Automata-based Stream Processing

Rajeev Alur, Konstantinos Mamouras, and Caleb Stanford

Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104, USA
{alur,mamouras,castan}cis.upenn.edu

Abstract

We propose an automata-theoretic framework for modularly expressing computations on streams of data. With weighted automata as a starting point, we identify three key features that are useful for an automaton model for stream processing: expressing the regular decomposition of streams whose data items are elements of a complex type (e.g., tuple of values), allowing the hierarchical nesting of several different kinds of aggregations, and specifying modularly the parallel execution and combination of various subcomputations. The combination of these features leads to subtle efficiency considerations that concern the interaction between nondeterminism, hierarchical nesting, and parallelism. We identify a syntactic restriction where the nondeterminism is unambiguous and parallel subcomputations synchronize their outputs. For automata satisfying these restrictions, we show that there is a space- and time-efficient streaming evaluation algorithm. We also prove that when these restrictions are relaxed, the evaluation problem becomes inherently computationally expensive.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases weighted automata, Quantitative Regular Expressions, stream processing

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Finite-state automata have been used very successfully to solve the problem of pattern matching in strings [1]. For simple patterns that are given as regular expressions, there have been proposed several pattern-matching algorithms based on Nondeterministic Finite Automata (NFAs) [31] or Deterministic Finite Automata (DFAs) [7] with strong efficiency guarantees. A particularly desirable feature of such automata-based algorithms is that they process the input text in one pass, i.e. by reading each letter of the input text consecutively from left to right, thus adhering to the so-called *streaming model* of computation [28].

Pattern-matching is one basic computational problem that arises in the context of data stream processing [14], i.e. the processing of data that arrives in real time at a high rate (e.g., for analyzing stock market data and web click-streams, or for monitoring sensor measurements and network traffic). To process data streams, the core computational problem that typically needs to be solved is the aggregation of parts of the stream into numerical values. For example, calculating the average price of a stock, monitoring the amount of network traffic an IP address has generated so far, or maintaining for a sensor the minimum and maximum measurements it has recorded over the last 10 minutes. Given the usefulness of automata for finding patterns in streams of symbols, the question arises whether similar automata-based techniques can be employed for computing quantitative summaries of data streams.

We are thus led to consider weighted automata [19], which extend classical nondeterministic automata by annotating transitions with *weights* and can be used for the computation of simple quantitative properties on finite or infinite strings of symbols [10]. Weighted automata

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Acces;
Article No. 23; pp. 23:1–23:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

have found applications in speech and language processing [26], and they are also used for modeling systems and verifying quantitative properties of these systems [12]. However, the computational problems that are relevant for quantitative verification are analysis questions such as universality and equivalence. These questions are decidable only when the weights and the operations used on them are very simple [24, 2], so the studied models are usually equipped with a very limited set of primitive operations that are insufficient for expressing realistic streaming computations.

Since weighted automata are not expressive enough for typical streaming computations, our goal is to extend them for this purpose while maintaining the efficiency of their evaluation. First, we notice that the elements of data streams are typically not symbols from a finite alphabet but rather structured objects such as tuples of values. It is therefore necessary to work in the *symbolic* setting [33, 34]: the input elements belong to a potentially infinite alphabet D, and we consider a collection of primitive predicates on D for describing subclasses of elements using Boolean formulas over the primitive predicates. Additionally, realistic computations often involve the parsing of an input stream and aggregation of subcomputations; for example, we may want to subsample a sequence of sensor measurements by averaging them in groups of three consecutive measurements, and then compute the maximum measurement of every minute. Naturally describing such calculations requires that we allow *hierarchical nesting* of operations. In general, the required subcomputations may be disjoint from one another, and need to be executed in parallel. For example, suppose the automaton $\mathcal{A}_1$ describes a long-term average (e.g., over the last month) of a sensor measurement, $\mathcal{A}_2$ calculates a short-term average (e.g., over the last minute), and op is the "absolute difference" binary operation. Then, the construct $\mathrm{op}(\mathcal{A}_1, \mathcal{A}_2)$ describes the *parallel execution* of $\mathcal{A}_1$ and $\mathcal{A}_2$ and the *combination* of their results using the op operation. Thus, the overall computation outputs the distance between the short-term and long-term average. This construct for parallelism facilitates the modular description of computations.

Our contribution. Putting these desired features together in a model that supports nondeterministic parsing, hierarchical nesting of quantitative operations and modular parallelism is challenging. The core computational problem is the incremental evaluation of automata on unbounded data streams, and the goal is to provide an algorithm with strong space- and time-efficiency guarantees. We will establish formally that the naive combination of the desired features makes efficient evaluation impossible. Moreover, we will show that by restricting to unambiguous nondeterminism [9] and by constraining the parallel execution of $\mathrm{op}(\mathcal{A}_1, \ldots, \mathcal{A}_k)$ so that the automata $\mathcal{A}_i$ synchronize their outputs, we can achieve very efficient evaluation. More specifically, our main results are the following:

(1) The evaluation problem for automata that allow ambiguous nondeterminism and nesting of quantitative operations requires space that is linear in the size of the input stream.

(2) The evaluation problem for automata with unambiguous nondeterminism and unsynchronized parallel execution requires space that is exponential in the size of the automaton.

(3) For automata that are unambiguous and allow only synchronized parallel execution, the evaluation problem requires space and time-per-element that is quadratic in the size of the automaton and independent of the size of the stream.

Related work. The features of our Streaming Automata (SAs) were inspired by the Quantitative Regular Expressions (QREs) of [5], which have constructs for parallelism and nesting of sequential aggregators. QREs were extended in [25] with streaming relational operations [22], and an efficient implementation was given for processing realistic workloads (Yahoo streaming benchmark [13] and NEXMark benchmark [32]). However, the evaluation algorithm of [5] and the implementation of [25] were not based on automata-theoretic techniques. A simplified version of the QREs of [5] without parameters allows a straightforward translation into our SAs that is very similar to the translation of unambiguous regexes into unambiguous NFAs. This translation is desirable not only because it gives rise to a cleaner evaluation algorithm, but also because it opens the door for systematic query optimization using automata-theoretic techniques, which could be explored in future research.

The model of Cost Register Automata (CRAs) was proposed in [4] and was shown in [5] to be expressively equivalent to QREs. However, CRAs cannot be used for the efficient evaluation of QREs, because the translation of QREs into CRAs incurs a doubly exponential blowup. The model of Streaming Automata that is proposed here is an appropriate setting for the efficient evaluation of QREs.

A two-level variant of weighted automata for infinite strings has recently been proposed [11] that can express long-run quantitative properties of a stream, for example, the average response time of a system. By restricting both the nesting depth (to 1) and the allowed aggregation operations, the model of [11] is shown to have decidable emptiness and universality problems. With the goal of modeling realistic streaming computations, we focus on arbitrary nesting and a general set of operations. We are therefore concerned primarily with evaluation complexity rather than decidability of these problems.

Symbolic automata and transducers [33, 34, 15, 16] have been introduced for matching and transforming strings over large or infinite alphabets. Our work builds on symbolic automata but instead addresses the problem of quantitative aggregation.

There is also related work on data words and data/register automata and their associated logics [23, 29, 18, 8]. These models operate on words over an infinite alphabet, which is typically of the form $\Sigma \times \mathbb{N}$, where Σ is a finite set of tags. They allow the comparison of infinite values using only the equality predicate. In contrast, our SAs do not allow binary predicates on stream elements, but instead allow a rich set of operations on the values.

More broadly, there is a vast line of research on efficient algorithms for the streaming model of computation. See the survey [28] and some illustrative works [27, 20, 3, 17, 6] that have been influential. The algorithms studied in this line of research are designed for specific problems (for example, finding the number of distinct elements in a stream) and typically use approximation and randomization. Our considerations here are orthogonal, and complementary, to the literature on streaming algorithms. We study the hierarchical nesting of several different kinds of aggregations, and we study the computational resources that are needed for parsing the stream and combining all intermediate results.

2 Streaming Automata

Symbolic input. Figure 1 shows two symbolic weighted automata over different inputs. $\mathcal{M}_1$ implements MaxBlockSum: on an input stream of natural numbers separated into (possibly empty) blocks by the separator 0, it returns the maximum sum of a block. As we may view $\mathcal{M}_1$ as a weighted automaton over the semiring $(\mathbb{N} \cup \{-\infty\}, \max, +)$, it does not yet introduce anything new to our model except the symbolic input. All transitions use the formal variable x to denote the current input data item, a natural number; the syntax $\varphi(x) \mapsto \alpha(x)$ means that if x matches predicate φ, then the transition can be taken, and has weight $\alpha(x)$. We write simply $x \mapsto \alpha(x)$ if φ is True, i.e. if any x is allowed. A transition labeled with $\varepsilon \mapsto r$ matches the empty string and has weight r.

$\mathcal{M}_1$ starts at q_0 for some time, mapping each input x to weight 0 (effectively ignoring it).

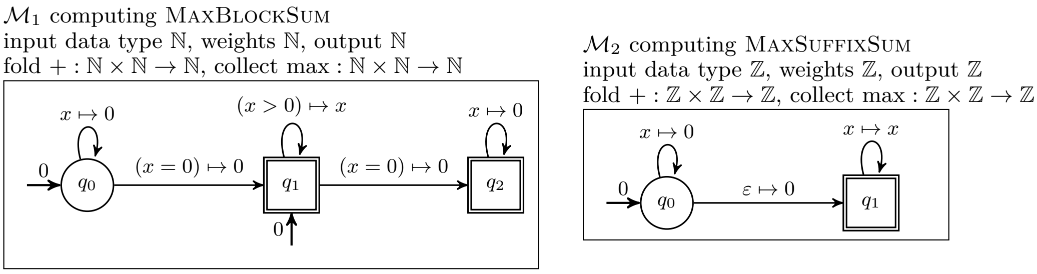

$\mathcal{M}_1$ computing MaxBlockSum
input data type $\mathbb{N}$, weights $\mathbb{N}$, output $\mathbb{N}$
fold $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, collect $\max : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$x \mapsto 0$
$(x > 0) \mapsto x$
$x \mapsto 0$
0
q_0
$(x = 0) \mapsto 0$
q_1
$(x = 0) \mapsto 0$
q_2
0
$\mathcal{M}_2$ computing MaxSuffixSum
input data type $\mathbb{Z}$, weights $\mathbb{Z}$, output $\mathbb{Z}$
fold $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, collect $\max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
$x \mapsto 0$
$x \mapsto x$
0
q_0
$\varepsilon \mapsto 0$
q_1

Figure 1 Weighted automata with symbolic input.

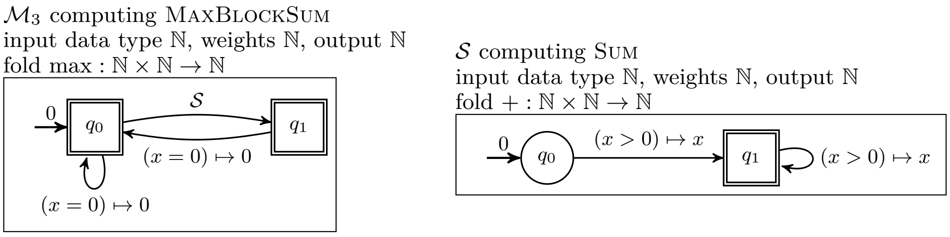

$\mathcal{M}_3$ computing MaxBlockSum
input data type $\mathbb{N}$, weights $\mathbb{N}$, output $\mathbb{N}$
fold $\max : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$\mathcal{S}$
0
q_0
q_1
$(x = 0) \mapsto 0$
$(x = 0) \mapsto 0$
$\mathcal{S}$ computing Sum
input data type $\mathbb{N}$, weights $\mathbb{N}$, output $\mathbb{N}$
fold $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
0
q_0
$(x > 0) \mapsto x$
q_1
$(x > 0) \mapsto x$

Figure 2 A streaming automaton employing hierarchy.

Then, it nondeterministically picks a block by transitioning to q_1 on input the separator $x = 0$. (q_1 is also a start state, which corresponds to the first block, before any 0 has occurred.) At q_1, all inputs matching the predicate $x > 0$ are assigned weight x. Finally, on input $x = 0$, the end of the block, it transitions to q_2, where future x are again assigned weight 0. $\mathcal{M}_1$ adds up (*folds*) all the assigned weights to obtain the total weight of the path, which is by construction the sum of the particular block chosen. The output of the automaton is the maximum weight (*collect*) over all paths.

$\mathcal{M}_2$ implements MaxSuffixSum: on an input stream of integers, it returns the maximum sum of a suffix of those integers. The input data type is now $\mathbb{Z}$ rather than $\mathbb{N}$. $\mathcal{M}_2$ (like $\mathcal{M}_1$) starts at q_0 and assigns inputs x to weight 0 for some time. Then, it nondeterministically guesses the start of the suffix by switching to q_1, where each future input x is assigned weight x. The fold operation is again $+$, so that the weight of the path is the sum of that particular suffix. The collect operation returns the max over all paths, i.e. over all suffixes.

Hierarchy. The nondeterminism of $\mathcal{M}_2$ is very natural: exactly where the best suffix starts cannot be known ahead of time, so we choose it nondeterministically. In contrast, since the input to $\mathcal{M}_1$ is *parsable* into a sequence of blocks, using nondeterminism to choose a block seems artificial. Instead, we would like to deterministically parse the stream into blocks, then call a subroutine (sum) on each block. Figure 2 shows how to do this in our model. First, the weighted automaton $\mathcal{S}$ is built to compute the sum of a nonempty input stream by straightforwardly folding with $+$. $\mathcal{M}_3$ parses the stream into blocks separated by 0 and calls $\mathcal{S}$ as a *subautomaton* on each block, where the weight of that transition is the return value of $\mathcal{S}$. All the block sums returned by $\mathcal{S}$ are now weights along a single path, and they are folded with the operation max.

The example of MaxBlockSum is a typical case where the two operations of a nondeterministic weighted automaton (fold $\otimes$ and collect $\oplus$) can be replaced by a hierarchy of two streaming automata, each of which is *unambiguous*: there is at most one accepting path on any given input string. The fold operation of $\mathcal{M}_1$ $(+)$ becomes the fold operation of $\mathcal{S}$, and

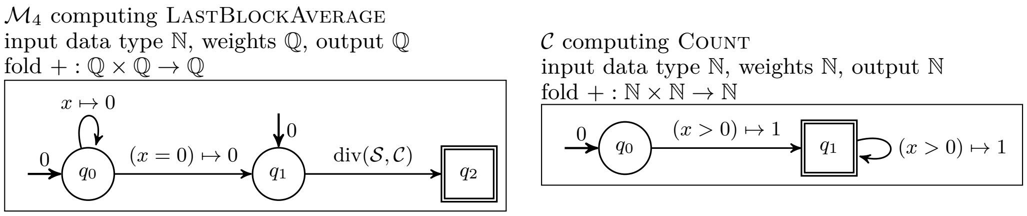

$\mathcal{M}_4$ computing LastBlockAverage
input data type $\mathbb{N}$, weights $\mathbb{Q}$, output $\mathbb{Q}$
fold $+ : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$
$x \mapsto 0$
0
q_0
$(x = 0) \mapsto 0$
0
q_1
$\mathrm{div}(\mathcal{S}, \mathcal{C})$
q_2
$\mathcal{C}$ computing Count
input data type $\mathbb{N}$, weights $\mathbb{N}$, output $\mathbb{N}$
fold $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
0
q_0
$(x > 0) \mapsto 1$
q_1
$(x > 0) \mapsto 1$

Figure 3 A streaming automaton employing parallelism.

the collect operation of $\mathcal{M}_1$ (max) becomes the fold operation of $\mathcal{M}_3$. Unambiguity implies that the collect operations in $\mathcal{M}_3$ and $\mathcal{S}$ are never used, and need not be specified.

Parallelism. After parsing a stream into blocks, multiple computations may be required on each block. For this purpose, in our model a transition may be labeled not just with a single subautomaton (as in $\mathcal{M}_3$), but with a call $\mathrm{op}(\mathcal{A}_1, \ldots, \mathcal{A}_m)$ where each $\mathcal{A}_i$ is a subautomaton. In a simple example, the stream is separated by 0 into blocks, and we want to report the average of the last block. Figure 3 gives an automaton $\mathcal{M}_4$ implementing this. On every 0 character $\mathcal{M}_4$ may nondeterministically guess that we are now going to the last block, and move from q_0 to q_1. It subsequently makes an invocation $\mathrm{div}(\mathcal{S}, \mathcal{C})$ to two subautomata. $\mathcal{S}$ (from Figure 2) returns the sum of the elements in the block if there is at least one, and $\mathcal{C}$ returns the count if there is at least one. $\mathrm{div} : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$ then divides the two results to get average. The *parallelism* arises because the stream is read into both $\mathcal{S}$ and $\mathcal{C}$ in parallel.

Like $\mathcal{M}_3$, $\mathcal{M}_4$ is unambiguous, with at most one accepting path on each input. $\mathcal{M}_4$ also satisfies *parallel-consistency*: in the call to $\mathrm{div}(\mathcal{S}, \mathcal{C})$, $\mathcal{S}$ and $\mathcal{C}$ were defined on the same input strings. Our definition of a *streaming automaton* requires both unambiguity and parallel-consistency; the necessity of these restrictions is justified by Section 4.

Formal definition

The general definition is parameterized by a *signature* $(\mathcal{D}, \mathcal{O}, D, \mathcal{P})$, where $\mathcal{D}$ is a collection of (possibly infinite) types, and $\mathcal{O}$ is a collection of operations $D_1 \times D_2 \times \cdots \times D_k \to D_{k+1}$ with each D_i a type in $\mathcal{D}$. We write $\mathcal{O}[D_1 \times D_2 \times \cdots \times D_k \to D_{k+1}]$ for the set of operations in $\mathcal{O}$ which are functions of the specific indicated function type. $D \in \mathcal{D}$ is a specific set for the input stream, and $\mathcal{P}$ is a set of predicates, which are identified with subsets of D. We require that $\mathcal{P}$ is closed under Boolean operations, and that satisfiability for $\varphi \in \mathcal{P}$ is decidable as in [34]. From this point, we assume the fixed signature $(\mathcal{D}, \mathcal{O}, D, \mathcal{P})$.

The class of *nondeterministic streaming automata* is defined hierarchically as $\mathrm{NSA} := \bigcup_{k=0}^{\infty} \mathrm{NSA}_k$. For $k \geq 0$, an element of NSA_k is a tuple $(Q, X, Y, \Delta, I, F, \otimes, \oplus)$, semantically representing a partial function from D^* to Y. Q is a finite set of *states*, $X \in \mathcal{D}$ is the *weight type*, $Y \in \mathcal{D}$ is the *output type*, and Δ is a set of *transitions*. Each transition goes from a state $q \in Q$ to a state $q' \in Q$, and has a *label*, which is one of three *kinds*: (i) A satisfiable predicate $\varphi \in \mathcal{P}$ and a *weight assignment* $\alpha \in \mathcal{O}[D \to X]$. (ii) An epsilon ($\varepsilon$) and a *weight* $x \in X$. (iii) A call to $\mathrm{op}(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m)$, where $\mathrm{op} \in \mathcal{O}[Y_1 \times Y_2 \times \cdots \times Y_m \to X]$ and each $\mathcal{A}_i \in \mathrm{NSA}_{k-1}$, such that the output type of $\mathcal{A}_i$ is Y_i. The weight of the transition in this case will be op applied to the outputs of the $\mathcal{A}_i$.

$I : Q \rightharpoonup Y$ is the *initialization function*, a partial function assigning an initial value to the computation. Its domain is the set of *initial states*, denoted $Q_I \subseteq Q$. Conversely, $F : Q \rightharpoonup X$ is the *final function*; it allows for slightly more flexibility than in our examples by appending a final weight to accepting paths. Its domain is the set of *final states* or *accepting states*, denoted $Q_F \subseteq Q$. The *fold operation* $\otimes \in \mathcal{O}[Y \times X \to Y]$ folds together the weights along a path, and the *collect operation* $\oplus \in \mathcal{O}[Y \times Y \to Y]$ combines the results of all accepting paths to arrive at a final output value. The operation $\oplus$ must be commutative and associative, and $\otimes$ must be left-distributive over $\oplus$.

The class NSA_0, in which there are no transitions of kind (iii), consists of symbolic weighted automata. A *subautomaton* of $\mathcal{A}$ is an automaton $\mathcal{A}_i \in \mathrm{NSA}_{k-1}$ appearing in a transition of kind (iii) in A. The *size* of A is the sum of the number of states $|Q|$, the number of transitions $|\Delta|$, and the sizes of all the subautomata, counted with multiplicity. Effectively, an automaton must be written down once for every time it is used.

As in the examples, the automaton $\mathcal{A}$ is semantically interpreted as a function $[\![\mathcal{A}]\!] : L(\mathcal{A}) \to Y$, where $L(\mathcal{A}) \subseteq D^*$ is the regular *language* of $\mathcal{A}$. $L(\mathcal{A})$ and $[\![\mathcal{A}]\!]$ are defined recursively by also defining $L(\tau)$ and $[\![\tau]\!]$ for each transition τ of the automaton. (i) For a transition τ labeled with predicate $\varphi \subseteq D$ and weight assignment $\alpha : D \to X$, $L(\tau) = \{d \in D \mid \varphi(d)\}$, and $[\![\tau]\!](d) = \alpha(d)$. (ii) For an epsilon transition τ with weight $x \in X$, $L(\tau) = \{\varepsilon\}$ and $[\![\tau]\!](\varepsilon) = x$. (iii) Finally, for a transition τ labeled with $\mathrm{op}(\mathcal{A}_1, \ldots, \mathcal{A}_m)$, the language $L(\tau) = L(\mathcal{A}_1) \cap \cdots \cap L(\mathcal{A}_m)$, and for any string $s \in L(\tau)$, $[\![\tau]\!](s) = \mathrm{op}([\![\mathcal{A}_1]\!](s), \ldots, [\![\mathcal{A}_m]\!](s))$.

For an automaton $\mathcal{A} \in \mathrm{NSA}_k$, a *path* on input $s \in D^*$ consists of a sequence of states $q_0, q_1, q_2, \ldots, q_n \in Q$, a sequence of strings $s_1, s_2, \ldots, s_n \in D^*$, and a sequence of transitions $\tau_1, \tau_2, \ldots, \tau_n \in \Delta$, such that $q_0 \in Q_I$, $s = s_1 s_2 \ldots s_n$, and for each i, τ_i is a transition from q_{i-1} to q_i such that $s_i \in L(\tau_i)$. A path is *accepting* if $q_n \in Q_F$. The *language* $L(\mathcal{A})$ is the set of strings s for which there exists an accepting path on input s. The *weight* of an accepting path is, with left-to-right evaluation order, $I(q_0) \otimes [\![\tau_1]\!](s_1) \otimes [\![\tau_2]\!](s_2) \otimes \cdots \otimes [\![\tau_n]\!](s_n) \otimes F(q_n) \in Y$.

An *implicit ε-transition* is a transition τ with $\varepsilon \in L(\tau)$. $\mathcal{A}$ is *well-formed* if it has no implicit ε-transition cycles, and all of its subautomata are well-formed. Finally, the evaluation of $\mathcal{A}$ on input $s \in L(\mathcal{A})$ is given by $[\![\mathcal{A}]\!](s) := y_1 \oplus \cdots \oplus y_N \in Y$, where $y_1, \ldots, y_N$ are the weights of *all* (finitely many) distinct accepting paths on input s. As $\oplus$ is commutative and associative, this is well-defined.

Streaming automata. We recursively say that an NSA $\mathcal{A}$ is *unambiguous* if there is at most one accepting path on every input string, and each subautomaton of $\mathcal{A}$ is unambiguous. An NSA $\mathcal{A}$ is called *parallel-consistent* if, at every transition of kind (iii) labeled with $\mathrm{op}(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m)$, $L(\mathcal{A}_1) = L(\mathcal{A}_2) = \cdots = L(\mathcal{A}_m)$, and every subautomaton is parallel-consistent. A *streaming automaton (SA)* is an NSA $\mathcal{A}$ that is unambiguous and parallel-consistent. The collect operation $\oplus$ of an SA may be left off, as it is never invoked. We additionally assume that every SA is *trim*: every state has an accepting path which goes through it, and all subautomata are trim.

Checking if an NSA is an SA. Both of the two restrictions (unambiguity and parallel-consistency) can be checked efficiently. The main idea is to assign to each subautomaton $\mathcal{A}$ an *underlying NFA* $\mathrm{NFA}(\mathcal{A})$, such that $L(\mathcal{A}) = L(\mathrm{NFA}(\mathcal{A}))$, from the bottom up. Given an NSA $\mathcal{A}$, the algorithm recursively verifies that $\mathcal{A}$ is unambiguous and parallel-consistent, and also returns the NFA $\mathrm{NFA}(\mathcal{A})$ such that $L(\mathcal{A}) = L(\mathrm{NFA}(\mathcal{A}))$. Assume this has been done for all subautomata of $\mathcal{A}$. Checking parallel-consistency of a transition labeled $\mathrm{op}(\mathcal{A}_1, \ldots, \mathcal{A}_m)$ is then the equivalence problem for the unambiguous NFAs $\mathrm{NFA}(\mathcal{A}_1), \ldots, \mathrm{NFA}(\mathcal{A}_m)$; exactly this problem is solved in polynomial time by a nontrivial algorithm of [30]. Once parallel-consistency is established, we form $\mathrm{NFA}(\mathcal{A})$ by replacing each transition labeled with $\mathrm{op}(\mathcal{A}_1, \ldots, \mathcal{A}_m)$ with ε-transitions to and from a copy of $\mathrm{NFA}(\mathcal{A}_1)$. Crucially, we assume parallel-consistency in only using $\mathcal{A}_1$. This guarantees that the NFA is linear in the size of $\mathcal{A}$, and avoids the alternative of constructing an NFA for $L(\mathcal{A}_1) \cap \cdots \cap L(\mathcal{A}_m)$. The construction preserves accepting paths, so $L(\mathcal{A}) = L(\mathrm{NFA}(\mathcal{A}))$, and if one is unambiguous, both are. Finally, checking that $\mathrm{NFA}(\mathcal{A})$ is unambiguous is a reachability check in $\mathrm{NFA}(\mathcal{A}) \times \mathrm{NFA}(\mathcal{A})$.

The necessary operations for the algorithm to work lift to the symbolic setting given the decidability restrictions on the predicates. See e.g. Corollary 1 of [34].

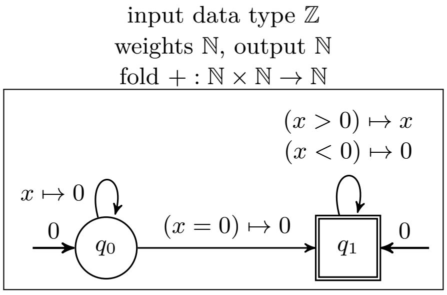
3 Evaluation Algorithm

In this section we present a space- and time-efficient evaluation algorithm for streaming automata, i.e. NSAs that are unambiguous and parallel-consistent. We will show that for such automata the space footprint of the evaluation algorithm and the time required to process each element are independent of the size of the stream and quadratic in the size of the automaton. As we will see in Section 4, both these syntactic restrictions on automata are necessary for the efficiency guarantees that we present.

Given an SA $\mathcal{A}$ and a sequence w of data items, the computation of $[\![\mathcal{A}]\!](w)$ amounts to discovering a global hierarchical path for w that may span several levels of subautomata and performing incrementally the aggregations that are prescribed by the top level and all subautomata. The crucial challenge is that the unambiguous nondeterminism of $\mathcal{A}$ requires the exploration of all possible paths *in parallel*. It is not obvious how this can be accomplished using a small amount of space, and indeed Theorem 5 in the next section shows that this is impossible in the presence of ambiguous nondeterminism. For plain NFAs or weighted automata, ambiguous nondeterminism is not an issue, because when two tokens end up at the same state during evaluation they can be merged. For streaming automata, however, such merging is not possible. The main insight is that unambiguity guarantees that no two tokens will ever end up at the same state, even at the lowest level of the automaton. As the evaluation algorithm explores each tentative path, it maintains a *stack of values* for that path, which holds the partial aggregates for the subpaths that have been discovered so far. We can think of these stacks as “execution tokens” that are updated whenever a simple transition occurs (upon consumption of a data item), and which are passed to subautomata as a way to implement the recursive definition of global accepting paths.

Before presenting the technical details, let us give a very high-level description of the evaluation algorithm and its correctness proof. First, we will introduce the notion of a *configuration*, which describes the assignment of stack tokens to the active states of the automaton. This is a generalization of configurations for NFAs, which only indicate the active states. We will define a semantics for configurations, which summarizes the accepting paths from active states as well as the computations that are performed along these paths. Then, the correctness proof of the algorithm can be reduced to establishing a simple semantic property for configurations: if C is the current configuration and C' is the configuration that the evaluation algorithm computes from C after consuming the data item d, then $[\![C']\!](w) = [\![C]\!](dw)$ for every possible suffix w. The presence of several nested levels of subautomata presents a major challenge for proving this property, since a subautomaton potentially has to compute simultaneously on several subsequences of the stream seen so far (we call these subsequences “parallel input threads”).

▶ **Example 1.** The automaton of Figure 4 computes on a stream of integers and outputs the sum of all strictly positive numbers that have occurred after the last occurrence of a 0 (or from the start if no 0 has occurred yet). We start the execution by supplying a *context stack*, which holds the partial aggregations of upper levels (if there are any), and then we supply the sequence of data items. The context stack [9] of this example is initialized by


input data type $\mathbb{Z}$
weights $\mathbb{N}$, output $\mathbb{N}$
fold $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$(x > 0) \mapsto x$
$(x < 0) \mapsto 0$
$x \mapsto 0$
0
q_0
$(x = 0) \mapsto 0$
q_1
0

start/next	configuration	input threads
[9]	$q_0 : [9, 0]$ $q_1 : [9, 0]$	$\{[9]\ \varepsilon\}$
2	$q_0 : [9, 0]$ $q_1 : [9, 2]$	$\{[9]\ 2\}$
-1	$q_0 : [9, 0]$ $q_1 : [9, 2]$	$\{[9]\ 2\ {-1}\}$
3	$q_0 : [9, 0]$ $q_1 : [9, 5]$	$\{[9]\ 2\ {-1}\ 3\}$
0	$q_0 : [9, 0]$ $q_1 : [9, 0]$	$\{[9]\ 2\ {-1}\ 3\ 0\}$
6	$q_0 : [9, 0]$ $q_1 : [9, 6]$	$\{[9]\ 2\ {-1}\ 3\ 0\ 6\}$

Figure 4 Example evaluation of an SA on one input thread.

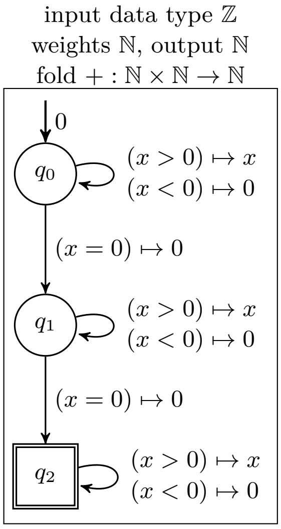
pushing the aggregate 0 onto it, and then every time an element is consumed the aggregate at the top of the stack is appropriately updated.

▶ **Example 2.** The automaton of Figure 5 computes on a stream of integers and outputs the sum of all strictly positive numbers that have occurred as long as there are exactly two occurrences of a 0. We can compute on several parallel input threads by supplying a new context stack every time we want to spawn a new thread of execution. Figure 5 shows an example execution with three different input threads. By starting a new input thread after the occurrence of a 0 we guarantee that there is at most one stack token on each state.

Epsilon transitions can be eliminated in a bottom-up fashion with a variant of the standard ε-elimination construction for weighted automata [19]. We consider in this section automata that are free of both explicit and implicit ε-transitions, and we assume w.l.o.g. that every invocation $\mathrm{op}(\mathcal{A}_1, \ldots, \mathcal{A}_k)$ has its own call state p, from which no other transition emanates.

Suppose that V is the type of all *values*. Let St be the type of all finite stacks of values, and $[]$ be the empty stack. We consider the total operation $\mathsf{push} : \mathsf{St} \times V \to \mathsf{St}$, and the partial operations $\mathsf{pop} : \mathsf{St} \rightharpoonup \mathsf{St}$ and $\mathsf{top} : \mathsf{St} \rightharpoonup V$. The operations pop and top are undefined on the empty stack. We write $s.\mathsf{push}(x)$ to denote the application of push on the stack s and the value x. Similarly, we write $s.\mathsf{pop}$ and $s.\mathsf{top}$ for the other operations. For example, we have $[].\mathsf{push}(x).\mathsf{push}(y) = [x].\mathsf{push}(y) = [x, y]$ and $[x, y].\mathsf{pop}.\mathsf{top} = [x].\mathsf{top} = x$. We write $\mathsf{St}[X_1, \ldots, X_{n-1}, X_n]$ for the type of stacks of size n whose top element is of type X_n, the next-to-top element is of type X_{n-1} and so on. We call all types of this form *bounded stack types*. If $T = \mathsf{St}[X_1, \ldots, X_n]$ then we write $T@[X_{n+1}, \ldots, X_{n+m}]$ to denote the type $\mathsf{St}[X_1, \ldots, X_n, X_{n+1}, \ldots, X_{n+m}]$. We also abbreviate $T@[X_{n+1}]$ by $T@X_{n+1}$.

The *rank* of an SA $\mathcal{A}$ is the smallest k such that $\mathcal{A} \in \mathrm{NSA}_k$, or in other words the nesting depth of the automaton. We define the notion of a configuration for an automaton by induction on its rank. For an automaton $\mathcal{A} = (Q, X, Y, \Delta, I, F, \otimes)$ of rank 0 and a bounded stack type T, an $(\mathcal{A}, T)$*-configuration* is a partial map $C : Q \rightharpoonup T@Y$; we denote the domain $\mathrm{dom}(C)$. Intuitively, the configuration describes the placement of stack tokens on some of the states of $\mathcal{A}$. For an automaton $\mathcal{A} = (Q, X, Y, \Delta, I, F, \otimes)$ of rank strictly greater than 0, an $(\mathcal{A}, T)$-configuration C is a vector consisting of a partial function $C_0 : Q \rightharpoonup T@Y$ and a subconfiguration for every subautomaton occurrence. More specifically, for every transition $(p, \mathrm{op}_i(\mathcal{A}_{i1}, \mathcal{A}_{i2}, \ldots, \mathcal{A}_{in}), q)$ of $\mathcal{A}$, the configuration C specifies an $(\mathcal{A}_{i1}, T@Y)$-configuration C_{i1} and a $(\mathcal{A}_{ij}, \mathsf{St}[])$-configuration C_{ij} for $j = 2, \ldots, n$. That is, the configuration describes the placement of stack tokens on the top-level states and specifies subconfigurations for the subautomata occurrences. We write $\mathrm{Cfg}\langle\mathcal{A}, T\rangle$ for the set of all $(\mathcal{A}, T)$-configurations.



start/next	configuration	input threads
[90]	$q_0 : [90, 0]$	{[90] ε}
3	$q_0 : [90, 3]$	{[90] 3}
−2	$q_0 : [90, 3]$	{[90] 3 −2}
0	$q_1 : [90, 3]$	{[90] 3 −2 0}
[70]	$q_0 : [70, 0]$ $q_1 : [90, 3]$	{[90] 3 −2 0, [70] ε}
2	$q_0 : [70, 2]$ $q_1 : [90, 5]$	{[90] 3 −2 0 2, [70] 2}
−1	$q_0 : [70, 2]$ $q_1 : [90, 5]$	{[90] 3 −2 0 2 −1, [70] 2 −1}
6	$q_0 : [70, 8]$ $q_1 : [90, 11]$	{[90] 3 −2 0 2 −1 6, [70] 2 −1 6}
0	$q_1 : [70, 8]$ $q_2 : [90, 11]$	{[90] 3 −2 0 2 −1 6 0, [70] 2 −1 6 0}
[30]	$q_0 : [30, 0]$ $q_1 : [70, 8]$ $q_2 : [90, 11]$	{[90] 3 −2 0 2 −1 6 0, [70] 2 −1 6 0, [30] ε}
−4	$q_0 : [30, 0]$ $q_1 : [70, 8]$ $q_2 : [90, 11]$	{[90] 3 −2 0 2 −1 6 0 −4, [70] 2 −1 6 0 −4, [30] −4}
−5	$q_0 : [30, 0]$ $q_1 : [70, 8]$ $q_2 : [90, 11]$	{[90] 3 −2 0 2 −1 6 0 −4 −5, [70] 2 −1 6 0 −4 −5, [30] −4 −5}
4	$q_0 : [30, 4]$ $q_1 : [70, 12]$ $q_2 : [90, 15]$	{[90] 3 −2 0 2 −1 6 0 −4 −5 4, [70] 2 −1 6 0 −4 −5 4, [30] −4 −5 4}

Figure 5 Example evaluation of an SA on several input threads.

For an automaton $\mathcal{A} = (Q, X, Y, \Delta, I, F, \otimes)$ and an $(\mathcal{A}, T)$-configuration C, we will define simultaneously C-paths, unambiguity of C, and the denotation $[\![C]\!] : D^* \rightharpoonup T@Y$ by induction on the rank of $\mathcal{A}$. A C-path is a path starting from the configuration C.

- ***Automaton $\mathcal{A}$ of rank 0***: A *C-path* (labeled with $d_1 d_2 \dots d_n \in D^*$) is a sequence of the following form: $q_0 \to^{\phi_1/\sigma_1}_{d_1/x_1} q_1 \to^{\phi_2/\sigma_2}_{d_2/x_2} \cdots \to^{\phi_n/\sigma_n}_{d_n/x_n} q_n$, such that $q_0 \in \mathrm{dom}(C)$ and $(q_{i-1}, \phi_i, \sigma_i, q_i) \in \Delta$ with $\phi_i(d_i) = \mathit{true}$ and $x_i = \sigma_i(d_i)$ for every $i = 1, \dots, n$. A C-path is said to be *accepting* if it ends with an accepting state. The *weight* of an accepting C-path is defined to be the value $\mathrm{fold}(y, \otimes, x_1 x_2 \dots x_n x_{n+1})$ where $y = C(q_0).\mathsf{top}$ and $x_{n+1} = F(q_n)$. The configuration C is *unambiguous* if for every label $w \in D^*$ there is at most one accepting C-path labeled with w. For an unambiguous configuration C, the denotation $[\![C]\!] : D^* \rightharpoonup T@Y$ is defined as follows: if there is an accepting C-path π labeled with w starting with the state q, then $[\![C]\!]\ w = s.\mathsf{pop}.\mathsf{push}(y)$ where $s = C(q)$ is the initial stack and y is the weight of π.
- ***Automaton $\mathcal{A}$ of rank greater than 0***: A *top-level C-path* is a sequence of top-level transitions that can be of the following two forms:

$$p \to^{\phi/\sigma}_{d/x} q \qquad \text{where } (p, \phi, \sigma, q) \in \Delta \text{ with } \phi(d) = \mathit{true} \text{ and } x = \sigma(d)$$

$$p \to^{\mathrm{op}(\mathcal{A}_1,\dots,\mathcal{A}_n)}_{w/x} q \qquad \text{where } w \neq \varepsilon \text{ and } (p, \mathrm{op}(\mathcal{A}_1, \dots, \mathcal{A}_n), q) \in \Delta \text{ with } x = \mathrm{op}([\![\mathcal{A}_1]\!]\ w, \dots, [\![\mathcal{A}_n]\!]\ w)$$

that starts with a state in the domain of C_0. Now, a *cross-level C-path* is a sequence of top-level transitions with an additional prefix called a cross-level transition:

$$\to^{\mathrm{op}(\mathcal{A}_{i1},\dots,\mathcal{A}_{in})}_{w/t} q \qquad \begin{array}{l} \text{where } w \neq \varepsilon \text{ and } (p, \mathrm{op}(\mathcal{A}_{i1}, \dots, \mathcal{A}_{in}), q) \in \Delta \text{ for some state } p \\ s_1 = [\![C_{i1}]\!](w) : T@[Y, Z_1] \text{ and } s_j = [\![C_{ij}]\!](w) : [Z_j] \text{ for } j = 2, \dots, n \\ t = s_1.\mathsf{pop}.\mathsf{pop}.\mathsf{push}(s_1.\mathsf{pop}.\mathsf{top} \otimes \mathrm{op}(z_1, \dots, z_n)) \text{ where } z_j = s_j.\mathsf{top} \end{array}$$

```
**Streaming automaton $\mathcal{A} = (Q, X, Y, \Delta, I, F, \otimes)$ of rank 0 & bounded stack type $T$.**
**state**: unambiguous $(\mathcal{A}, T)$-configuration $C : \mathrm{Cfg}\langle \mathcal{A}, T\rangle$, that is, $C : Q \rightharpoonup T@Y$

**initialize**($\mathrm{Cfg}\langle \mathcal{A}, T\rangle$ *this*) :
    $this.C := \bot$   // empty configuration

$T@Y$ **output**($\mathrm{Cfg}\langle \mathcal{A}, T\rangle$ *this*) :
    foreach $q \in Q_F$ do   // iterate over final states
        if ($this.C(q)$ is defined) then return $this.C(q)$
    return nil

**start**($\mathrm{Cfg}\langle \mathcal{A}, T\rangle$ *this*, $T\ s$) :  // precondition: $\llbracket this.C \rrbracket$ and $\langle\!\langle \mathcal{A} \rangle\!\rangle_T(s)$ are disjoint
    foreach $q \in Q_I$ do   // place token on each initial state
        // $this.C(q)$ must be undefined
        $this.C(q) := s.\mathsf{push}(I(q))$

**next**($\mathrm{Cfg}\langle \mathcal{A}, T\rangle$ *this*, $D\ d$) :
    $\mathrm{Map}\langle Q, T@Y\rangle\ C_{\text{next}} := \bot$
    foreach transition $(p, \phi, \sigma, q)$ in $\Delta$ do
        if $\phi(d) = true$ then
            $T@Y\ s := this.C(p)$   // current stack
            $Y\ y := s.\mathsf{top} \otimes \sigma(d)$   // new value
            $C_{\text{next}}(q) := s.\mathsf{pop}.\mathsf{push}(y)$   // new stack
    $this.C := C_{\text{next}}$
```

Figure 6 General evaluation algorithm for an SA of rank 0.

Such a prefix summarizes a path in the lower levels, and its annotation w/t specifies both a label $w \neq \varepsilon$ and a stack $t : T@Y$ for continuing at the top level. The label of a path is the concatenation from left to right of the strings over D that annotate the transitions. The *weight* of a top-level C-path is defined as in the 0-rank case, and the *weight* of a cross-level C-path is similar but the initial stack is specified by the first (cross-level) transition. The configuration C is *unambiguous* if it satisfies the following two conditions:

1. For every label $w \in D^*$ there is at most one accepting C-path (top-level or cross-level) labeled with w.
2. For every transition $(p, \mathrm{op}(\mathcal{A}_{i1}, \ldots, \mathcal{A}_{in}), q)$, the denotations $\llbracket C_{i1} \rrbracket$, …, $\llbracket C_{in} \rrbracket$ have equal domains.

For an unambiguous configuration C, the denotation $\llbracket C \rrbracket : D^* \rightharpoonup T@Y$ is defined as follows: if there is an accepting C-path π (top-level or cross-level) labeled with w, then $\llbracket C \rrbracket\ w = s.\mathsf{pop}.\mathsf{push}(y)$ where s is the initial stack (specified by C_0 for top-level C-paths, and by the initial transition for the cross-level C-paths) and y is the weight of π.

For an SA $\mathcal{A} = (Q, X, Y, \Delta, I, F, \otimes)$ and a bounded stack type T, we define the denotation $\langle\!\langle \mathcal{A} \rangle\!\rangle_T : T \to (D^* \rightharpoonup T@Y)$ as $\langle\!\langle \mathcal{A} \rangle\!\rangle_T\ s\ w = s.\mathsf{push}(\llbracket \mathcal{A} \rrbracket\ w)$.

Figure 6 describes the evaluation algorithm for the base case of a streaming automaton of rank 0. Observe that the algorithm specifies a procedure **next**(d) for consuming the element d, and a procedure **start**(s) for starting a new input thread given the context stack s. This generalization of being able to start several parallel input threads is necessary when the automaton is nested beneath other upper-level automata.

Figure 7 describes the evaluation algorithm for the case of a streaming automaton of rank strictly greater than 0. The interface is the same as for the base case: there are procedures **start**(s) and **next**(d). The main difference is that the algorithm in this case has to deal with the invocation transitions: every time a token is at a call state the corresponding subautomata are restarted, and every time the subautomata have output the corresponding return state is updated with the output stack.

▶ **Lemma 3.** Let $\mathcal{A} = (Q, X, Y, \Delta, I, F, \otimes)$ be an SA and T be a bounded stack type. Then:

1. Let C be an unambiguous $(\mathcal{A}, T)$-configuration and s a stack of type T so that $\llbracket C \rrbracket$ and

```
Streaming automaton $\mathcal{A} = (Q, X, Y, \Delta, I, F, \otimes)$ of rank $> 0$ & bounded stack type $T$.
state: unambiguous $(\mathcal{A}, T)$-configuration $C : \mathrm{Cfg}\langle \mathcal{A}, T \rangle$
initialize($\mathrm{Cfg}\langle \mathcal{A}, T \rangle$ $this$) :
    $this.C_0 := \bot$   // no top-level tokens
    foreach occurrence $\mathcal{A}_{ij}$ in $\Delta$ do initialize($this.C_{ij}$)
$T@Y$ output($\mathrm{Cfg}\langle \mathcal{A}, T \rangle$ $this$) :
    foreach $q \in Q_F$ do   // iterate over final states
        if ($this.C_0(q)$ is defined) then return $this.C_0(q)$
    return nil
start($\mathrm{Cfg}\langle \mathcal{A}, T \rangle$ $this$, $T$ $s$) :  // precondition: $[\![this.C]\!]$ and $\langle\!\langle \mathcal{A} \rangle\!\rangle_T(s)$ are disjoint
    $\mathrm{Map}\langle Q, T@Y \rangle$ $C_0^{\mathrm{new}} := \bot$
    foreach $q \in Q_I$ do $C_0^{\mathrm{new}}(q) := s$.push($I(q)$)   // place token on each initial state
    foreach transition $(p, \mathrm{op}(\mathcal{A}_{i1}, \ldots, \mathcal{A}_{in}), q)$ in $\Delta$ do  // restart subautomata
        if ($C_0^{\mathrm{new}}(p) \neq$ nil) then  // check if there is token on invocation state
            start($this.C_{i1}, C_0^{\mathrm{new}}(p)$); start($this.C_{ij}$, []) for all $j = 2, \ldots, n$
            $C_0^{\mathrm{new}}(p)$ := nil
    $this.C_0 := this.C_0 \sqcup C_0^{\mathrm{new}}$
next($\mathrm{Cfg}\langle \mathcal{A}, T \rangle$ $this$, $D$ $d$) :
    $\mathrm{Map}\langle Q, T@Y \rangle$ $C_0^{\mathrm{next}} := \bot$
    foreach transition $(p, \phi, \sigma, q)$ in $\Delta$ do
        if $\phi(d) = true$ then
            $T@Y$ $s := this.C(p)$   // current stack
            $Y$ $y := s$.top $\otimes\, \sigma(d)$   // new value
            $C_0^{\mathrm{next}}(q) := s$.pop.push($y$)   // new stack
    foreach transition $(p, \mathrm{op}(\mathcal{A}_{i1}, \ldots, \mathcal{A}_{in}), q)$ in $\Delta$ do  // propagate $d$ to subautomata, collect outputs
        next($this.C_{ij}, d$) for all $j = 1, 2, \ldots, n$
        $T@[Y, Z_1]$ $s_1$ := output($this.C_{i1}$)
        if ($s_1 \neq$ nil) then
            $z_1 := s_1$.top; $s_1' := s_1$.pop; $y := s_1'$.top
            $z_j$ := output($this.C_{ij}$).top for all $j = 2, \ldots, n$
            $C_0^{\mathrm{next}}(q) := s_1'$.pop.push($y \otimes \mathrm{op}(z_1, z_2, \ldots, z_n)$)
    foreach transition $(p, \mathrm{op}(\mathcal{A}_{i1}, \ldots, \mathcal{A}_{in}), q)$ in $\Delta$ do  // restart subautomata
        if ($C_0^{\mathrm{next}}(p) \neq$ nil) then  // check if there is token on invocation state
            start($this.C_{i1}, C_0^{\mathrm{next}}(p)$); start($this.C_{ij}$, []) for $j = 2, \ldots, n$
            $C_0^{\mathrm{next}}(p)$ := nil
    $this.C_0 := C_0^{\mathrm{next}}$
```

Figure 7 General evaluation algorithm for an SA of rank strictly greater than 0.

$\langle\!\langle \mathcal{A} \rangle\!\rangle_T(s)$ are disjoint. Then, the configuration **start**(C, s), as described operationally in Figure 6 and Figure 7, is unambiguous and satisfies $[\![\mathbf{start}(C, s)]\!] = [\![C]\!] \sqcup \langle\!\langle \mathcal{A} \rangle\!\rangle_T(s)$.
Notation: If f and g are partial functions with disjoint domains, the partial function $f \sqcup g$ has domain $\mathrm{dom}(f) \cup \mathrm{dom}(g)$ and agrees with both f and g.

2. Let C be an unambiguous $(\mathcal{A}, T)$-configuration and $d \in D$. Then, the configuration **next**(C, d), as described operationally in Figure 6 and Figure 7, is unambiguous and satisfies $[\![\mathbf{next}(C, d)]\!]\ w = [\![C]\!]\ dw$ for all sequences $w \in D^*$.

Lemma 3 establishes the main semantic property for configurations that is needed for proving the correctness of the evaluation algorithm.

▶ **Theorem 4.** *The streaming algorithm of Figure 6 and Figure 7 solves the evaluation problem for streaming automata. The space footprint of the algorithm and the processing time per element are independent of the length of the stream and quadratic in the size of the automaton (assuming that the data types require unit space and the operations unit time).*

The guarantees of Theorem 4 apply unconditionally to the case of constant-size types and operations (e.g., integers and floating-point numbers specified by machine architectures).

In the case of infinite data types, one may need to account for the additional complexity of computing on their unbounded values to obtain a more precise analysis. In any case, however, Theorem 4 can be understood as saying that the computational overhead of parsing the input stream and combining the intermediate results is not significant.

4 Lower Bounds

The efficient evaluation algorithm of the previous section depends crucially on the unambiguity and parallel-consistency of the automata. In fact, both these syntactic restrictions are essential for efficient evaluation. More specifically, ambiguous nondeterminism can make the streaming space complexity of evaluation linear in the size of stream. Moreover, the absence of parallel-consistency allows the encoding of unambiguous regular expressions with intersection. The streaming matching problem for such expressions requires space that is exponential in the size of the expression. These lower bounds highlight the difficulty of efficiently evaluating quantitative automata that allow for the interaction between nondeterminism and parallelism.

Consider a stream of natural numbers and the problem MinAvgSuffix for the streaming computation of the function $f(x_1 x_2 \ldots x_n) = \min_{i=1}^{n} \text{average}(x_i, x_{i+1}, \ldots, x_n)$, where $x_1 x_2 \ldots x_n$ is the stream seen so far. An NSA similar to M_2 of Figure 1 may be constructed which computes from each suffix a pair $(\text{sum}, \text{count})$, and that is nested inside an automaton dividing the components of the pair to obtain the average. Since this automaton computes MinAvgSuffix, the following theorem asserts a lower bound for the evaluation problem of NSAs with two-level nesting but without parallelism.

▶ **Theorem 5.** *Any streaming algorithm for* MinAvgSuffix *requires $\Omega(n)$ bits of memory, where n is the size of the stream seen so far.*

The following theorem states that the parallel-consistency requirement is essential for evaluation that is quadratic in the size of the automaton. The idea is based on [21].

▶ **Theorem 6.** *The evaluation problem for unambiguous streaming automata without the parallel-consistency restriction requires space exponential in the size of the automaton.*

5 Conclusion

We have considered symbolic weighted automata extended with two crucial features for expressing streaming computations: hierarchical nesting of several aggregators, and parallel execution. The following table summarizes the space complexity of the evaluation problem, where m is the size of the automaton and n the length of the data stream:

	no nesting	nesting without parallelism	consistent parallelism	general parallelism
unambiguous nondeterminism	$O(m)$	$O(m^2)$	$O(m^2)$ [Thm 4]	$O(\exp(m))$ [Thm 6]
general nondeterminism	$O(m)$	$\Omega(n)$ [Thm 5]	$\Omega(n)$	$\Omega(n)$

In *nesting without parallelism*, a transition may call a single subautomaton. *General parallelism* allows transitions with the construct $\text{op}(\mathcal{A}_1, \ldots, \mathcal{A}_m)$, which matches only those strings accepted by every $\mathcal{A}_i$. *Consistent parallelism* restricts this to require $L(\mathcal{A}_1) = \cdots = L(\mathcal{A}_m)$. These complexities assume that the types of the signature require unit space, and that the operations and predicates require unit time.

References

1 Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter 5, pages 255–300. MIT Press/Elsevier, 1990.

2 Shaull Almagor, Udi Boker, and Orna Kupferman. What's decidable about weighted automata? In Tevfik Bultan and Pao-Ann Hsiung, editors, *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA '11)*, pages 482–491. Springer Berlin Heidelberg, 2011.

3 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

4 Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '13)*, pages 13–22, 2013.

5 Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In *Proceedings of the 25th European Symposium on Programming (ESOP '16)*, pages 15–40, 2016.

6 Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '04)*, pages 286–296, 2004.

7 Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

8 Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Transactions on Computational Logic (TOCL)*, 12(4):27:1–27:26, 2011.

9 Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, C-20(2):149–153, 1971.

10 Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Quantitative languages. *ACM Transactions on Computational Logic (TOCL)*, 11(4):23, 2010.

11 Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop. Nested weighted automata. In *Proceedings of the 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '15)*, pages 725–737, 2015.

12 Krishnendu Chatterjee, Thomas A. Henzinger, and Jan Otop. Quantitative monitor automata. In Xavier Rival, editor, *Proceedings of the 23rd International Symposium on Static Analysis (SAS '16)*, pages 23–38. Springer Berlin Heidelberg, 2016.

13 S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1789–1792, 2016.

14 Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15:1–15:62, 2012.

15 Loris D'Antoni and Margus Veanes. Equivalence of extended symbolic finite transducers. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV '13)*, pages 624–639, 2013.

16 Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*, pages 541–553, 2014.

17 Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

18 Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic (TOCL)*, 10(3):16:1–16:30, 2009.

19 Manfred Droste, Werner Kuich, and Heiko Vogler, editors. *Handbook of Weighted Automata*. Springer, 2009.

20 Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

21 Martin Fürer. The complexity of the inequivalence problem for regular expressions with intersection. In *Proceedings of the 7th International Colloquium on Automata, Languages and Programming (ICALP '80)*, pages 234–245, 1980.

22 Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment*, 1(2):1379–1390, 2008.

23 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

24 Daniel Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. *International Journal of Algebra and Computation*, 4(3):405–425, 1994.

25 Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. 2017. To appear in PLDI'17.

26 Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

27 J. Ian Munro and Michael S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12(3):315–323, 1980.

28 Shanmugavelayutham Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

29 Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic (TOCL)*, 5(3):403–435, 2004.

30 Richard Edwin Stearns and Harry B. Hunt III. On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. *SIAM Journal on Computing*, 14(3):598–611, 1985.

31 Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

32 Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. NEXMark: A benchmark for queries over data streams. Available at `http://datalab.cs.pdx.edu/niagara/NEXMark/`, 2002.

33 Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST '10)*, pages 498–507. IEEE, 2010.

34 Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*, pages 137–150, 2012.