Complexity of Certain Query Answering on Hyperstreams

Momar Sakho, Iovka Boneva, Joachim Niehren

Complexity of Certain Query Answering on Hyperstreams

Complexité du calcul des réponses certaines sur les hyperflux

Momar Sakho
Inria, Lille
Links team
momar.sakho@inria.fr

Iovka Boneva
Université de Lille
Links team
iovka.boneva@univ-lille1.fr

Joachim Niehren
Inria, Lille
Links team
joachim.niehren@inria.fr

ABSTRACT

A hyperstream is a sequence of streams with references to others. We study the complexity of computing certain answers for queries evaluated on hyperstreams of words. We show that the problem is PSpace-complete for deterministic query automata, but that it can be solved in PTime for linear hyperstreams even with factorization.

RÉSUMÉ

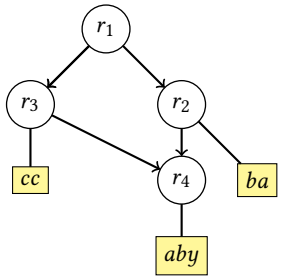
Un hyperflux est une séquence de flux ayant des références vers d'autres flux. Nous nous intéressons à la complexité du problème du calcul des réponses certaines pour les requêtes évaluées sur des hyperflux de mots. Nous montrons que ledit problème est PSpace-complet pour les automates déterministes représentant des requêtes, mais qu'il peut être résolu dans PTime pour les hyperflux linéaires, même avec une factorisation de ces derniers.

1 INTRODUCTION

Complex event processing [7, 12, 18, 20, 22] is the problem to process streams of semi-structured data, with the objective to provide low latency, low memory consumption, and very high time efficiency. This way, ever-running streams of complex events can be processed in real time, such as streams produced by social networks or trading systems. Furthermore, it becomes possible to process very large collections of semi-structured data that cannot be stored in main memory, or which size is unbounded as the stream is running forever, e.g. a Twitter stream, or a stream of events produced by a sensor.

A stream is usually seen as an object that has a known prefix, and a yet-to-come suffix. For instance, *abcy* is a stream which prefix is *abc*, and which yet unknown suffix is represented by the variable y. When pipelining stream transformations, as for define by Xslt transformations [15] or tree transducers [17], one of the transformations may produce several output streams that furthermore have references to one another. Multiple streams of nested words with forward references where called hyperstreams in [14].

In this paper we study hyperstreams restricted to words (rather than nested words) and queries defined by automata. We note that such hyperstreams are known as Dag compressed string patterns in formal language theory. Consider for example the hyperstream D_0 at Figure 1, which has the references $r_1, \ldots, r_4$, the letter a, b, c, and the string variable y. The word ab is the already known prefix of stream referred to by r_4, and the variable y represents its yet unknown continuation. As illustrated in the figure, hyperstreams can be drawn as Dags whose inner nodes are the references, and whose leafs contain streams. Then, the contents of stream r_1 can be equivalently represented by the *string pattern baabyccaby*. Such



$r_1 = r_2 r_3,$
$r_2 = bar_4,$
$r_3 = ccr_4,$
$r_4 = aby$

Figure 1: The hyperstream D_0 and its Dag.

string patterns are compositions of streams, so they may have variables in the middle and not only at the end. Conversely, the system of equations here above can be seen as a Dag compression of that string pattern. This hyperstream is nonlinear in that its string pattern contains more than one occurrence of y, due to decompression.

Example 1.1. Consider a book sales company whose workflow is illustrated in Figure 2. The company that has 3 repositories, a French repository F providing a book stream $r_F = ff'y_1$, a German repository G with books stream $r_G = gg'y_2$, and an English repository E with book stream $r_E = ey_3$. The company has a shop B in Berlin and a shop P in Paris. Shop B provides the books from E and G, coming on the concatenation of the streams $r_B = r_G r_E$, while shop P provides the books from E and F, coming on the concatenation of the streams $r_P = r_F r_E$. Furthermore, both shops B and P stream all their books to the online store O, which provides the concatenation of the streams $r_O = r_G r_G$ to his customer C. This customer may then query the resulting hyperstream, whose Dag is illustrated in Figure 3. This hyperstream factorizes the book stream r_E that is used by both shops B and P. Nevertheless, when querying this hyperstreams, matching books coming on r_E need to be returned twice, since they are available in both shops B and P. This is modeled by the fact that the appear twice on any word described by r_O, even though described in a factorized manner.

We consider queries that select positions in hyperstreams. A position of a hyperstream is called a certain answer of a query if the query selects that position in all the instances of the hyperstream, i.e., how so ever the variables in the hyperstream will be instantiated to strings. For instance in the hyperstream r_1 here above, the first position of r_2 that holds the letter b is a certain answer of the query selecting all b-positions that are immediately followed by an a-position.

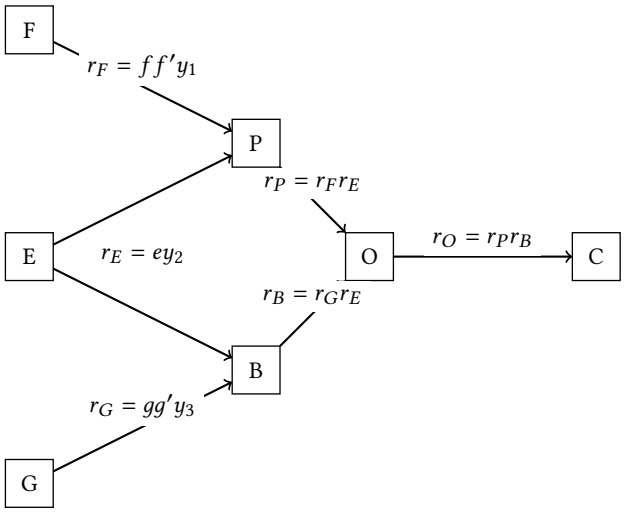


Figure 2: Workflow of the book sales company

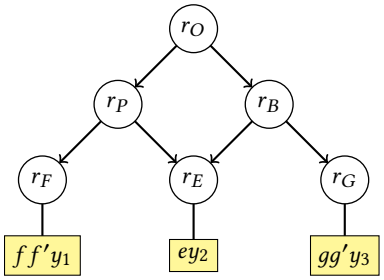


Figure 3: The hyperstream output of the workflow.

We prove that the problem to decide whether a position is a certain query answer on a hyperstream is PSpace-complete for queries defined by deterministic finite automata, while it can be solved in combined linear time (in the size of the query and the hyperstream) for linear hyperstreams without compression. For linear hyperstreams but with compression, we prove that the problem remains in PTime. This efficiency result is relevant in practice for XPath queries answering on streams, since XPath queries are usually compiled to automata [7, 16, 18]. They show that whenever an XPath query can be compiled to a small deterministic automaton, it can be evaluated efficiently on linear hyperstreams.

For obtaining our results, we introduce the problem of matching compressed string patterns against regular languages, to the best of our knowledge for the first time. An independent contribution of this paper is that this problem is PSpace-complete even without Dag compression. Previous results showed that matching compressed string patterns against rigid strings is np-complete [8].

Applications. Efficient streaming evaluation of XPath queries can be performed by compiling them to nested word automata to be evaluated on the linearization of Xml trees [7]. Then query evaluation requires to run the automaton on the input (nested) word while caching partial results. The efficiency of the evaluation depends on two factors: the per-event processing (i.e. the processing performed when some of the variables of the hyperstream is instantiated), and the quantity of memory consumed for caching. The problem of certain query answers is directly related to the latter. A partial result is a certain query answer if it would be an answer of the query how so ever the stream is extended in the future. When an answer is known to be certain (also called safe for selection in [10]), it can be immediately output, which has two advantages: first, possibly free some memory, and second, provide input for subsequent processes (if any) as illustrated in Example 1.1. On the other hand, a partial result can also be a certain nonanswer if it wonâĂŹt be part of the query answer however the stream is extended in the future. In this case discarding the partial result frees space in the cache. The hardness results of this paper directly transfer to XPath streaming evaluation on hyperstreams.

Related Work. The notion of certain answers is well known in database theory, and was initially used when querying (relational) database instances that are not known exactly, for example instances obtained by data exchange or defined in a data integration setting. The definition of certain answers was extended to Xml queries, where the challenge is that queries do not select tuples but produce tree structures as results [6], such as for XQuery. In the context of streaming (and not hyperstreaming), certain query answering was called earliest query answering [9], and certain nonanswers were called or fast-fail in [3]. It was shown in [9] that for any query language with mild restrictions, the problem of whether a tuple is a certain query answer on a stream is harder than deciding query satisfiability (the existence of an answer) and also than deciding query validity (whether all positions are answers). Therefore, the problem is np-hard for most logical query languages as well as for most subsets of XPath. The situation is better for queries defined by *deterministic* query automata, for which the problem can be solved in combined linear time. For nondeterministic query automata, in contrast, the problem becomes PSpace-complete.

Our contributions to streams with references are orthogonal to those of [17]. They study tree transformations defined by top-down tree transducers, so without any queries, while we study automata queries without transformations. Furthermore, it should be noticed that the "XML streams" used there do contain a sequence of trees (rather than the linearization of a single tree) and references to trees, rather than to other streams.

Certain query answering for automata queries requires the incremental evaluation of automata, as proposed earlier for Nfas by [4]. On the other hand side, the results of the present paper show that the case of Nfas queries is more difficult to process incrementally than Dfa queries.

The idea of producing trees with references arose in the context of Active Xml [1], but even much earlier in functional programming languages with futures [11, 19].

Outline. After some preliminaries in Section 2 on query automata and automata theory, we discuss in Section 4.2 the case of string patterns that we also call partial streams, and in Section 5.3 the case of compressed string patterns that are equivalently called hyperstreams. In both cases, we study the problems of pattern matching against regular languages, and the problem of certain query answering. As we will see, both problems are closely related.

2 PRELIMINARIES

We recall basic notions of mathematics and of automata theory on words, and recall related complexity results.

2.1 Basics

Functions. Let C and D be sets. A partial function f from C to D is a relation $f \subseteq C \times D$ that is functional, i.e., for any $c \in C$ there exists at most one $d \in D$ such that $(c, d) \in f$. In this case we write $f(c) = d$. The domain of a partial function f is the set $dom(f) = \{c \in C \mid \exists d \in D.\ f(c) = d\}$. A (total) function $f : C \to D$ is partial function from C to D with $dom(f) = C$.

Words. The set of natural numbers with 0 is denoted by $\mathbb{N}$. Let Σ be a set. A word on alphabet Σ is a tuple $(a_1, \ldots, a_n) \in \Sigma^n$ where $n \in \mathbb{N}$. We denote such a word by $a_1 \ldots a_n$ if $n \neq 0$ and by ϵ otherwise. We denote the i-th letter of a word $u = a_1 ... a_n$ by $u[i] =_{df} a_i$. The set of all words over Σ is denoted by Σ^*. The concatenation of two words $u_1, u_2 \in \Sigma^*$ is denoted by $u_1 \cdot u_2 \in \Sigma^*$. For instance, if $\Sigma = \{a, b\}$ then $aba \cdot a = abaa$. The set of positions of a word $u = a_1 \ldots a_n$ is $pos(u) = \{1, \ldots, n\}$. For any subset $\Sigma' \subseteq \Sigma$ the set $pos_{\Sigma'}(u)$ is the subset of positions i of u such that $a_i \in \Sigma'$. Given a word $w = a_1 \ldots a_n$ and a second word $u = b_1 \ldots b_n$ of the same length possibly with a different alphabet, we define the zipped word over the product alphabet by $w{*}u = (a_1, b_1) \ldots (a_n, b_n)$.

Monoids. A monoid with neutral element is a triple $(M, \cdot^M, 1^M)$ where $\cdot^M : M \times M \to M$ is an associative binary operation and $1^M \in M$ such that $1^M \cdot^M m = m \cdot^M 1^M = m$ for all $m \in M$. Given a word $u = m_1 \ldots m_n \in M^*$ we define its evaluation by $u^M = m_1 \cdot^M \ldots \cdot^M m_n$ where $\epsilon^M = 1^M$.

Most typically, we will consider the monoid of words $(\Sigma^*, \cdot, \epsilon)$ on some alphabet Σ, with the concatenation operation $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$, and the ϵ as neutral element. Alternatively, given another set Q, we will consider the *transition monoid* $(T_Q, \circ, id)$, where $T_Q = 2^{Q \times Q}$ is the set of binary relations over Q, which we will call transitions as usual in the context of automata theory [21], $\circ : T_Q \times T_Q \to T_Q$ is the composition operation of binary relations on Q, and $id = \{(q, q) \mid q \in Q\}$ is the identity transition.

2.2 Automata Theory

A nondeterministic finite-state automaton (NFA) is a tuple $A = (Q, \Sigma, \delta, I, F)$ where Q is the set of states, Σ an alphabet, $\delta \subseteq Q \times \Sigma \times Q$ a relation and $I, F \subseteq Q$ respectively the set of initial and final states. Elements $(q, a, q') \in \delta$ are denoted by $q \xrightarrow{a} q'$ and called transition rules. An NFA A is deterministic, or a DFA, if it has exactly one initial state and for every pair $(q, a) \in Q \times \Sigma$, there is at most one transition rule $q \xrightarrow{a} q' \in \delta$.

An NFA A defines a transition in T_Q for any letter of the alphabet: the transition of $a \in \Sigma$ is $a^A = \{(q, q') \mid (q, a, q') \in \delta\}$. We can also assign transitions to words $w = a_1 \ldots a_n \in \Sigma^*$ by $w^A = (a_1^A \ldots a_n^A)^{T_Q}$, i.e., by composing the transitions of all letters of w based on the operations of the transition monoid T_Q, while starting with the neutral element (the identity transition). Note that if A is a DFA then all transitions w^A are partial functions.

A transition τ is called *successful* if $\tau \cap (I \times F) \neq \emptyset$. The language of A is the set $\mathcal{L}(A) = \{w \in \Sigma^* \mid w^A \text{ is successful}\}$. The size $|A|$ of A is the sum of the number of its states and transitions. Since we fixed Σ, the size of a DFA with alphabet Σ is linear in the number of its states.

Given an NFA A with alphabet Σ and state set Q, and a transition $\tau \in T_Q$, we call τ A-inhabited, if there exists a word $w \in \Sigma^*$ such that $w^A = \tau$.

Definition 2.1 (The Inh_Σ problem). The DFA transition inhabitation problem Inh_Σ is:

Input: A DFA A with alphabet Σ, and a transition $\tau \in T_Q$ where Q is the set of states of A.
Output: The truth value of whether τ is A-inhabited.

Note that the DFA transition inhabitation problem is also known as the *membership problem of the transformation monoid* [5].

THEOREM 2.2 (KOZEN [13]). *For any set Σ with at least 2 elements, the DFA transition inhabitation problem Inh_Σ is PSPACE-complete.*

The proof can be done by reduction to the well known problem of *nonemptiness of the intersection of finite automata*. The latter problem was proven to be PSPACE-complete in the same paper [13].

3 QUERIES AND THEIR LANGUAGES

We recall a usual notion of queries on words. Our notion is semantic, so that it remains independent of how the query is defined, by logical formulas or automata. This permits us to identify queries with languages of annotated strings that are often called $\mathcal{V}$-structures [23]. Then we define query automata that recognize languages of $\mathcal{V}$-structures, thus allow to define queries.

Let $\mathcal{V}$ be a finite set of *query variables*. An assignment σ of variables to positions of a word $w \in \Sigma^*$ is a partial function σ from $\mathcal{V}$ to $pos(w)$.

Definition 3.1 (Query). A *(tuple selection)* query $\mathbf{Q}$ on words over Σ is a function that maps any word $w \in \Sigma^*$ to a set $\mathbf{Q}(w)$ of total variable assignments to $pos(w)$.

A Boolean query is a query where $\mathcal{V} = \emptyset$. While all our hardness results will apply to Boolean queries already, all our algorithms will also work in the non boolean case.

Example 3.2. Let $\mathcal{V} = \{x, x'\}$. The query $\mathbf{Q}_0$ selects all pairs of letters (x, x') such that position x is labeled by a, position x' immediately follows x and is labeled by b. This query then satisfies

$$\mathbf{Q}_0(aa) = \emptyset, \qquad \mathbf{Q}_0(ab) = \{[x/1, x'/2]\}$$
$$\mathbf{Q}_0(abab) = \{[x/1, x'/2], [x/3, x'/4]\}$$

, etc.

We will identify assignments of variables to the positions of a word as words themselves, whose letters are sets of variables. For any partial function σ from $\mathcal{V}$ to $pos(w)$ where $w \in \Sigma^n$, we define a corresponding word in $(2^{\mathcal{V}})^n$ by $word(\sigma) = \sigma^{-1}\{1\} \ldots \sigma^{-1}\{n\}$. Remark that σ is a total assignment if and only if every variable of $\mathcal{V}$ belongs to the set $word(\sigma)[i]$ for exactly one position i in $pos(w)$. Furthermore, for any word $w \in \Sigma^*$ and variable assignment σ into positions of w, we define a word over $\Sigma_V = \Sigma \times 2^{\mathcal{V}}$ by $w{*}\sigma = w{*}word(\sigma)$. In examples we will write a^V instead of letters $(a, V) \in \Sigma_{\mathcal{V}}$. For instance, $ab{*}[x/1, x'/2] = a^{\{x\}}b^{\{x'\}}$.

Definition 3.3 ($\mathcal{V}$-structure). Let $\Sigma_{\mathcal{V}} = \Sigma \times 2^{\mathcal{V}}$ be the set of letters annotated by variables. The set of *$\mathcal{V}$-structures* then is the following set of words over $\Sigma_{\mathcal{V}}$:

$$Struct_{\mathcal{V}} = \{w{*}\sigma \in \Sigma_{\mathcal{V}}^* \mid w \in \Sigma^*,\ \sigma : \mathcal{V} \to pos(w) \text{ total}\}.$$

$\mathcal{V}$-structures have the advantage that they represent assignments of variables to positions independently of how the positions were named. For instance let $\mathcal{V} = \{x, x'\}$ and $\Sigma = \{a, b\}$. Then words $a^{\emptyset}b^{\{x',x\}}$ and $a^{\{x'\}}b^{\{x\}}$ are $\mathcal{V}$-structures while the words $a^{\emptyset}b^{\{x'\}}$ and $a^{\{x'\}}b^{\{x',x\}}$ are not.

Definition 3.4 ($\mathcal{V}$-structures language of a query). We define the *language of $\mathcal{V}$-structures of a query* $\mathbf{Q}$ by:

$$\mathcal{L}(\mathbf{Q}) = \{w{*}\sigma \mid w \in \Sigma^*,\ \sigma \in \mathbf{Q}(w)\}.$$

Moreover, any language of $\mathcal{V}$-structures uniquely defines a query. We can thus identify a query by its language of $\mathcal{V}$-structures.

Definition 3.5 (Query automata). A query automaton with variables in $\mathcal{V}$ is an NFA A such that $\mathcal{L}(A)$ is a language of $\mathcal{V}$-structures. The unique query $\mathbf{Q}$ such that $\mathcal{L}(\mathbf{Q}) = \mathcal{L}(A)$ is called the query defined by automaton A and denoted by $\mathbf{Q}(A)$.

We show in Fig. 4 an example of a DFA that defines the query $a[\ldots b\ldots] + b[b\ldots]$ with variables $\mathcal{V} = \{x\}$ on words over $\Sigma = \{a, b\}$. This query selects for x all a positions that are followed by a b position sometimes later on, and selects also all b positions that are directly followed by another b.

4 CERTAIN QUERY ANSWERING ON STRING PATTERNS

We generalize the problem of certain query answering from streams [3, 10] to string patterns, where not only the end may be open, but also some factors may be unknown. This problem can also be seen as the particular case of certain query answering on hyperstreams without compression.

We first introduce string patterns and the corresponding notion of certain query answering in Sect. 4.1. Then in Sect. 4.2 we study the problem of matching a string pattern against a regular language, which allows us to establish in Sect. 4.3 complexity bounds of certain query answering on string patterns.

4.1 String Patterns and Certain Answers

We fix an infinite set $\mathcal{Y}$ of string variables for the rest of the paper. A string alphabet is a set Σ that is disjoint from $\mathcal{Y}$. A string over Σ is a word in Σ^*. An *open stream* over Σ is a string pattern in $\Sigma^*\mathcal{Y}$, starting with a closed prefix and kept open by a single variable occurrence at its open end. A *stream* over Σ is either an open stream or a string over Σ.

Definition 4.1 (String pattern). A *string pattern* over Σ is a word in $(\Sigma \cup \mathcal{Y})^*$. The set of all string patterns over Σ is denoted by $Pat(\Sigma)$.

String patterns can be instantiated step by step by using substitutions of string variables to strings patterns. For the problem of certain query answering, we are interested only on complete instantiations, called instances. An instance is obtained by a substitution $S : \mathcal{Y} \to \Sigma^*$ of string variables to strings. Any such substitution can be lifted to a substitution on string patterns $\hat{S} : Pat(\Sigma) \to \Sigma^*$ such that for all $p, p' \in Pat(\Sigma)$, $a \in \Sigma$, and $y \in \mathcal{Y}$:

$$\hat{S}(pp') = \hat{S}(p){\cdot}\hat{S}(p'), \qquad \hat{S}(\epsilon) = \epsilon, \qquad \hat{S}(a) = a, \qquad \hat{S}(y) = S(y).$$

We define the set of *instances* of a string pattern p by:

$$Inst(p) = \{\hat{S}(p) \mid S : \mathcal{Y} \to \Sigma^*\}$$

A string pattern is called *linear*, if all string variables occur at most once in the pattern. A factor of a string pattern is called *closed* if it does not contain variables, and *open* otherwise.

Clearly, all streams (open or closed) are linear string patterns, while in the general case string patterns may be nonlinear. However, even linear string patterns are more difficult to process than streams, since they may contain string variables in the middle and not only at the end. Consider for instance the linear string pattern $p = aabaybbay'$ which is the concatenation of the two open streams $p_1 = aabay$ and $p_2 = bbay'$. Recall that in streaming evaluation, all parts of the stream are to be processed as soon as they are known. Thus, an algorithm processing string patterns must be able to process the known factors $aaba$ and bba while allowing for all possible instantiations of y and y' later on. In particular, querying the string pattern $p = p_1p_2$ is more complicated than querying the two open streams p_1 and p_2 independently, since whether a query on p may certainly select a position of the sub-pattern p_2 may depend on whether the sub-pattern p_1 got sufficiently instantiated. The situation becomes even more complicated for nonlinear string patterns, where some variables have multiple occurrences, so that their instantiation must by synchronized.

Intuitively, a certain query answer on a string pattern is a tuple that answers the query on all its instances, so independently how the string pattern is completed.

Example 4.2. Consider the stream $s_0 = aabay$ over an alphabet $\Sigma = \{a, b\}$ and the query $\mathbf{Q}_0$ that selects all a-positions that are followed directly by a b-position. Then position 2 of s_0 is a certain answer of query $\mathbf{Q}_0$, since it will be selected independently of how the string variable y will be instantiated.

Definition 4.3 (Valuation). Let $\mathcal{V}$ be a finite set. A *valuation* of $\mathcal{V}$ on a string pattern p is a partial function σ from $\mathcal{V}$ to $pos_{\Sigma}(p)$.

For any valuation σ on a pattern p, the word $w{*}p$ is a string pattern with signature $\Sigma_{\mathcal{V}}$ and string variables in $\mathcal{Y}$. Therefore, the set $Inst(p{*}\sigma)$ is a well-defined set of words over $\Sigma_{\mathcal{V}}$. Note, however that some of these words may not be $\mathcal{V}$-structures. For instance, if $x \in \mathcal{V}$, $a \in \Sigma$, and $y \in \mathcal{V}$, then $a^{\{x\}}a^{\{x\}} \in Inst(y{*}[])$ where [] is the empty valuation.

Definition 4.4 (Certain answer). Let $\mathbf{Q}$ be a query on strings with alphabet Σ and query variables in $\mathcal{V}$. We call a valuation σ of $\mathcal{V}$ on a string pattern p over Σ:

- a *certain answer* for query $\mathbf{Q}$ if σ is total and $Inst(p{*}\sigma) \cap Struct_{\mathcal{V}} \subseteq \mathcal{L}(\mathbf{Q})$, and
- a *certain nonanswer* for query $\mathbf{Q}$ if $Inst(p{*}\sigma) \cap \mathcal{L}(\mathbf{Q}) = \emptyset$.

Each (partial) valuation σ describes a set of total valuations on given a instance $w \in Inst(p)$, where all remaining variables outside $dom(\sigma)$ must be mapped to positions "created" by the instantiation. More formally:

$$Complete_{p,w}(\sigma) = \{\sigma' \mid \sigma' \text{ total valuation on } w,\ w{*}\sigma' \in Inst(p{*}\sigma)\}.$$

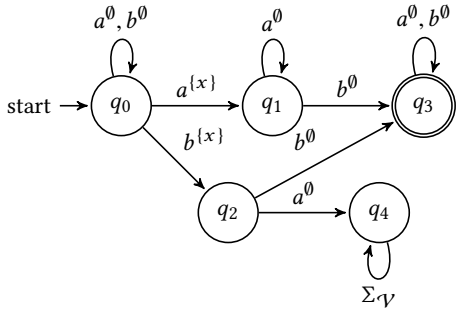


Figure 4: Example of a DFA for query $a[\ldots b\ldots] + b[b\ldots]$ where $\Sigma = \{a, b\}$ and $\mathcal{V} = \{x\}$.

There are two tedious issues here. First, even if σ is total, the set $Complete_{p,w}(\sigma)$ might not be a singleton. The reason is that there may be several substitutions S such that $\hat{S}(p) = w$. Second, it is possible that $\sigma \notin Complete_{p,w}(\sigma)$. This is due to offsets raised by the instantiation of variables in the middle of p.

Example 4.5. Consider the string pattern $p = ay_1ay_2$, the string $w = aaa$ in $Inst(p)$, $\mathcal{V} = \{x\}$, and $\sigma = [x/3]$. Then σ is a total valuation on p and $Complete_{p,w}(\sigma) = \{[x/2], [x/3]\}$. In order to see this, note that there are two substitutions that make p match w, $S_1 = [y_1/\epsilon, y_2/a]$ and $S_2 = [y_1/a, y_2/\epsilon]$ since $S_1(p) = w = S_2(p)$. These matchings satisfy $\hat{S}_1(p{*}\sigma) = w{*}[x/2]$ and $\hat{S}_2(p{*}\sigma) = w{*}[x/3]$. This shows that $[x/2]$ and $[x/3]$ belong to $Complete_{p,w}(\sigma)$. There is no further matching of p in w, so no further completion.

The next proposition shows that our notion of certain query answers and nonanswers makes good sense, in that they indeed correspond to query answers and nonanswers resp. up to completion.

PROPOSITION 4.6. *Let σ be a valuation for string pattern p, $w \in Inst(p)$ be an instance of p, and $\mathbf{Q}$ be a query:*

- *If σ is a certain answer for query $\mathbf{Q}$ then $Complete_{p,w}(\sigma) \subseteq \mathbf{Q}(w)$.*
- *If σ is certain nonanswer for query $\mathbf{Q}$ then $Complete_{p,w}(\sigma) \cap \mathbf{Q}(w) = \emptyset$.*

PROOF. For all $\sigma' \in Complete_{p,w}(\sigma)$, the definition of completion yields that $w{*}\sigma' \in Inst(p{*}\sigma)$. Furthermore, $w{*}\sigma' \in Struct_{\mathcal{V}}$ holds trivially, since it holds for any partial function σ'.

- If σ is a certain answer for query $\mathbf{Q}$ on p then $Inst(p{*}\sigma) \cap Struct_{\mathcal{V}} \subseteq \mathcal{L}(Q)$, so $\sigma' \in \mathcal{L}(\mathbf{Q})$ which is equivalent to $\sigma' \in \mathbf{Q}(w)$.
- If σ is a certain nonanswer for query $\mathbf{Q}$ then $Inst(p{*}\sigma) \cap \mathcal{L}(Q) = \emptyset$, so $w{*}\sigma' \notin \mathcal{L}(Q)$, which is equivalent to $\sigma' \notin \mathbf{Q}(w)$.

□

4.2 Regular String Pattern Matching

As we will see, our complexity results on certain query answering are closely related to a generalization of the folklore problem of matching string patterns against strings [2] to the problem of matching string pattern against regular languages of strings. This generalized problem has not been studied before, so it is of its own interest.

Definition 4.7 (The problem REGSPM$_\Sigma$). The problem of regular string pattern matching REGSPM$_\Sigma$ is:

Input: A string pattern p and a DFA A, both over Σ.
Output: The truth value of $\mathcal{L}(A) \cap Inst(p) = \emptyset$.

String pattern matching is the restriction of this problem where A is replaced by a DFA recognizing a single word. The restricted problem is well known to be NP-complete for all alphabets Σ with at least 2 letters. Furthermore, the restriction of the problem to linear pattern is known to be in P, since from a linear pattern one can construct an NFA that recognizes all its instances in linear time.

Our algorithms for solving the generalized problem REGSPM$_\Sigma$ will rely on substitutions that map string variables to transitions of some automaton. More generally, consider a monoid $(M, \cdot^M, 1^M)$, and a function $h : \Sigma \to M$. Then, any substitution on variables $S : \mathcal{Y} \to M$ can be lifted to a substitution on string patterns $S^h : Pat(\Sigma) \to M$ (depending on h) such that for all $p, p' \in Pat(\Sigma)$ and $a \in \Sigma$:

$$S^h(pp') = S^h(p) \cdot^M S^h(p'), \qquad S^h(\epsilon) = 1^M, \qquad S^h(a) = h(a),$$
$$S^h(y) = S(y)$$

Given an NFA A over Σ with state set Q, we will chose the function $h : \Sigma \to T_Q$ with $h(a) = a^A$. Then we can lift any substitution $S : \mathcal{Y} \to T_Q$ to a substitution $S^h : Pat(\Sigma) \to T_Q$. We define $S^A = S^h$, so that S^A is the substitution that assigns to any pattern $p \in Pat(\Sigma)$ a transition $S^A(p) \in T_Q$.

LEMMA 4.8. *Let A be an NFA with state set Q. A pattern p matches $\mathcal{L}(A)$ if and only if there exists a substitution of $S : \mathcal{Y} \to T_Q$ mapping to inhabited transitions such that $S^A(p)$ is successful.*

PROOF. Omitted since elementary. □

LEMMA 4.9. *Given and NFA A with state set Q, a substitution $S : \mathcal{Y} \to T_Q$, and a string pattern p, the transition $S^A(p) \in T_Q$ can be computed in time $O(|A| + |Q|^3|p|)$*

PROOF. The product $\tau \cdot^{T_Q} \tau'$ of two transitions $\tau_1, \tau_2 \in T_Q$ can be computed in time $O(|Q|^3)$, by representing transitions as Boolean $|Q| \times |Q|$ matrices, and multiplying these matrices. We have to compute $|p|$ such products. In addition, we need to inspect at most

once all transitions of A for computing the transitions a^A for all letters $a \in \Sigma$ occurring in p. □

Proposition 4.10. *RegSPM$_\Sigma$ is PSpace-complete for all alphabets Σ with at least 2 letters.*

Proof. The PSpace hardness follows by reduction to the emptiness problem of the intersection of a sequence of Dfas (or Nfas). Consider a sequence of Dfas $A_1, \ldots, A_n$ with signature Σ for some $n \geq 0$ and # a fresh symbol not in Σ. Let $A = (Q, \Sigma, \delta, I, F)$ be a Dfa that recognizes the language $\{u_1\# \ldots \#u_n \mid u_i \in \mathcal{L}(A_i)$ for all $1 \leq i \leq n\}$. Note that such a Dfa A can be constructed in linear time for the sequence $A_1 \ldots A_n$. Let p be the pattern $p = y\# \ldots \#y$ with n occurrences of pattern variable y. We then have $Inst(p) \cap \mathcal{L}(A) = \emptyset$ if and only if $\mathcal{L}(A_1) \cap \ldots \cap \mathcal{L}(A_n) = \emptyset$.

A decision algorithm in PSpace can be obtained as follows. Let p be a string pattern. By Lemma 4.8, we have that $Inst(p) \cap \mathcal{L}(A) = \emptyset$ if and only if there exists no substitution S of variables of p to inhabited transitions for Q such that the evaluation of $S^A(p)$ is successful. It is thus sufficient to generate all substitutions S of pattern variables to inhabited transitions of A, to compute the transition $S^A(p)$ and to test whether it is successful. This can be done in PSpace, since whether a transition is A-inhabited can be tested in PSpace by Theorem 2.2, and since computing $S^A(p)$ can be done in PTime by Lemma 4.9. □

When restricted to linear string patterns p, the language $Inst(p)$ will always be regular and can be recognized by a Dfa of size $O(p)$. Therefore, regular string pattern matching restricted to linear string patterns is in combined linear time.

4.3 Complexity of CQA on String Patterns

Before we can study the problem of certain query answering on string patterns, we need to specify how queries will be defined. Since we are mainly motivated in navigation XPath queries, and these can be compiled to automata [7, 16, 18], we will consider queries defined by query automata.

We consider two decision problems for certain query answers and nonanswers respectively. Both problems are parameterized by a string alphabet Σ and a finite set of query variables $\mathcal{V}$.

Definition 4.11 (The problems Cert$^{sp}_{sel}(\Sigma, \mathcal{V})$ and Cert$^{sp}_{rej}(\Sigma, \mathcal{V})$). Certainty for selection Cert$^{sp}_{sel}(\Sigma, \mathcal{V})$ and respectively rejection Cert$^{sp}_{rej}(\Sigma, \mathcal{V})$ are the following problems for string patterns:

Input: A string pattern p over Σ, a valuation σ for $\mathcal{V}$ on p, and a Dfa A which is a query automaton for strings over Σ and with variables in $\mathcal{V}$.

Output: The truth value of whether σ is a certain query answer (resp. nonanswer) for query $Q(A)$ on p.

We next show that we can reduce both problems to regular string pattern matching.

Proposition 4.12. *For any alphabet Σ and variable set $\mathcal{V}$, the problems Cert$^{sp}_{sel}(\Sigma, \mathcal{V})$ and Cert$^{sp}_{rej}(\Sigma, \mathcal{V})$ can be reduced in PTime to RegSPM$_{\Sigma_\mathcal{V}}$. Conversely, RegSPM$_\Sigma$ can be reduced in PTime to Cert$^{sp}_{sel}(\Sigma, \emptyset)$ and to Cert$^{sp}_{rej}(\Sigma, \emptyset)$.*

Proof. Let A be a Dfa query automaton for strings over Σ with variables in $\mathcal{V}$. A valuation σ mapping variables from $\mathcal{V}$ to positions of $pos_\Sigma(p)$ is a certain query answer for query $Q(A)$ if and only if $Inst(p{*}\sigma) \cap Struct_\mathcal{V} \subseteq \mathcal{L}(A)$ which is equivalent to that $Inst(p{*}\sigma) \cap Struct_\mathcal{V} \cap \overline{\mathcal{L}(A)} = \emptyset$. It is a certain query nonanswer for $Q(A)$ if and only if $Inst(p{*}\sigma) \cap \mathcal{L}(A) = \emptyset$. Since Dfas can be complemented in polynomial time, and since the set $\mathcal{V}$ is a parameter of the problem so that a Dfa for $Struct_\mathcal{V}$ can be constructed in constant time, both problems Cert$^{sp}_{sel}(\Sigma, \mathcal{V})$ and Cert$^{sp}_{rej}(\Sigma, \mathcal{V})$ can be reduced in PTime to the problem of regular string pattern matching RegSPM$_{\Sigma_\mathcal{V}}$.

If $\mathcal{V} = \emptyset$, then the converse reductions from RegSPM$_\Sigma$ to certainty for selection Cert$^{sp}_{sel}(\Sigma, \emptyset)$ and certainty for rejection Cert$^{sp}_{rej}(\Sigma, \emptyset)$ are obvious too. □

Theorem 4.13. *If Σ contains at least two elements, then independently of the choice of $\mathcal{V}$, the certainty problems for string patterns Cert$^{sp}_{sel}(\Sigma, \mathcal{V})$ and Cert$^{sp}_{rej}(\Sigma, \mathcal{V})$ are both PSpace complete.*

Proof. PSpace algorithms solving the problems Cert$^{sp}_{sel}(\Sigma, \mathcal{V})$ and Cert$^{sp}_{rej}(\Sigma, \mathcal{V})$ can be obtained by reduction to RegSPM$_{\Sigma_\mathcal{V}}$ by Proposition 4.12. The latter problem can be solved in PSpace by Proposition 4.10.

The converse reduction from Proposition 4.12 shows that Cert$^{sp}_{sel}(\Sigma, \emptyset)$ and Cert$^{sp}_{rej}(\Sigma, \emptyset)$ are harder than RegSPM$_\Sigma$ and thus PSpace-hard by Proposition 4.10. Furthermore, clearly Cert$^{sp}_{sel}(\Sigma, \mathcal{V})$ is harder than Cert$^{sp}_{sel}(\Sigma, \emptyset)$, and respectively for rejection. Hence both problems Cert$^{sp}_{sel}(\Sigma, \mathcal{V})$ and Cert$^{sp}_{rej}(\Sigma, \mathcal{V})$ are PSpace complete independently of the choice of $\mathcal{V}$. □

It should be noticed that the complexity of deciding certainty of selection or rejection for Dfa defined queries on string patterns is much higher than for ordinary streams, where the problem can be solved in combined linear time [10] in the size of the stream and the query. As we will show next, it is still possible to decide certainty efficiently for linear string patterns.

Theorem 4.14. *Restricted to linear string patterns p, the certainty problems Cert$^{sp}_{sel}(\Sigma, \mathcal{V})$ and Cert$^{sp}_{rej}(\Sigma, \mathcal{V})$ can both be solved in combined linear time $O(|p||A|)$ where A is the automaton defining the query.*

Proof. The problems can be reduced in linear time to the restriction of RegSPM$_{\Sigma_\mathcal{V}}$ to linear patterns $\mathcal{L}(A) \cap Inst(p)$ as shown in the proof of Proposition 4.12. By computing a Dfa recognizing $Inst(p)$ in time $O(|p|)$ this problem can be solved in combined linear time $O(|p||A|)$. □

5 CERTAIN QUERY ANSWERING ON HYPERSTREAMS

We next lift the concept of certain query answering to hyperstreams as considered in [14]. These are multiple streams with forward references to other streams, similarly to the streams with references of [17]. References allow to introduce factorization as in factorized databases [20], a concept that is also called compression in formal language theory [8].

5.1 Hyperstreams and Certain Answers

We introduce hyperstreams and lift the concepts of certain query answers and non-answers from string patterns to hyperstreams.

Definition 5.1 (Hyperstreams [14, 17]). Let R and Σ be disjoint sets, which do not contain string variables. A *hyperstream* with references in R and alphabet Σ, is a sequence of equations $r_1 = s_1, \ldots, r_n = s_n$ where $n \geq 1$, $r_1, \ldots, r_n \in R$ are pairwise distinct, and all s_i are streams with alphabet $\Sigma \cup \{r_{i+1}, \ldots, r_n\}$ where $1 \leq i \leq n$.

For any i, j, the references r_j that occur in streams s_i refer forwards to the stream s_j, coming later in the sequence since $i < j$ is imposed. Such forward are most natural in a streaming setting, since they refer to future information coming later.

Example 5.2. For instance, the hyperstream D_0 from page 1 of the introduction has the set of references $R = \{r_1, r_2, r_3, r_4\}$ and signature $\Sigma = \{a, b, c\}$. Its equations are as given in the introduction. The stream $r_2 r_3$ in the equation for r_1 contains the references r_2 and r_3. The streams of r_2 and r_3 both contain the reference r_4. The stream r_4 contains no references and has an open end $y \in \mathcal{Y}$.

For any hyperstream D, we define the string pattern $pat(D)$ by recursion on the number of equations, such that for all r, s, D:

$$pat(r = s) = s, \qquad pat(D, r = s) = pat(D[r/s])$$

where $D[r/s]$ is the hyperstream obtained from D by replacing by s all occurrences of reference r in the streams of the right hand sides of D. The pattern is well defined, since by definition of a hyperstream, the last stream s does not contain any reference, so that $D[r/s]$ is again a hyperstream. For instance, for the hyperstream D_0 from Example 5.2, we have $pat(D_0) = baabyccaby$.

We next lift the notions of certain query answers and nonanswers to hyperstreams.

Definition 5.3 (Certain answer). Let $\mathbf{Q}$ be a query on strings, and D a hyperstream, both over Σ. We call a valuation on $pat(D)$:

- a *certain answer* of $\mathbf{Q}$ on D if it is a certain answer of $\mathbf{Q}$ on $pat(D)$.
- a *certain non-answer* of $\mathbf{Q}$ on D if it is a certain non-answer of $\mathbf{Q}$ on $pat(D)$.

The size $|D|$ of the hyperstream D is $n + \sum_{i=1}^{n} |s_i|$ if D is equal to $r_1 = s_1, \ldots, r_n = s_n$ and $|s_i|$ is the length of the word s_i.

Linear Hyperstreams with Compression. As we will see, efficient algorithm can be obtained mainly for linear hyperstreams, i.e. hyperstreams that support the compression of linear string patterns.

Definition 5.4. A hyperstream D is called *linear* if its pattern $pat(D)$ is linear.

Example 5.5. We illustrate that linear hyperstreams may indeed represent linear string patterns in a compressed manner. Consider:

$$r_1 = r_5 r_4 r_2 r_3 r_4, \qquad r_2 = abcy_1, \qquad r_3 = bay_2 \qquad r_4 = r_5 bbr_5$$
$$r_5 = c$$

This hyperstream has the string pattern $ccbbcabcy_1bay_2cbbc$ so it is linear. It represents this string pattern in a compressed manner by using the reference r_4 and r_5 twice.

Due to compression the pattern of a hyperstream $pat(D)$ may be exponentially larger than the the hyperstream D itself, even if D is linear, see Example 5.6. Therefore, decompression, i.e., the computation of $pat(D)$ from D, must be avoided whenever possible in order to avoid exponential blow-ups.

Example 5.6. For any $n \geq 1$, consider the linear hyperstream D_n with:

$$r_1 = r_2 r_2 y, \qquad r_2 = r_3 r_3, \qquad \ldots \qquad r_n = r_{n+1} r_{n+1}, \qquad r_{n+1} = a$$

The pattern of D_n is $pat(D_n) = a^{2^n} y$ and is exponentially larger than D_n.

5.2 Regular Compressed String Pattern Matching

We study the complexity of the problem of whether a hyperstream matches some string of a regular language.

We first argue that this problem is equivalent to matching compressed string patterns against regular languages. A compressed string pattern [8] is like a hyperstream, except that the streams s_i are generalized to string patterns, so that the string variables may appear in the middle and not only at the end of s_i. It is not difficult to see that any compressed string pattern C can be converted in linear time into a hyperstream D with the same string pattern. Therefore, the difference between a compressed string pattern and a hyperstream doesn't matter for our purpose.

Definition 5.7. Regular compressed string pattern matching RegCompSPM$_\Sigma$ is the following problem:

Input: A hyperstream D over Σ, and a Dfa A over Σ.
Output The truth value of $Inst(pat(D)) \cap \mathcal{L}(A) = \emptyset$.

It also should be noticed that the more restricted problem of matching compressed string patterns against compressed strings is np-complete (see Theorem 4.4 of [8]).

For solving instances of regular compressed string pattern matching, we must be able to lift substitutions to transitions from string variables to hyperstreams.

Lemma 5.8. *Given an Nfa A with state set Q, a substitution $S : \mathcal{Y} \to T_Q$, and a hyperstream D, then the transition $S^A(pat(D)) \in T_Q$ can be computed in time $O(|A| + |Q|^3|D|)$*

Proof. We can precompute the transitions a^A for all letters $a \in \Sigma$ in time $O(|A|)$. The hyperstream D can be represented as a Dag whose nodes are the references of D. We can then compute a transition for each node of the Dag. The transition of the root of the Dag will be equal to $pat(D)^A$. The work that we have to do at node r_i is to replace in s_i each letter a by a^A, each string variable y by $S(y)$, and each reference r by the transition of node r in the Dag. We then have to multiply the sequence of transitions obtained in this way. When representing transitions as boolean $|Q| \times |Q|$ matrices, this can be done in $O(|s||Q|^3)$, so the overall time without the precomputation is in $O(|D||Q|^3)$. □

Proposition 5.9. *Given a query automaton A with state set Q and query variables $\mathcal{V}$, a substitution $S : \mathcal{Y} \to T_Q$, a hyperstream D, and a valuation σ of $\mathcal{V}$ on $pat(D)$, then the transition $S^A(pat(D) * \sigma) \in T_Q$ can be computed in PTime.*

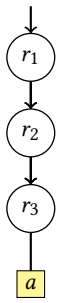


Figure 5: Hyperstream D

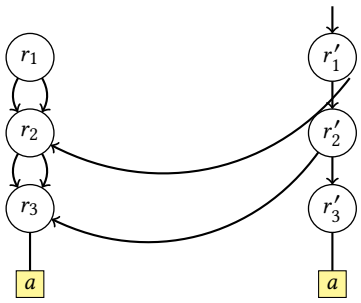


Figure 6: Newly created hyperstream D'. Here $r'_i = new(r_i)$ for all $i \in \{1, 2, 3\}$

Proof. The proof is by reduction to the special case where the positions of $dom(\sigma)$ are represented by D without sharing. This can be done by partially decompressing D in PTime.

First, we need a method to identify positions of letters in a hyperstream. Therefore, we define the set:

$$Pos_D = \{(r, j) \mid r = s \text{ in } D, j \in pos(s)\}$$

Second, we need a way to address positions of D by navigating in the Dag of D. For this, let an address be a word in $\mathbb{N}^*$, as for the usual Dewey notation for addresses in trees. For instance, if D is $r_1 = r_2r_2, r_2 = yr_3, r_3 = a$ then the address $\pi = 221$ refers to $pos_D(\pi) = (r_3, 1) \in Pos_D$. The evaluation of π always starts at the root of the Dag of D which here is r_1. Address $\pi = 221$ then requires to move to letter 2 of the stream of r_1 which is r_2, then to go to letter 2 of the stream of r_2 which is r_3, and finally to select letter 1 of the stream of r_3, which is position $pos_D(\pi) = (r_3, 1)$. We omit the general definition of $pos_D(\pi)$ which should be straightforward from the intuition. The set of all addresses of D is:

$$Addr_D = \{\pi \in \mathbb{N}^* \mid pos_D(\pi) \text{ is defined}\}$$

Third, for any position $(r, j) \in Pos_D$ we define $letter_D(r, j)$ as the j-th letter of the stream s for which $r = s$ belongs to D. For instance the $letter_D(r_3, 1) = a$. Note that $letter_D(r, j) \in \Sigma \cup R \cup \mathcal{Y}$. We then call an address $\pi \in Addr_D$ *sharing-free* in D, if there exists no different address $\pi' \in Addr_D$, such that $pos_D(\pi) = pos_D(\pi')$.

Claim 1. *For any hyperstream D and set of addresses $\Pi \subseteq Addr_D$ we can compute in PTime a hyperstream D' with $pat(D) = pat(D')$ such that all elements of Π are sharing-free addresses of D'.*

Proof. We define the set of positions of D on the paths leading to π by:

$$Pos_D^{paths}(\Pi) = \{pos_D(\pi') \mid \pi' \text{ proper prefix of } \pi \in \Pi\}$$

Note that $letter(v) \in R$ for all $v \in Pos_D^{paths}(\Pi)$. Let R' be a set disjoint from R but of the same cardinality and $new : R \to R'$ a bijection. Let D'' be like D but such that for all $v \in Pos_D^{paths}(\Pi)$ the reference at position v is replaced by $new(letter_D(v))$. The sequence $D' = D'', D$ is then a hyperstream with references in $R \cup R'$ as required by the claim. Figure 6 shows how this can be achieved, starting from the hyperstream in Figure 5.

□

We next establish a correspondence between addresses of D and positions of $pat(D)$. In our example, the address $\pi = 221$ corresponds to position 4 of $pat(D) = yaya$. More generally, we define the address $addr_D(m)$ for any position $m \in pos(pat(D))$ as follows. If D has the form $r = s$ where $1 \le m \le |s|$, then $addr_D(m) = m$. If D has the form $r = s_1ls_2, D'$ for some patterns s_1 and s_2 with references, letter $l \in \Sigma \cup \mathcal{Y} \cup R$, and hyperstream D', so that $|pat(r = s_1, D')| < m \le |pat(r = s_1l, D')|$ then:

$$addr_D(m) = m'\pi \text{ where } m' = |s_1| + 1$$

and

$$\pi' = addr_{D'}(m - |pat(r = s_1, D')|).$$

We note that $Addr_D = \{addr_D(m) \mid m \in pos(pat(D))\}$.

Claim 2. *For any m and D, we can compute $addr_D(m)$ in PTime.*

Proof. We construct the Dag of D, and compute for all its nodes (the reference of D) the size of the pattern of the subdag rooted there. This can be done in PTime in an ascending manner. Given these number of can find $addr_D(m)$ in a descending manner. □

Claim 3. *For any hyperstream D over Σ and σ a valuation of $\mathcal{V}$ to $pat(D)$ such that all addresses in the set $\{addr_D(m) \mid m \in ran(\sigma)\}$ are sharing-free in D, then we can compute in PTime a hyperstream D' over $\Sigma_{\mathcal{V}}$ such that $pat(D') = pat(D)*\sigma$.*

Proof. We obtain D' from D by applying for all $v \in pos(D)$ with $letter(v) \in \Sigma$: If $v = pos_D(addr_D(m))$ for some $m \in pos_\Sigma(pat(D))$ then replace the letter of v in D by $(letter_D(v), \sigma^{-1}(m))$, otherwise replace the letter of v by $(letter(v), \emptyset)$. Note that the choice of m is unambiguous by sharing-freeness. Furthermore, $pat(D') = pat(D)*\sigma$ since all Σ-letters that got annotated by nonempty sets of variables are at sharing-free positions. □

We can now prove Proposition 5.9. Consider a hyperstream D over Σ and a valuation σ from $\mathcal{V}$ to $\nu_\Sigma(pat(D))$. We first compute the set of addresses $\Pi = \{addr_D(m) \mid m \in ran(\sigma)$ in PTime by Claim 2. We then compute a hyperstream D' with $pat(D) = pat(D')$ such that all addresses in Π are sharing-free in D'. This can be done in PTime by Claim 2. We can then compute hyperstream D'' over $\Sigma_{\mathcal{V}}$ such that $pat(D'') = pat(D) \times \sigma$. This can be done in PTime by Claim 3.

Finally, we have that $S^A(pat(D)*\sigma) = S^A(pat(D''))$ which can be computed in PTime by Lemma 5.8, applied with the signature $\Sigma_{\mathcal{V}}$ instead of Σ. □

Proposition 5.10. *For any finite alphabet Σ with at least 2 letters, regular compressed string pattern matching RegCompSPM_Σ is PSpace-complete.*

Proof. PSpace-hardness holds already without compression, that is for the simpler problem RegSPM_Σ (Proposition 4.10). A decision algorithm in PSpace can be obtained as follows. Let D be a compressed string pattern. We have that $Inst(pat(D)) \cap \mathcal{L}(A) = \emptyset$ if and only if there exists no substitution S mapping string variables in D to inhabited transitions of A such that $S(pat(D))$ evaluates to a successful transition. It is thus sufficient to generate all substitutions S of string variables in D to A-inhabited transitions (which we can do in PSpace by Corollary 2.2), and to test whether $S(pat(D))$ evaluates to a successful transition. This can be done in PSpace by Lemma 5.9, so without ever storing the whole pattern $pat(D)$ in memory. □

We next consider a generalization of regular compressed string matching where valuations σ are part of the input.

Definition 5.11. The problem of generalized regular compressed string pattern matching $\text{GenRegCompSPM}_{\Sigma,\mathcal{V}}$ is the following:

Input: A hyperstream D over Σ, a Dfa A over $\Sigma_{\mathcal{V}}$ and a valuation σ of $\mathcal{V}$ on $pat(D)$.
Output: The truth value of $Inst(pat(D)*\sigma) \cap \mathcal{L}(A) = \emptyset$.

Lemma 5.12. *The problems $\text{GenRegCompSPM}_{\Sigma,\mathcal{V}}$ are in PSpace for all set Σ and $\mathcal{V}$ (and thus PSpace-complete if Σ contains at least 2 elements).*

Proof. Let D be a hyperstream . We have that $Inst(pat(D)*\sigma) \cap \mathcal{L}(A) = \emptyset$ if and only if there exists no substitution S of string variables of $pat(D)$ to inhabited transition of A such that $S(pat(D)*\sigma)$ evaluated to a successful transition. It is thus sufficient to generate all substitutions S of string variables to A-inhabited transitions (which we can do in PSpace by Corollary 2.2, and to test whether $S(pat(D)*\sigma)$ evaluates to a successful transition. This can be done in PTime by Lemma 5.8, without generating $pat(D)$. □

5.3 Complexity of CQA on Hyperstreams

We next study the problems of certainty for selection and rejection for hyperstreams.

Definition 5.13. Let Σ and $\mathcal{V}$ be finite sets. Certainty for selection $\text{Cert}_{sel}(\Sigma, \mathcal{V})$ and resp. rejection $\text{Cert}_{rej}(\Sigma, \mathcal{V})$ are the following problems:

Input: A hyperstream D over Σ, a valuation σ on $pat(D)$ with variables in $\mathcal{V}$, and a Dfa A which is a query automaton for strings over Σ and with variables in $\mathcal{V}$.
Output: The truth value of whether σ is a certain answer (resp. nonanswer) of query $Q(A)$ on D.

These problems generalize $\text{Cert}^{sp}_{sel}(\Sigma, \mathcal{V})$ and resp. $\text{Cert}^{sp}_{rej}(\Sigma, \mathcal{V})$ from string patterns to hyperstreams.

Theorem 5.14. *For any signature Σ with at least 2 letters and for any finite set $\mathcal{V}$ of variables, the problems of certainty for selection $\text{Cert}_{sel}(\Sigma, \mathcal{V})$ and resp. rejection $\text{Cert}_{rej}(\Sigma, \mathcal{V})$ are both PSpace-complete.*

Proof. From Theorem 4.13 we know that both problems restricted to string patterns are already PSpace-hard. It remains to show that both problems are in PSpace.

We first show that $\text{Cert}_{rej}(\Sigma, \mathcal{V})$ is in PSpace. We have to decide $Inst(pat(D)*\sigma) \cap \mathcal{L}(A) = \emptyset$, given a Dfa $A = (Q, \Sigma, \delta, I, F)$, a hyperstream D, and σ. This is equivalent to that there exists a substitution S from string variables in $\mathcal{Y}$ to transitions for Q, so that the transition $(I \times F) \cap S^A(pat(D)*\sigma)$ is A-inhabited. It is thus sufficient to generate and test all such substitution S in PSpace. We can compute the transition $S^A(pat(D)*\sigma)$ in PSpace by Lemma 5.9, and then to test whether $(I \times F) \cap S^A(pat(D)*\sigma)$ is inhabited in PSpace by Corollary 2.2.

We next consider $\text{Cert}_{sel}(\Sigma, \mathcal{V})$. A valuation σ on $pat(D)$ is a certain answer for query $Q(A)$ on D if and only if $Inst(pat(D)*\sigma) \cap Struct_{\mathcal{V}} \subseteq \mathcal{L}(A)$ which is equivalent to that $Inst(pat(D)*\sigma) \cap Struct_{\mathcal{V}} \cap \overline{\mathcal{L}(A)} = \emptyset$. Since we fixed the set $\mathcal{V}$ as a parameter of the problem, we can compute a Dfa recognizing $Struct_{\mathcal{V}}$ in constant time (it may be of size $2^{\mathcal{V}}$ though). So we can compute a Dfa A' with $\mathcal{L}(A') = Struct_{\mathcal{V}} \cap \overline{\mathcal{L}(A)}$ in linear time in the size of A. The valuation σ is a certain answer of $Q(A)$ if and only if $Inst(pat(D)*\sigma) \cap \overline{\mathcal{L}(A')} = \emptyset$, which is equivalent to that σ is a certain nonanswer of $Q(A')$ on D. We thus have reduced the problem in PTime to $\text{Cert}_{rej}(\Sigma, \mathcal{V})$. □

Theorem 5.15. *The problems of certainty of selection and resp. rejection on* linear *hyperstreams for queries defined by Dfas are both in PTime.*

Proof. Given that certainty for selection can be reduced to certainty for rejection in PTime, it is sufficient to show that certainty for rejection on linear hyperstreams is in PTime. So consider as inputs, a Dfa query automaton $A = (Q, \Sigma, \delta, I, F)$, and a linear hyperstream D. We have to decide whether $Inst(pat(D)*\sigma) \cap \mathcal{L}(A) = \emptyset$. Given that D is linear, all its string variables can be instantiated independently. Therefore, we can consider the transition τ_{acc}, that contains all pairs (q, q') of A such that q' is accessible from q by some path in A, and the substitution S_{acc} that maps all string variables of D to τ_{acc}. It then holds that $(I \times F) \cap {S_{acc}}^A(pat(D))$ is a A-inhabited if and only if $Inst(pat(D)*\sigma) \cap \mathcal{L}(A) \neq \emptyset$. Now, by Proposition 5.9 we can compute ${S_{acc}}^A(pat(D))$ in PTime, so certainty for rejection can be tested in PTime too. □

6 CONCLUSION

We have shown that deciding whether a valuation is a certain query answer can be decided in PTime for linear hyperstreams. This result is nontrivial due to the factorization that comes with compressed string patterns. Without the linearity restriction, the general problem becomes PSpace-complete. Both the hardness and the decidability result are new.

From a practical perspective, what is missing most urgently is an algorithm that computes all certain query answers of a linear hyperstream in an earliest manner, that is an algorithm that outputs

valuations as soon as they become certain answers. The example in the introduction shows that it is also of interest to lift such algorithms to general hyperstreams. Our hardness result, however, prove that this is not possible efficiently. What could still be done is to relax the condition on earliest query answering. We believe that efficient early algorithm can be done by approximating the set of certain query answers. It may also be of interest to consider certain query answering for other kinds of uncomplete structures, such as partially known graphs or trees.

Another point is to lift our streaming algorithm to hyperstreams with data values from an infinite signature. We believe that this can be done by decomposing queries into parts that are comparing data values, and others that do not. But still the feasability of hyperstreaming in general needs to be proven in practice.

Acknowledgments. We are thankful to Charles Paperman, who saw the PSpace-hardness of regular string pattern matching in a discussion on the topic. We also thank him for having pointing us to existing work on the transition monoid. We thank Sylvain Salvati and Sophie Tison for discussions on regular string pattern matching. It is a pleasure to thank the reviewers of the BDA national conference for their extraordinary helpful feedback.

REFERENCES

[1] S. Abiteboul, O. Benjelloun, and T. Milo. The active XML project: an overview. *VLDB J.*, 17(5):1019–1040, 2008.
[2] D. Angluin. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21(1):46–62, 1980.
[3] M. Benedikt, A. Jeffrey, and R. Ley-Wild. Stream Firewalling of XML Constraints. In *ACM SIGMOD International Conference on Management of Data*, pages 487–498. ACM-Press, 2008.
[4] H. Björklund, W. Gelade, and W. Martens. Incremental xpath evaluation. *ACM Trans. Database Syst.*, 35(4):29, 2010.
[5] M. Blondin, A. Krebs, and P. McKenzie. The complexity of intersecting finite automata having few final states. *computational complexity*, 25(4):775–814, Dec 2016.
[6] C. David, L. Libkin, and F. Murlak. Certain answers for XML queries. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 191–202. ACM, 2010.
[7] D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, and M. Zergaoui. Early nested word automata for xpath query answering on XML streams. *Theor. Comput. Sci.*, 578:100–125, 2015.
[8] A. Gascón, G. Godoy, and M. Schmidt-Schauß. Context matching for compressed terms. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 93–102. IEEE Computer Society, 2008.
[9] O. Gauwin and J. Niehren. Streamable fragments of forward XPath. In B. B. Markhoff, P. Caron, J. M. Champarnaud, and D. Maurel, editors, *International Conference on Implementation and Application of Automata*, volume 6807 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2011.
[10] O. Gauwin, J. Niehren, and S. 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.
[11] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.
[12] M. Kay. A streaming XSLT processor. In *Balisage: The Markup Conference 2010. Balisage Series on Markup Technologies*, volume 5, 2010.
[13] D. Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 254–266. IEEE Computer Society, 1977.
[14] P. Labath and J. Niehren. A functional language for hyperstreaming XSLT. Technical report, INRIA Lille, 2013.
[15] P. Labath and J. Niehren. A uniform programming language for implementing XML standards. In *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings*, pages 543–554, 2015.
[16] P. Madhusudan and M. Viswanathan. Query automata for nested words. In *34th International Symposium on Mathematical Foundations of Computer Science*, volume 5734 of *Lecture Notes in Computer Science*, pages 561–573. Springer Verlag, 2009.
[17] S. Maneth, A. O. Pereira, and H. Seidl. Transforming XML streams with references. In C. S. Iliopoulos, S. J. Puglisi, and E. Yilmaz, editors, *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, volume 9309 of *Lecture Notes in Computer Science*, pages 33–45. Springer, 2015.
[18] B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over XML streams. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, A. Fuxman, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *SIGMOD Conference*, pages 253–264. ACM, 2012.
[19] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, Nov. 2006.
[20] D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. on Know. Data Eng.*, 19(7):934–949, 2007.
[21] J.-E. Pin. *Mathematical Foundations of Automata Theory*. 2016.
[22] M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *23rd IEEE International Conference on Data Engineering*, pages 236–245, 2007.
[23] H. Straubing. *Finite Automata, Formal Logic, and Circuit Complexity*. Progress in Computer Science and Applied Series. Birkhäuser, 1994.