Locating matches of tree patterns in forests. In *FSTTCS*, volume 1530 of *LNCS*, pages 134–145, 1998.
21. A Nizar and S Kumar. Efficient Evaluation of Forward XPath Axes over XML Streams. In *COMAD*, pages 222–233, 2008.
22. D Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. on Know. Data Eng.*, 19(7):934–949, 2007.
23. P Ramanan. Evaluating an XPath Query on a Streaming XML Document. In *COMAD*, pages 41–52, 2005.
24. P Ramanan. Worst-case optimal algorithm for XPath evaluation over XML streams. *J. of Comp. Syst. Sci.*, 75:465–485, 2009.
25. P Ramanan. Memory lower bounds for XPath evaluation over XML streams. *J. of Comp. Syst. Sci.*, In Press, Corrected Proof:–, 2010.
26. N Schweikardt. Machine models and lower bounds for query processing. In *ACM PODS*, pages 41–52, 2007.
27. L Segoufin and C Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *ICDT*, pages 299–313, 2007.
28. L Segoufin and V Vianu. Validating streaming XML documents. In *ACM PODS*, pages 53–64, 2002.
29. X Wu and D Theodoratos. Evaluating Partial Tree-Pattern Queries on XML Streams. In *CIKM*, pages 1409–1410, 2008.

## References

1. Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *Journal of the ACM*, 56(3):1–43, 2009.

2. Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. Buffering in query evaluation over XML streams. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 216–227. ACM-Press, 2005.
3. Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the memory requirements of XPath evaluation over XML streams. *Journal of Computer and System Science*, 73(3):391–441, 2007.
4. Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM*, 55(2):1–79, 2008.
5. Michael Benedikt and Alan Jeffrey. Efficient and expressive tree filters. In *Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 461–472. Springer Verlag, 2007.
6. Michael Benedikt, Alan Jeffrey, and Ruy Ley-Wild. Stream Firewalling of XML Constraints. In *ACM SIGMOD International Conference on Management of Data*, pages 487–498. ACM-Press, 2008.
7. Nadime Francis, Claire David, and Leonid Libkin. A direct translation from XPath to nondeterministic automata. In *5th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2011.
8. Olivier Gauwin. *Streaming Tree Automata and XPath*. PhD thesis, Université Lille 1, 2009.
9. Olivier Gauwin and Joachim Niehren. Streamable fragments of Forward XPath (long version). 2011. <http://www.grappa.univ-lille3.fr/~niehren/Papers/streamability/0.pdf>.
10. Olivier Gauwin, Joachim Niehren, and Sophie Tison. Bounded delay and concurrency for earliest query answering. In *3rd International Conference on Language and Automata Theory and Applications*, volume 5457 of *Lecture Notes in Computer Science*, pages 350–361. Springer Verlag, 2009.
11. Olivier Gauwin, Joachim Niehren, and Sophie Tison. Earliest query answering for deterministic nested word automata. In *17th International Symposium on Fundamentals of Computer Theory*, volume 5699 of *Lecture Notes in Computer Science*, pages 121–132. Springer Verlag, 2009.
12. Gang Gou and Rada Chirkova. Efficient algorithms for evaluating XPath over streams. In *36th ACM SIGMOD International Conference on Management of Data*, pages 269–280. ACM-Press, 2007.
13. Martin Grohe, Christoph Koch, and Nicole Schweikardt. Tight lower bounds for query processing on streaming and external memory data. In *ICALP'05: 32nd International Colloquium on Automata, Languages and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 1076–1088. Springer Verlag, 2005.
14. Michael Kay. Saxon diaries, 2009.
15. Michael Kay. XSLT 2.1 – W3C working draft, May 2010.
16. Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *30th International Conference on Very Large Data Bases*, pages 228–239. Morgan Kaufmann, 2004.
17. Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Visibly pushdown automata for streaming XML. In *16th international conference on World Wide Web*, pages 1053–1062. ACM-Press, 2007.
18. Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
19. Leonid Libkin. Logics over unranked trees: an overview. *Logical Methods in Computer Science*, 3(2):1–31, 2006.

20. Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 134–145. Springer Verlag, 1998.
21. Abdul Nizar and Sreenivasa Kumar. Efficient Evaluation of Forward XPath Axes over XML Streams. In *14th International Conference on Management of Data (COMAD)*, pages 222–233, 2008.
22. Dan Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. on Know. Data Eng.*, 19(7):934–949, 2007.
23. Prakash Ramanan. Evaluating an XPath Query on a Streaming XML Document. In *12th International Conference on Management of Data (COMAD)*, pages 41–52, 2005.
24. Prakash Ramanan. Worst-case optimal algorithm for XPath evaluation over XML streams. *Journal of Computer and System Science*, 75:465–485, 2009.
25. Prakash Ramanan. Memory lower bounds for XPath evaluation over XML streams. *Journal of Computer and System Sciences*, In Press, Corrected Proof, 2010.
26. Nicole Schweikardt. Machine models and lower bounds for query processing. In *Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 41–52. ACM-Press, 2007.
27. Luc Segoufin and Cristina Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *Database Theory - ICDT 2007, 11th International Conference*, pages 299–313, 2007.
28. Luc Segoufin and Victor Vianu. Validating streaming XML documents. In *Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 53–64, 2002.
29. Xiaoying Wu and Dimitri Theodoratos. Evaluating Partial Tree-Pattern Queries on XML Streams. In *17th ACM International Conference on Information and Knowledge Management (CIKM'08)*, pages 1409–1410. ACM Press, 2008.

## A FXP to Deterministic Automata

We assume that  $|\Sigma| \geq 2$ .

**Proposition 1.** *For every formula  $F$  of  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge, \neg)$ , we can build a dNWA  $A$  such that  $\llbracket A \rrbracket = \llbracket F \rrbracket$  in time  $O(|F|^{2 \cdot \text{width}(F)} \cdot |\Sigma|^{\text{width}(F)+1} \cdot 45^{\text{width}(F)})$ .*

*Proof.* We start with the compiler which is by induction on the structure of expressions, and then analyse its complexity in a second step.

We extend annotations, in order to deal with non-canonical ones. For a tree  $t \in \mathcal{T}_\Sigma$  and a function  $\nu: \text{nod}(t) \rightarrow 2^{\{x\}}$ , let  $t\tilde{*}\nu$  be the tree with  $\text{nod}(t\tilde{*}\nu) = \text{nod}(t)$  and for all nodes  $\pi \in \text{nod}(t)$ ,  $\text{lab}^{t\tilde{*}\nu}(\pi) = (\text{lab}^t(\pi), \nu(\pi))$ . The semantics of formulas is adapted in the natural way, by changing the semantics of variable  $x$ :  $\llbracket x \rrbracket_{t,\nu} = \{\pi \in \text{nod}(t) \mid x \in \nu(\pi)\}$ . The invariant verified by the translation is that for every tree  $t \in \mathcal{T}_\Sigma$  and every annotation  $\nu: \text{nod}(t) \rightarrow 2^{\{x\}}$ :  $t\tilde{*}\nu \in L(A) \iff \epsilon \in \llbracket F \rrbracket_{t,\nu}$ . This implies  $\llbracket A \rrbracket = \llbracket F \rrbracket$ .

**Case  $F = F_1 \wedge F_2$**  Let  $A_1$  (resp.  $A_2$ ) be the pseudo-complete dNWA for  $F_1$  (resp.  $F_2$ ) and  $A$  be the synchronized product of  $A_1$  and  $A_2$ . Determinism and pseudo-completeness are preserved, and  $A$  recognizes the correct tree language, as for all trees  $t \in \mathcal{T}_\Sigma$  and  $\nu: \text{nod}(t) \rightarrow 2^{\{x\}}$ :

$$t\tilde{*}\nu \in L(A) \iff t\tilde{*}\nu \in L(A_1) \wedge t\tilde{*}\nu \in L(A_2) \stackrel{\text{ind. hyp.}}{\iff} \epsilon \in \llbracket F_1 \rrbracket_{t,\nu} \wedge \epsilon \in \llbracket F_2 \rrbracket_{t,\nu} \iff \epsilon \in \llbracket F \rrbracket_{t,\nu}$$

**Case  $F = \neg F'$**  Let  $A'$  be the pseudo-complete dNWA built for  $F'$ . Let  $A$  be the NWA obtained from  $A'$  by swapping the final states, i.e.  $F^A = Q^{A'} - F^{A'}$ .  $A'$  is deterministic and pseudo-complete, so we get:

$$t\tilde{*}\nu \in L(A) \iff t\tilde{*}\nu \notin L(A') \stackrel{\text{ind. hyp.}}{\iff} \epsilon \notin \llbracket F' \rrbracket_{t,\nu} \iff \epsilon \in \llbracket F \rrbracket_{t,\nu}$$

**Case  $F = \text{true}$**  As  $\llbracket \text{true} \rrbracket_{t,\nu} = \text{nod}(t)$ , a universal dNWA  $A$  suffices, i.e.  $L(A) = \mathcal{T}_{\Sigma \times 2^{\{x\}}}$ .

**Case  $F = ch(F')$**  Let  $A'$  be the automaton built for  $F'$ . The automaton  $A$  for  $F$  has to launch  $A'$  when opening each child of the root (see Fig. 8). Here we need three additional event states  $Q^A = Q^{A'} \uplus \{\text{start}, 0, 1\}$  and one additional node state  $\Gamma^A = \Gamma^{A'} \uplus \{0\}$ .

1. State **start** is only used as initial state, to open the root node:  $i^A = \{\text{start}\}$  and a rule  $\text{start} \xrightarrow{\text{op}(a,V):0} 0$  is added to  $\delta^A$  for every possible  $(a, V) \in \Sigma \times 2^{\{x\}}$ .
2. State 0 is used when closing a child of the root, if no matching for  $F'$  has been found so far. When a child is opened from 0, we start testing  $F'$  and assign node state 0 to this child. We have to add new rules, from rules starting from the initial state of  $A'$  (note that node states  $\gamma$  are lost):

$$\frac{q_1 \in i^{A'} \quad q_1 \xrightarrow{\text{op}(a,V):\gamma} q_2 \in \delta^{A'}}{0 \xrightarrow{\text{op}(a,V):0} q_2 \in \delta^A}$$

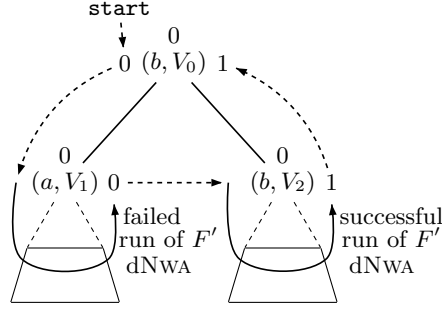


Fig. 8: Successful run of the dNWA recognizing  $F = ch(F')$ .

3. State 1 is a universal accepting state, where we stay once a matching has been found:  $1 \xrightarrow{\alpha (a,V):0} 1 \in \delta^A$  for every  $(\alpha, a, V) \in \{\text{op}, \text{cl}\} \times \Sigma \times 2^{\{x\}}$ , and  $F^A = \{1\}$ .
4. Then a test of  $F'$  is launched: rules of  $A'$  are also rules of  $A$ :  $\delta^{A'} \subseteq \delta^A$ .
5. When closing a child of the root, we have to check whether the test of  $F'$  succeeded or not. *Pseudo-completeness* is essential, so that failure does not block the run, and is equivalent to ending in a non-final state of  $A'$ . We have to consider which rule of  $A'$  is applied when closing this child: its node state  $\gamma$  has *not* been assigned as node state at opening, as we had to mark children of the root with node state 0 (storing  $(\gamma, 0)$  would imply a blow-up). *Hopefully, in a dNWA, only one rule can be applied when opening a root of a given label, so we know exactly which rule of  $A'$  has been applied when opening this child, and thus  $\gamma$  can be retrieved.* In case of success, we close in state 1; otherwise, we close in state 0.

$$\frac{q'_1 \in i^{A'} \quad q'_1 \xrightarrow{\text{op} (a,V):\gamma} q'_2 \in \delta^{A'} \quad q_1 \xrightarrow{\text{cl} (a,V):\gamma} q_2 \in \delta^{A'} \quad q_2 \in F^{A'}}{q_1 \xrightarrow{\text{cl} (a,V): 0} 1 \in \delta^A}$$

$$\frac{q'_1 \in i^{A'} \quad q'_1 \xrightarrow{\text{op} (a,V):\gamma} q'_2 \in \delta^{A'} \quad q_1 \xrightarrow{\text{cl} (a,V):\gamma} q_2 \in \delta^{A'} \quad q_2 \notin F^{A'}}{q_1 \xrightarrow{\text{cl} (a,V): 0} 0 \in \delta^A}$$

6. Finally, to remain pseudo-complete, we have to propagate state 0 when closing the root node:  $0 \xrightarrow{\text{cl} (a,V):0} 0 \in \delta^A$  for every  $(a, V) \in \Sigma \times 2^{\{x\}}$ .

$A$  is deterministic. The fact that all axes in  $\mathcal{D}$  are downwards permits to decide, when closing a child, whether this child matches  $F'$ . By a left-to-right induction on the children of the root of  $t * \nu$ , we can prove that the run  $r$  of  $A$  on  $t * \nu$  assigns 1 to  $(\text{cl}, i)$  if there is an accepting run of  $A'$  on a child  $j$  (with  $1 \leq j \leq i$ ) of  $\epsilon$ , and 0 otherwise. As this Boolean is kept when closing the root, and is set to 0 if there is no child, we have, for  $t = a(t_1, \dots, t_k)$  and

for  $\nu_i$  the restriction of  $\nu$  to nodes of  $t_i$ :

$$t * \nu \in L(A) \iff \exists 1 \leq i \leq k, t_i * \nu_i \in L(A') \stackrel{\text{ind. h.}}{\iff} \exists 1 \leq i \leq k, \epsilon \in \llbracket F' \rrbracket_{t_i, \nu_i} \iff \epsilon \in \llbracket F \rrbracket_{t, \nu}$$

**Case  $F = o\text{-}ch_a^*(F')$**  Let  $A'$  be the pseudo-complete dNWA constructed for  $F'$ .

The definition of the pseudo-complete dNWA  $A$  for  $F$  is very similar to the case where  $F = ch(F')$ , and illustrated in Fig. 9. The only difference is the way state 0 is used, i.e. Step 2 of the translation of  $F = ch(F')$ . Instead of using event state 0 between each child of the root, we use it to *wait for an  $a$ -node*, by adding rules  $0 \xrightarrow{\alpha (b,V):0} 0 \in \delta^A$  for every  $b \neq a$ , and every  $(\alpha, V) \in \{\text{op}, \text{cl}\} \times 2^{\{x\}}$ . When an  $a$ -node is found, we start testing  $F'$  using the following rules:

$$\frac{q_1 \in i^{A'} \quad q_1 \xrightarrow{\text{op } (a,V):\gamma} q_2 \in \delta^{A'}}{0 \xrightarrow{\text{op } (a,V):0} q_2 \in \delta^A}$$

At every time point there is at most one  $a$ -node to be considered, according to the outermost semantics, and the fact that  $F'$  is depends only on the subtree of the node to be tested. Now one can show that there exists a

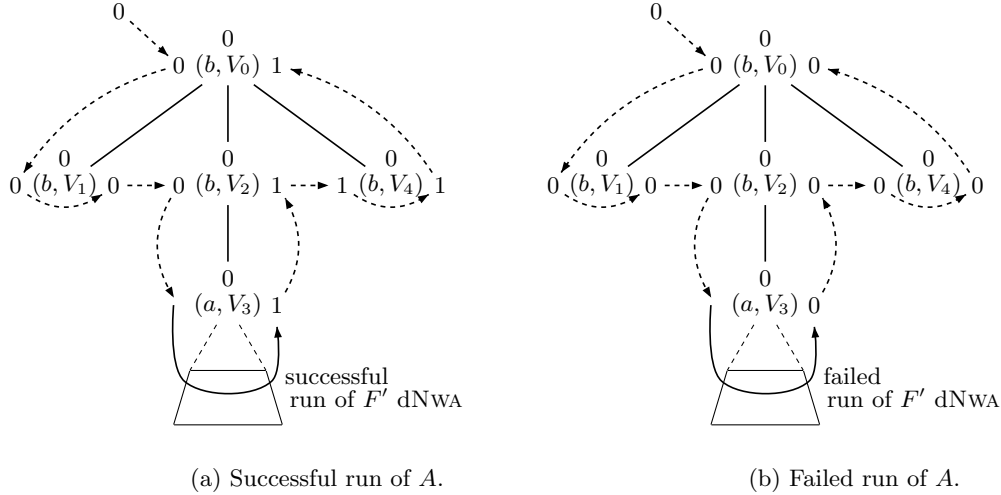


Fig. 9: Example runs of  $A$  recognizing  $F = o\text{-}ch_a^*(F')$ .

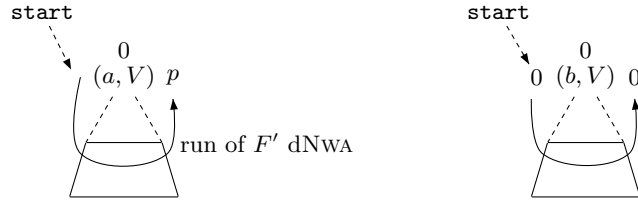
successful run of  $A$  on some tree  $t * \nu$  iff  $\epsilon \in \llbracket F \rrbracket_{t, \nu}$ . No match of  $a$  can be missed, since no node above  $a$  is labeled by  $a$  (outermost semantics). The only reason to move into a state different from 0 before opening the  $a$ -node is another  $a$ -node on the left. Either the run of  $F'$  there succeeds, and the automaton goes into the universal state 1, or else, it finishes but fails ( $A'$

being pseudo-complete), and returns back into state 0, so that new  $a$ -nodes can be tested. Determinism and pseudo-completeness are preserved.

**Case  $F = a(F')$**  Let  $A'$  be the automaton built for  $F'$ . We can build  $A$  from  $A'$  by adding two event states and one node state:  $Q^A = Q^{A'} \uplus \{\mathbf{start}, 0\}$  with  $i^A = \{\mathbf{start}\}$  and  $F^A = F^{A'}$ , and  $\Gamma^A = \Gamma^{A'} \uplus \{0\}$ .

1. Event state 0 is a sink:  $0 \xrightarrow{\alpha (b,V):0} 0 \in \delta^A$  for every  $(\alpha, b, V) \in \{\mathbf{op}, \mathbf{cl}\} \times \Sigma \times 2^{\{x\}}$ .
2. If the root is not labeled by  $a$ ,  $A$  goes to the sink state 0: for every  $b \neq a$  and every  $V \subseteq \{x\}$ ,  $\mathbf{start} \xrightarrow{\mathbf{op} (b,V):0} 0 \in \delta^A$ .
3. If the root is labeled by  $a$ ,  $A$  performs the run of  $A'$  until the end. Hence  $\delta^{A'} \subseteq \delta^A$ , and this run is launched using one of these rules:

$$\frac{q_1 \in i^{A'} \quad q_1 \xrightarrow{\mathbf{op} (a,V):\gamma} q_2 \in \delta^{A'}}{\mathbf{start} \xrightarrow{\mathbf{op} (a,V):\gamma} q_2 \in \delta^A}$$



(a) Run of  $A$  on  $a$ -rooted tree. (b) Run of  $A$  on  $b$ -rooted tree ( $b \neq a$ ).

Fig. 10: Example runs of the dNWA  $A$  recognizing  $F = a(F')$ .

**Case  $F = x$**  Suppose that the root of the tree  $t\tilde{x}\nu$  is labeled by  $(a, V)$ . Then the automaton  $A$  only needs to check that  $x \in V$ , as performed by the dNWA in Fig. 11a. When opening the root, it goes to event state 0 and associates node state 0 to the root. Below the root, it stays in event state 0 and uses node state 1. When closing the root, it stays in 0 if  $x \in V$ , and to 1 otherwise.  $A$  is deterministic and pseudo-complete, and the correctness is immediate, as node state 0 can only be used at the root node.

We next analyse complexity of our compiler. We first show that this translation verifies the following property:

There exists  $c > 0$  such that for every formula  $F$  of  $\text{FXP}(ch, o-ch_a^*, \wedge, \neg)$ , a pseudo-complete dNWA  $A$  over signature  $\Sigma \times 2^{\{x\}}$  with at most  $(3 \cdot |F|)^{\text{width}(F)}$  event and node states can be computed in time at most

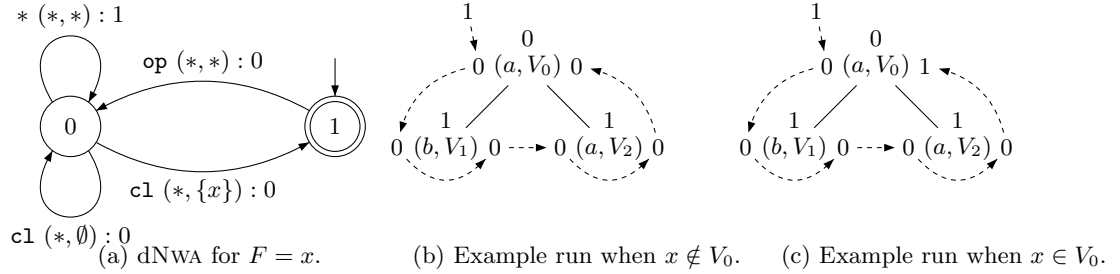


Fig. 11: The dNWA  $A$  recognizing  $F = x$ , and some example runs.

$c \cdot (|\delta^A| \cdot (5 \cdot |\Sigma|)^{width(F)} + |F|)$  such that for every tree  $t \in \mathcal{T}_\Sigma$  and  $\nu: nod(t) \rightarrow 2^{\{x\}}$ :

$$t\tilde{*}\nu \in L(A) \quad \text{iff} \quad \epsilon \in \llbracket F \rrbracket_{t,\nu}$$

For  $\alpha \in \{\text{op}, \text{cl}\}$  and  $a \in \Sigma$ , we write  $\delta_{\alpha,a}^A = \{q_1 \xrightarrow{\alpha \ a:\gamma} q_2 \in \delta^A\}$ .

At each inductive step different from  $\wedge$ , at most three event states are added. When  $F_1 \wedge F_2$  is translated, the number of event states is the product of the number of event states for  $F_1$  and the number of event states for  $F_2$ . Hence, the number of event states for translating  $F$  is bounded by  $(3 \cdot |F|)^{width(F)}$ .

For inductive steps different from  $\wedge$ , the translation can be performed in time bounded by  $c_2 \cdot |\delta^A| + c_3 \cdot |\Sigma|$  for some constants  $c_2$  and  $c_3$ . Indeed, rules can be yielded by considering each rule of the subformula  $F'$  once, and adding a number of rules depending only on  $|\Sigma|$ . Translation of  $F = \neg F'$  only requires constant time (in addition to the time required for translating  $F'$ ): we use a Boolean indicating whether final states have to be interpreted as their complement. The only exception is for axes steps  $F = ch(F')$  and  $F = o\text{-}ch_a^*(F')$ , where one rule for  $F'$  may yield several rules for  $F$ . However, each rule can be yielded in constant time, if the set of rules starting from an initial state can be computed in constant time, which is a reasonable assumption.

The time needed for computing a dNWA  $A$  for  $F_1 \wedge F_2$  is greater. Assume that for every pair of pseudo-complete dNWAs  $(A_1, A_2)$ , a pseudo-complete dNWA for  $A_1 \cap A_2$  with  $|Q^{A_1}| \cdot |Q^{A_2}|$  event states,  $|I^{A_1}| \cdot |I^{A_2}|$  node states and such that  $|\delta^A| = \sum_{\alpha \in \{\text{op}, \text{cl}\}, a \in \Sigma} |\delta_{\alpha,a}^{A_1}| \cdot |\delta_{\alpha,a}^{A_2}|$  can be computed in time  $c_1 \cdot |\delta^A|$ . This is also a reasonable assumption on the data structure for dNWAs.

Building  $A$  consists in building  $A_1$  and  $A_2$  for  $F_1$  and  $F_2$  (which can be done in time  $c \cdot (|\delta^{A_1}| \cdot (5 \cdot |\Sigma|)^{width(F_1)} + |F_1|) + c \cdot (|\delta^{A_2}| \cdot (5 \cdot |\Sigma|)^{width(F_2)} + |F_2|)$  by induction hypothesis) and then  $A$  from these two dNWAs, which can be done in time  $c_1 \cdot |\delta^A|$ . Hence the total time for building  $A$  is:

$$c \cdot (|\delta^{A_1}| \cdot (5 \cdot |\Sigma|)^{width(F_1)} + |F_1|) + c \cdot (|\delta^{A_2}| \cdot (5 \cdot |\Sigma|)^{width(F_2)} + |F_2|) + c_1 \cdot |\delta^A| \\ = \Theta + c \cdot (|F_1| + |F_2|) + c_1 \cdot |\delta^A|$$

with

$$\begin{aligned}
\Theta &= c \cdot (|\delta^{A_1}| \cdot (5 \cdot |\Sigma|)^{\text{width}(F_1)}) + c \cdot (|\delta^{A_2}| \cdot (5 \cdot |\Sigma|)^{\text{width}(F_2)}) \\
&\leq c \cdot (|\delta^{A_1}| + |\delta^{A_2}|) \cdot (5 \cdot |\Sigma|)^{\text{width}(F)-1} \quad \text{as } \text{width}(F)-1 \geq \text{width}(F_i) \\
&\leq c \cdot |\delta^A| \cdot \frac{|\delta^{A_1}| + |\delta^{A_2}|}{|\delta^A|} \cdot (5 \cdot |\Sigma|)^{\text{width}(F)-1} \\
&\leq c \cdot |\delta^A| \cdot 4 \cdot |\Sigma| \cdot (5 \cdot |\Sigma|)^{\text{width}(F)-1} \quad \text{cf below}
\end{aligned}$$

For the last inequality, we have that  $|\delta^A| = \sum_{\alpha \in \{\text{op}, \text{cl}\}, a \in \Sigma} |\delta_{\alpha, a}^{A_1}| \cdot |\delta_{\alpha, a}^{A_2}|$ . By grouping by actions and letters, we get:

$$\begin{aligned}
\frac{|\delta^{A_1}| + |\delta^{A_2}|}{|\delta^A|} &= \frac{|\delta^{A_1}| + |\delta^{A_2}|}{\sum_{\alpha \in \{\text{op}, \text{cl}\}, a \in \Sigma} |\delta_{\alpha, a}^{A_1}| \cdot |\delta_{\alpha, a}^{A_2}|} = \frac{\sum_{\alpha \in \{\text{op}, \text{cl}\}, a \in \Sigma} |\delta_{\alpha, a}^{A_1}| + |\delta_{\alpha, a}^{A_2}|}{\sum_{\alpha \in \{\text{op}, \text{cl}\}, a \in \Sigma} |\delta_{\alpha, a}^{A_1}| \cdot |\delta_{\alpha, a}^{A_2}|} \\
&\leq \sum_{\alpha \in \{\text{op}, \text{cl}\}, a \in \Sigma} \frac{|\delta_{\alpha, a}^{A_1}| + |\delta_{\alpha, a}^{A_2}|}{|\delta_{\alpha, a}^{A_1}| \cdot |\delta_{\alpha, a}^{A_2}|} \leq 4 \cdot |\Sigma|
\end{aligned}$$

Note that  $|\delta_{\alpha, a}^{A_1}| > 0$  and  $|\delta_{\alpha, a}^{A_2}| > 0$  (easily checked at every step). We assume wlog that  $\frac{c_1}{c} \leq 2$ , and that  $|\Sigma| \geq 2$ . Finally, the total time for computing  $A$  is:

$$\begin{aligned}
&\Theta + c \cdot (|F_1| + |F_2|) + c_1 \cdot |\delta^A| \\
&\leq \Theta + c \cdot |F| + c_1 \cdot |\delta^A| \quad \text{as } |F| = |F_1| + |F_2| + 1 \\
&\leq c \cdot |\delta^A| \cdot 4 \cdot |\Sigma| \cdot (5 \cdot |\Sigma|)^{\text{width}(F)-1} + c \cdot |F| + c_1 \cdot |\delta^A| \\
&\leq c \cdot (|\delta^A| \cdot ((5 \cdot |\Sigma|)^{\text{width}(F)-1} \cdot (4 \cdot |\Sigma|) + \frac{c_1}{c}) + |F|) \\
&\leq c \cdot (|\delta^A| \cdot ((5 \cdot |\Sigma|)^{\text{width}(F)-1} \cdot (4 \cdot |\Sigma| + \frac{c_1}{c})) + |F|) \\
&\leq c \cdot (|\delta^A| \cdot ((5 \cdot |\Sigma|)^{\text{width}(F)-1} \cdot 5 \cdot |\Sigma|) + |F|) \quad \text{as } \frac{c_1}{c} \leq 2 \leq |\Sigma| \\
&\leq c \cdot (|\delta^A| \cdot (5 \cdot |\Sigma|)^{\text{width}(F)} + |F|)
\end{aligned}$$

Let  $A$  be the pseudo-complete dNWA obtained for  $F$  in time  $c \cdot (|\delta^A| \cdot (5 \cdot |\Sigma|)^{\text{width}(F)} + |F|)$ . We have shown that  $|Q^A| \leq (3 \cdot |F|)^{\text{width}(F)}$  and  $|\Gamma^A| \leq (3 \cdot |F|)^{\text{width}(F)}$ . As  $A$  is a deterministic NWA over alphabet  $\Sigma \times 2^{\{x\}}$ ,  $|\delta^A|$  is in  $O(|Q^A| \cdot |\Gamma^A| \cdot |\Sigma|)$ , so  $A$  can be computed in time  $O((3 \cdot |F|)^{2 \cdot \text{width}(F)} \cdot 5^{\text{width}(F)} \cdot |\Sigma|^{\text{width}(F)+1})$ .

## B Hardness

We present hardness results for finite streamability of languages of descending queries. We show that finite streamability may require to decide aliveness of candidates for some query languages, where the concurrency is not bounded. The consequences are considerable, since deciding aliveness is hard, even for small query languages, including numerous fragments of XPath.

We now characterize the streamability of query languages  $\mathcal{Q}$ . The following theorem states that being finitely streamable (while verifying two other properties) implies that the satisfiability of descending Boolean queries defined from  $\mathcal{Q}$  is in PTIME. This can be used to prove that some query language is not finitely streamable. The theorem relies on the translation of a query  $\Phi$  to another one  $\text{exists}(\Phi)$ , also monadic, but with higher concurrency (see Appendix C).

For queries  $\Phi$  of arbitrary arity we define a tree language  $L_\Phi = \{t \in \mathcal{T}_\Sigma \mid \Phi(t) \neq \emptyset\}$ .

**Proposition 2 (Reduction to Satisfiability).** *Let  $\mathcal{Q} = (E, \llbracket \cdot \rrbracket, |\cdot|)$  be a language of monadic queries and  $r, p_0, p_1, p_2$  polynomials such that:*

1. *queries  $exists(\llbracket e \rrbracket)$ , with  $e \in E$ , are definable by expressions in  $E$  of size  $r(|e|)$  in time  $O(r(|e|))$ ;*
2.  *$\mathcal{Q}$  is finitely streamable with polynomials  $p_0, p_1, p_2$ .*

*Then the satisfiability  $L_{\llbracket e \rrbracket} \neq \emptyset$  for definitions  $e \in E$  of descending queries can be solved in time  $O(p_0(r(|e|)) + r(|e|) + p_1(r(|e|), 0, 2) \cdot p_2(r(|e|), 0, 2))$ .*

Satisfiability remains infeasible for  $\text{FXP}(ch, \wedge, \neg)$  even if we restrict ourselves to non-recursive trees, so that this query language is not finitely streamable.

**Proposition 3.** *Satisfiability of queries in  $\text{FXP}(ch, \wedge, \neg)$  on non-recursive trees of depth at most 2 is NP-hard.*

**Theorem 3**  $\text{FXP}(ch, \wedge, \neg)$  is *not* finitely streamable, and remains non finitely streamable when restricted to non-recursive trees, unless  $P = \text{NP}$ .

*Proof.*  $\text{FXP}(ch, \wedge, \neg)$  defines descending queries, and permits to define the operator *exists* in linear time. Satisfiability of  $\text{FXP}(ch, \wedge, \neg)$  is PSPACE-hard [4], so this fragment cannot be finitely streamable. When restricted to non-recursive trees, satisfiability remains NP-hard as shown in Proposition 3.

This result has some surprising consequences on streaming algorithms in the literature. The algorithms LQ (Lazy Querying) and EQ (Eager Querying) proposed in [12] answer  $\text{FXP}(ch, ch^*, \wedge, \neg)$  queries. Both can be implemented by algorithms constructed in PTIME, using per-event time in  $O(|e|)$ , per-event space (not taking candidates into account) in  $O(|e| \cdot \text{depth}(t))$ , and the number of stored candidates equals the concurrency<sup>5</sup>. This is in contradiction with Theorem 3, as it would imply finite streamability of  $\text{FXP}(ch, \wedge, \neg)$ .<sup>6</sup>

The streaming algorithm in [2] answers queries on non-recursive trees in some superset of  $\text{FXP}(ch, ch^*, \wedge, \neg)$ . The complexity of this algorithm is similar to algorithms LQ and EQ: per-event time cost is in  $O(|e|)$ , while per-event space is in  $O(|e| \cdot (\log(|e|) + \log(|t|)))$  plus  $O(\text{concur}_{\llbracket e \rrbracket}(t))$  for storing the candidates. Here,  $\log(|t|)$  bits are used to store one node, and at most one candidate node  $\pi$  of  $t$  is stored for each symbol of  $e$ . Hence, this algorithm can be implemented by a RAM machine built in PTIME from  $e$ , using per-event space and time polynomial in  $|e|$  and  $\text{concur}_{\llbracket e \rrbracket}(t)$ . Once more, this contradicts Theorem 3.<sup>7</sup>

An algorithm for another extension of  $\text{FXP}(ch, ch^*, \wedge, \neg)$  on recursive document was proposed in [23,24]. This algorithm uses less than  $O((\text{depth}(t) + c) \cdot |e|)$  space and  $O(|e|^2 + \text{depth}(t))$  per-event time, where  $c$  is the maximal number  $c$  of candidates stored simultaneously. This amount of candidates is not precisely analyzed. We can assert, using Theorem 3, that  $c$  can not be polynomially bounded in  $\text{concur}_{\llbracket e \rrbracket}(t)$ .

<sup>5</sup> Optimal buffering of candidates is asserted, e.g., in Theorem 3.

<sup>6</sup> A flaw in this algorithm was already noticed in [24].

<sup>7</sup> These problems will be fixed in the journal version of [2].

The memory costs of all these algorithms exceed the concurrency of the query on some trees, as they do not check for satisfiability of the computed matches wrt possible continuations. They only check whether all predicates of these matches are satisfiable individually, but do not test whether they can be satisfied simultaneously. As mentioned in the conclusion of [24], adding such a feature is not straightforward.

## C Proofs of Hardness Results

For convenience, we sometimes use an extended XPath style syntax, in order to define new monadic queries from other monadic queries  $\Phi$  defined elsewhere.

$$G ::= P \mid \neg G \mid G_1 \wedge G_2 \mid \Phi \quad \text{where } \Phi \text{ is monadic}$$

The semantics is lifted by  $\llbracket \Phi \rrbracket_{filter}(t) = \Phi(t)$ .

**Hard Classes of Queries.** We present classes of queries that are hard for streaming algorithms, in that the concurrency of some queries in the class is unbounded, for which deciding aliveness is hard.

The main idea is to start from a monadic query  $\Phi$ , and define a query  $exists(\Phi)$  that is harder to evaluate in streaming than  $\Phi$ .  $exists(\Phi)$  selects children of the root if another child  $\pi$ , late enough, is such that the subtree rooted at  $\pi$  is in  $L_\Phi$ , i.e.  $\Phi$  selects something in this subtree. One solution would be to define the last child of the root to be late enough, but this would require a query language in which one can express the next-sibling axis. Here we apply a more tedious trick, that exploits the form of the input stream. We fix two letters  $a, b \in \Sigma$ . Given a monadic query  $\Phi$  we define a monadic query  $exists(\Phi)$  with the signature  $\Sigma$  by the extended XPath expression below.

$$exists(\Phi) =_{df} \llbracket //self::*[ch::a[ch::*[\Phi]]/ch::b]_{filter} \rrbracket$$

Query  $exists(\Phi)$  selects all  $b$ -labeled children of the root, if there exists an  $a$ -labeled child of the root, which has a child that belongs to  $L_\Phi$ .

Let us consider the tree  $a(b^j)$  which has  $j$   $b$ -leaves below the root. We will show in the next lemma, that all these  $b$ -children are alive at event  $(op, j)$  if and only if  $\Phi$  is satisfiable, i.e., iff  $L_\Phi \neq \emptyset$ . This requires that query  $\Phi$  is descending in the following sense. We call a monadic query  $\Phi$  *descending* if node selection by  $\Phi$  is independent of the node's upper context, i.e. if  $\pi \in \Phi(t)$  is equivalent to  $\epsilon \in \Phi(t.\pi)$ , where  $t.\pi$  is the subtree of  $t$  rooted at  $\pi$ .

We say that a node  $\pi$  is *safely* selected (resp. rejected) by a query at event  $\eta$  if  $\pi$  is selected (resp. rejected) in all valid continuations of the stream after  $\eta$ .

**Lemma 1.** *Let  $\Phi$  be a descending monadic query,  $t = a(b^j) \in \delta(\Phi)$  and  $1 \leq k \leq j$  a natural number. It then holds that:*

1. *node  $k$  is safely rejected by  $exists(\Phi)$  at event  $(op, j)$  in  $t$  iff  $L_\Phi = \emptyset$ .*
2. *node  $k$  is alive for  $exists(\Phi)$  at  $(op, j)$  in  $t$  iff  $L_\Phi \neq \emptyset$ .*

*Proof.* ( $\Leftarrow$ ) First we assume  $L_\Phi = \emptyset$ . Then no continuation  $t' \in \mathcal{T}_\Sigma$  of  $t = a(b^j)$  beyond  $(\text{op}, j)$  can select any node, since no grandchild  $\pi$  of the root of  $t'$  can satisfy  $\pi \in \Phi(t')$ , which is equivalent to  $\epsilon \in \Phi(t'.\pi)$  since  $\Phi$  is descending. Thus  $\text{exists}(\Phi)(t') = \emptyset$  for all continuations  $t'$  of  $t$  beyond  $(\text{op}, j)$ , so that all nodes  $k$  with  $1 \leq k \leq j$  can be safely rejected.

Second, we assume the opposite  $L_\Phi \neq \emptyset$  and show that  $k$  is alive at event  $(\text{op}, j)$  in trees  $t = a(b^j)$  by query  $\text{exists}(\Phi)$ . To see this, note that the  $b$ -children of the root are selected in all continuations  $a(b^j, a(t'))$  with  $t' \in L_\Phi$ , and rejected in valid continuation  $a(b^j)$ .

( $\Rightarrow$ ) Since these cases are exhaustive, all inverse implications follow.

As a consequence, the concurrency of queries  $\text{exists}(\Phi)$ , with  $\Phi$  descending, on the collection of trees  $\{a(b^j) \mid j \in \mathbb{N}\}$  is bounded only if  $L_\Phi = \emptyset$ , since the above Lemma shows:

$$\text{concur}_{\text{exists}(\Phi)}(a(b^j)) = \begin{cases} 0 & \text{if } L_\Phi = \emptyset \\ j & \text{otherwise} \end{cases}$$

The only complete candidates that may be alive, are the  $b$ -children of the root. The concurrency is 0 if  $L_\Phi = \emptyset$ , since all children of the root can be safely rejected. Otherwise, all  $b$ -children of the root are alive at event  $(\text{op}, j)$ .

### Hardness of Streamability.

**Proposition 2.** *Let  $\mathcal{Q} = (E, \llbracket \cdot \rrbracket, |\cdot|)$  be a language of monadic queries and  $r, p_0, p_1, p_2$  polynomials such that:*

1. *queries  $\text{exists}(\llbracket e \rrbracket)$ , with  $e \in E$ , are definable by expressions in  $E$  of size  $r(|e|)$  in time  $O(r(|e|))$ ;*
2.  *$\mathcal{Q}$  is finitely streamable with polynomials  $p_0, p_1, p_2$ .*

*Then the satisfiability  $L_{\llbracket e \rrbracket} \neq \emptyset$  for definitions  $e \in E$  of descending queries can be solved in time  $O(p_0(r(|e|)) + r(|e|) + p_1(r(|e|), 0, 2) \cdot p_2(r(|e|), 0, 2))$ .*

*Proof.* Our polynomial time satisfiability test for descending queries defined in  $E$  is shown in Fig. 12. The construction of  $e'$  defining  $\text{exists}(\llbracket e \rrbracket)$  with size  $|e'| = r(|e|)$  requires time  $O(r(|e|))$ . The whole algorithm requires time  $O(p_0(r(|e|)) + r(|e|) + j \cdot p_2(|e'|))$ , which is  $O(p_0(r(|e|)) + r(|e|) + p_1(r(|e|), 0, 2) \cdot p_2(r(|e|), 0, 2))$ . It remains to argue the correctness of the algorithm.

Suppose that  $L_{\llbracket e \rrbracket} = \emptyset$ . Since  $e$  is descending, we have  $\text{concur}_{\llbracket e \rrbracket}(t) = 0$  for  $t = a(b^j)$  from Lemma 1. Since  $E$  is finitely streamable,  $\mathcal{M}_{e'}$  requires on input trees  $t$  space at most  $p_1(|e'|, \text{concur}_{\llbracket e \rrbracket}(t), \text{depth}(t)) = p_1(|e'|, 0, 2)$  per step. Hence  $\text{mem}$  is strictly less than  $p_1(|e'|, 0, 2) + 1 = j$ , and our algorithm returns *false* as expected.

Suppose now that  $L_{\llbracket e \rrbracket} \neq \emptyset$ . The RAM machine  $\mathcal{M}_{e'}$  is run on tree  $t$ , but cannot output nor discard anything until event  $(\text{op}, j)$  since all nodes  $k \in \text{nod}(t)$  with  $1 \leq k \leq j$  are still alive for  $\llbracket e \rrbracket = \text{exists}(\llbracket e \rrbracket)$  by part 2 of Lemma 1. Thus,  $\text{mem} = j$  so that our algorithm returns *true* as expected.

---

```

fun satisfQ(e) # with Q = (E, [[·]], |·|).
# for all e ∈ E such that [[e]] is descending,
# test satisfiability L[[e]] ≠ ∅.
let a, b ∈ Σ as fixed by definition of exists
compute e' with [[e']] = exists([[e]])
let j = p1(|e'|, 0, 2) + 1
let t = a(bj)
let M = Me' # needs time p0(|e'|)
let mem = maximal memory usage when running M on t until (op, j)
if mem ≥ j
  then return true # satisfiability holds
  else return false # satisfiability fails

```

---

Fig. 12: Testing satisfiability using finite streamability.

**Proposition 3.** *Satisfiability of queries in  $\text{FXP}(ch, \wedge, \neg)$  on non-recursive trees of depth at most 2 is NP-hard.*

*Proof.* We reduce 3SAT in polynomial time to satisfiability of queries defined in  $\text{FXP}(ch, \wedge, \neg)$  on non-recursive trees of depth at most 2.

Let  $\mathcal{V}$  be a finite set of variables used by a 3SAT instance. A variable assignment is encoded by a tree over finite signature  $\mathcal{V} \cup \{True, False\}$ , such that value  $b$  is assigned to variable  $x$  iff the root has an  $x$ -child having a  $b$ -child. Let  $c_1 \wedge \dots \wedge c_k$  be an instance of 3SAT using variables  $\mathcal{V}$ . We have to ensure that every variable in  $\mathcal{V}$  has at most one value:

$$\phi_1 = \bigwedge_{x \in \mathcal{V}} \neg(ch(x(ch(True(true)))) \wedge ch(x(ch(False(true))))))$$

and at least one value (for convenience we use operator  $\vee$ ):

$$\phi_2 = \bigwedge_{x \in \mathcal{V}} ch(x(ch(True(true)))) \vee ch(x(ch(False(true))))$$

We translate 3SAT instances using the following rules, where  $x \in \mathcal{V}$  is a variable,  $l_i$  is a literal  $x$  or  $\neg x$  for some  $x \in \mathcal{V}$ , and  $c_i$  a clause:

$$\begin{aligned} F(x) &= ch(x(ch(True(true)))) & F(\vee_{i \in I} l_i) &= \vee_{i \in I} F(l_i) \\ F(\neg x) &= ch(x(ch(False(true)))) & F(\wedge_{i \in I} c_i) &= \wedge_{i \in I} F(c_i) \end{aligned}$$

Then the satisfiability of a 3SAT instance  $c_1 \wedge \dots \wedge c_k$  is equivalent to the satisfiability of the  $\text{FXP}(ch, \wedge, \neg)$  formula

$$\phi_1 \wedge \phi_2 \wedge F(c_1 \wedge \dots \wedge c_k)$$

on non-recursive trees of depth at most 2 over finite signature  $\mathcal{V} \cup \{True, False\}$ .

## D Forward XPath versus FXP

We present a fragment of Forward XPath in variable-free syntax closer to the standards, and show how to map this fragment to our logic  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge, \neg)$ .

The syntax is provided in Fig. 13 and its semantics in Fig. 14. A tree  $t$  defines a binary relation  $d^t \subseteq \text{nod}(t) \times \text{nod}(t)$  for each axis  $d$  in a standard way. Path expressions  $P$  with signature  $\Sigma$  define binary queries  $\llbracket P \rrbracket_{\text{path}}$  with domain  $\mathcal{T}_\Sigma$ , while filter expressions  $G$  define monadic queries  $\llbracket G \rrbracket_{\text{filter}}$ . Note that all path expressions are also filters, so they have two distinct semantics  $\llbracket P \rrbracket_{\text{path}}$  and  $\llbracket P \rrbracket_{\text{filter}}$ . Rooted paths  $/P$  define monadic queries  $\llbracket /P \rrbracket_{\text{filter}}$ .

axes	$d ::= \text{self} \mid \text{ch}$	
paths	$P ::= S \mid P[G] \mid P_1/P_2$	
steps	$S ::= d::a \mid d::* \mid \text{outermost}(\text{ch}^*::a)$	$(a \in \Sigma)$
filters	$G ::= P \mid \neg G \mid G_1 \wedge G_2$	
rooted paths	$R ::= /P$	

Fig. 13: Syntax of Forward XPath fragment.

$$\begin{aligned}
\llbracket P \rrbracket_{\text{filter}}(t) &= \{\pi \mid \exists \pi'. (\pi, \pi') \in \llbracket P \rrbracket_{\text{path}}(t)\} \\
\llbracket \neg G \rrbracket_{\text{filter}}(t) &= \text{nod} - \llbracket G \rrbracket_{\text{filter}}(t) \\
\llbracket G_1 \wedge G_2 \rrbracket_{\text{filter}}(t) &= \llbracket G_1 \rrbracket_{\text{filter}}(t) \cap \llbracket G_2 \rrbracket_{\text{filter}}(t) \\
\llbracket /P \rrbracket_{\text{filter}}(t) &= \{\pi \mid (\epsilon, \pi) \in \llbracket P \rrbracket_{\text{path}}(t)\} \\
\llbracket d::a \rrbracket_{\text{path}}(t) &= \{(\pi, \pi') \in d^t \mid a = \text{lab}^t(\pi')\} \\
\llbracket d::* \rrbracket_{\text{path}}(t) &= d^t \\
\llbracket \text{outermost}(\text{ch}^*::a) \rrbracket_{\text{path}}(t) &= (o\text{-ch}_a^*)^t \\
\llbracket P[G] \rrbracket_{\text{path}}(t) &= \{(\pi, \pi') \in \llbracket P \rrbracket_{\text{path}}(t) \mid \pi' \in \llbracket G \rrbracket_{\text{filter}}(t)\} \\
\llbracket P_1/P_2 \rrbracket_{\text{path}}(t) &= \llbracket P_2 \rrbracket_{\text{path}}(t) \circ \llbracket P_1 \rrbracket_{\text{path}}(t)
\end{aligned}$$

Fig. 14: Semantics of Forward XPath fragment.

In Fig. 15, we map expressions of our Forward XPath fragment to  $\text{FXP}(\text{ch}, o\text{-ch}_a^*, \wedge, \neg)$  formulas. The translation of filters  $\mathcal{F}(G)$  is straightforward. Similarly, we translate rooted paths  $R$  to formulas  $\mathcal{F}(R(x))$  with a single free variable  $x$ . We annotate this variable before translation to  $R$  by using the extra filter  $[x]$ . The translation preserves the semantics: For filters, we have  $\llbracket G \rrbracket_{\text{filter}}(t) = \llbracket \mathcal{F}(G) \rrbracket_{t, \mu}$  for all variable assignments  $\mu$ . For rooted paths  $R$ , where  $x$  annotates the selection position, we have  $\llbracket R(x) \rrbracket_{\text{filter}}(t) = \llbracket \mathcal{F}(R(x)) \rrbracket(t)$ .

$$\begin{array}{ll}
\mathcal{F}(\mathit{self}::a) = a(\mathit{true}) & \mathcal{F}(\mathit{ch}::a) = \mathit{ch}(a(\mathit{true})) \\
\mathcal{F}(\mathit{self}::a[G]) = a(\mathcal{F}(G)) & \mathcal{F}(\mathit{ch}::a[G]) = \mathit{ch}(a(\mathcal{F}(G))) \\
\mathcal{F}(\mathit{self}::a/P) = a(\mathcal{F}(P)) & \mathcal{F}(\mathit{ch}::a/P) = \mathit{ch}(a(\mathcal{F}(P))) \\
\mathcal{F}(\mathit{self}::*) = \mathit{true} & \mathcal{F}(\mathit{ch}::*) = \mathit{ch}(\mathit{true}) \\
\mathcal{F}(\mathit{self}::*[G]) = \mathcal{F}(G) & \mathcal{F}(\mathit{ch}::*[G]) = \mathit{ch}(\mathcal{F}(G)) \\
\mathcal{F}(\mathit{self}::* / P) = \mathcal{F}(P) & \mathcal{F}(\mathit{ch}::* / P) = \mathit{ch}(\mathcal{F}(P)) \\
\\
\mathcal{F}(\mathit{outermost}(\mathit{ch}^*::a)) = o\text{-}\mathit{ch}_a^*(\mathit{true}) & \mathcal{F}(x) = x \\
\mathcal{F}(\mathit{outermost}(\mathit{ch}^*::a)[G]) = o\text{-}\mathit{ch}_a^*(\mathcal{F}(G)) & \mathcal{F}(\neg G) = \neg \mathcal{F}(G) \\
\mathcal{F}(\mathit{outermost}(\mathit{ch}^*::a) / P) = o\text{-}\mathit{ch}_a^*(\mathcal{F}(P)) & \mathcal{F}(G_1 \wedge G_2) = \mathcal{F}(G_1) \wedge \mathcal{F}(G_2) \\
& \mathcal{F}(/P) = \mathcal{F}(P_{[x]})
\end{array}$$

Fig. 15: Translation of Forward XPath fragment to  $\text{FXP}(ch, o\text{-}ch_a^*, \wedge, \neg)$ . We assume that the selecting position of a rooted path  $/P$  is marked by variable  $[x]$  in  $P_{[x]}$ .