Theoretical Computer Science 807 (2020) 15–41

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Streamable regular transductions

Rajeev Alur [a], Dana Fisman [b], Konstantinos Mamouras [c,*], Mukund Raghothaman [d], Caleb Stanford [a]

[a] University of Pennsylvania, Philadelphia, PA, USA
[b] Ben-Gurion University, Be'er Sheva, Israel
[c] Rice University, Houston, TX, USA
[d] University of Southern California, Los Angeles, CA, USA

ARTICLE INFO

Article history:
Received 15 June 2018
Received in revised form 3 November 2019
Accepted 13 November 2019
Available online 19 November 2019

Keywords:
Cost Register Automata
MSO transductions
Regular functions
Quantitative automata
Weighted automata
Stream processing

ABSTRACT

Motivated by real-time monitoring and data processing applications, we develop a formal theory of quantitative queries for streaming data that can be evaluated efficiently. We consider the model of unambiguous Cost Register Automata (CRAs), which are machines that combine finite-state control (for identifying regular patterns) with a finite set of data registers (for computing numerical aggregates). The definition of CRAs is parameterized by the collection of numerical operations that can be applied to the registers. These machines give rise to the class of *streamable regular transductions* (SR), and to the class of *streamable linear regular transductions* (SLR) when the register updates are *copyless*, i.e. every register appears at most once in the right-hand-side expressions of the updates. We give a logical characterization of the class SR (resp., SLR) using MSO-definable transformations from strings to DAGs (resp., trees) without backward edges. Additionally, we establish that the two classes SR and SLR are closed under operations that are relevant for designing query languages. Finally, we study the relationship with weighted automata (WA), and show that CRAs over a suitably chosen set of operations correspond to WA, thus establishing that WA are a special case of CRAs.

1. Introduction

Finite-state automata, and the associated class of regular languages, have many appealing properties. In particular, the class of regular languages has robust, and well understood, expressiveness with multiple characterizations, and is closed under a variety of language operations. Furthermore, a finite-state automaton processes an input string in a single left-to-right pass using only constant space, in compliance with the *streaming* model of computation. This forms the basis of practical applications of automata in design and implementation of query languages for pattern matching in strings.

In a diverse range of applications such as financial tickers, network traffic monitoring, and click-streams of web usage, the core computational problem is to map a stream of data items to a numerical value. For example, suppose the monitoring software at a network router wants to compute the average number of packets per VoIP (Voice over IP) session in the incoming stream of packets. This requires detection of VoIP sessions, which can be characterized by regular patterns in the input stream, and computing numerical aggregates in a hierarchical manner, where the cost of a VoIP session is the

* Corresponding author.
E-mail addresses: alur@cis.upenn.edu (R. Alur), dana@cs.bgu.ac.il (D. Fisman), mamouras@rice.edu (K. Mamouras), raghotha@usc.edu (M. Raghothaman), castan@cis.upenn.edu (C. Stanford).

https://doi.org/10.1016/j.tcs.2019.11.018
0304-3975/

number of packets it contains, and the cost of a stream is the average of costs of VoIP sessions it contains. Recent work on *Quantitative Regular Expressions* (QREs) [1] provides a declarative query language for specifying such computation in a modular fashion, and has led to prototype implementations StreamQRE [2,3] and NetQRE [4]. These quantitative queries involve a mix of regular patterns and numerical aggregates, and crucially, all queries can be evaluated in a streaming manner. In this paper we develop an understanding of the class of streaming transductions definable by such queries: the streamable regular transductions.

A natural starting point for this study is the model of *cost register automata* (CRA) and the associated class of *regular transductions* [5]. A CRA processes a data word, that is, a sequence over $\Sigma \times D$, where Σ is a finite set of tags and D is a set of data values, and outputs a value in D that is computed using a given set of operations over D (for example, D can be the set of natural numbers, and the set of operations can include addition, minimum, and maximum). A CRA has a finite-state control that is updated based only on the tag values of the input word, and a finite set of (write-only) registers that are updated at each step using the given operations. The partial functions from $(\Sigma \times D)^*$ to D computed by the CRAs are called regular with respect to the allowed set of operations. The computation of a regular transduction can be viewed as, first mapping a data word to a tree whose nodes are labeled with the allowed operations, and then evaluating the resulting term, where the former mapping is an MSO-definable word-to-tree transformation. This gives an alternative logical characterization of the class of regular transductions, and allows us to draw upon the well developed theory of regular tree transformations [6–8].

The original CRA model is a deterministic machine that processes its input in a single pass. Its registers can hold data values as well as functions represented by terms with parameters, and the update at each step is required to be *copyless*, that is, a register can appear at most once in the right-hand-side expressions of the updates. The CRA model we study in section 2 differs in three ways. First, since our focus is on streamability, and terms can grow linearly with the size of the input stream, we require registers to hold only values. Second, we allow the finite-state control to be updated nondeterministically based on the input tags, but it is required to be *unambiguous* (that is, any accepted data word has exactly one accepting run). While it is known that when registers can hold terms, unambiguous machines are no more expressive than deterministic ones [8], we prove that when registers can hold only values that are updated in a copyless manner, unambiguous machines are strictly more expressive than their deterministic counterparts. The complexity of evaluation for unambiguous CRAs is only a constant (equal to the number of states of the machine) factor more than the deterministic ones. Finally, we consider both copyless and copyful (that is, unrestricted) updates since a CRA has a constant number of registers that hold data values and get updated by executing a constant number of operations while processing each data item. The number of bits needed to store data values can grow with the length of the input stream when the data domain is unbounded, and the exact complexity of the streaming algorithm for evaluation depends on the nature of operations and implementation details.

We call the class of transductions over data words defined by such unambiguous CRAs with registers containing only values and copyful updates as *streamable regular transductions* (SR), and the subclass when updates are required to be copyless as *streamable linear regular transductions* (SLR). Both these classes turn out to be closed under operations relevant for design of query languages. For instance, consider the `split` operation, the quantitative analog of the (unambiguous) language concatenation: given two partial transductions f and g over data words and a binary data operation op, $\texttt{split}(f, g, op)$ is defined on an input string w if it can be split uniquely into two parts $w = w_1 w_2$ such that both $f(w_1)$ and $g(w_2)$ are defined, and if so returns $op(f(w_1), g(w_2))$. Both classes SR and SLR are closed under this operation. It is worth noting that the class of general streaming algorithms with an appropriate complexity bound is not closed under this operation. That is, it is possible that both f and g can be computed in a streaming fashion using memory that is logarithmic in the size of the input string, but $\texttt{split}(f, g, op)$ requires memory that grows linearly with the length of the input string (Theorem 15). This justifies the role of regularity in the design of streaming algorithms in a modular fashion.

In section 3, we give a logical characterization of the streamable regular transductions. Recall that the class of regular transductions corresponds to MSO-definable string-to-tree transformations. We show that streamability corresponds to the requirement that in the output graph edges cannot go backwards: the output graph has multiple copies corresponding to an index position i in the input string, and the children of each such node must be copies corresponding to input positions less than or equal to i. Furthermore, allowing updates in a CRA to be copyful corresponds to allowing sharing in the output graph as long as no cycles are created. More precisely, the class SR corresponds to MSO-definable transformations from string graphs to directed acyclic graphs without backward edges, and the class SLR corresponds to MSO-definable transformations from string graphs to trees without backward edges. The proof builds on ideas developed in [6–8], but is self-contained.

In section 4, we study the relationship with *weighted automata*, an extensively studied quantitative extension of automata [9]. A weighted automaton defines a (total) function from Σ^* to a set D equipped with two operations, product and sum, that form a semiring. Such an automaton is a nondeterministic finite-state machine, each transition of which is labeled with a tag in Σ and a weight in D. The cost of a run of the automaton is the product of all the weights of the transitions in the run, and the cost associated with a string is the sum of the costs of all the corresponding runs of the automaton. If we choose the set of operations to contain only multiplication by constants, for each value in D, then we show that the two classes SR and SLR coincide, and correspond exactly to unambiguous weighted automata. If we choose the set of operations to contain multiplication by constants, for each value in D, and sum, then we show that the class SR corresponds exactly to transductions definable by weighted automata.

Section 5 contains an extensive discussion of related work, including weighted automata, register automata, and MSO-definable transformations.

2. Cost register automata

Cost Register Automata (CRAs) were introduced in [5] with the goal of providing a machine-based characterization of the class of *regular transductions* (called *regular cost functions* in [5]) from strings to costs. To capture precisely the class of transductions considered in [5], it is sufficient to restrict the updates so that they are "copyless" (each variable is used at most once on the right-hand size of a parallel update), but it is necessary to allow variables to hold terms with at most one "hole" (parameter that can be used for substitution). For the classes of transductions that we consider here, we consider CRAs that can hold only values and the copylessness restriction is dropped to accommodate word-to-DAG transformations (defined later in section 3).

We will start by introducing the semantic objects of data transductions, which represent stream transformations, and several regular combinators on these objects. Then, we will present the formal computational model of (nondeterministic, copyful) Cost Register Automata (NCRAs) and illustrate its use with several examples. We define the class of *streamable regular* (resp., *streamable linear regular*) transductions, denoted SR (resp., SLR), as the class of transductions that are computed by copyful (resp., copyless) unambiguous CRAs. The restriction of "copylessness" essentially says that it is not allowed to copy the contents of registers. A key feature of the CRA model is that it is parameterized by the operations on data values that are allowed when updating the registers.

We will also show that in the case of copyless CRAs, unambiguous machines are more expressive than their deterministic counterparts. It will then be established that both classes SR and SLR are closed under regular combinators that are straightforward analogs of the familiar regular combinators for word languages. We will conclude with a short discussion of the complexity of evaluation in the CRA model.

2.1. Data transductions and combinators

We fix a finite alphabet Σ whose elements are called *tags*, a data set D of *data values*. A *data word* is a sequence of tagged values, i.e. a word over the alphabet $\Sigma \times D$. A *data transduction* is a partial function of type

$$(\Sigma \times D)^* \rightharpoonup D.$$

For a data word $w \in (\Sigma \times D)^*$, we write $w|_\Sigma$ to denote the elementwise projection of w to the tag component. More formally,

$$(a_1, d_1)\,(a_2, d_2)\,\cdots\,(a_n, d_n)\,|_\Sigma = a_1 a_2 \cdots a_n.$$

We will only consider transductions f that satisfy the following property: for all input sequences u, v with $u|_\Sigma = v|_\Sigma$, $f(u)$ is defined iff $f(v)$ is defined. The *rate* of f is the language $\mathsf{R}(f) \subseteq \Sigma^*$ defined as follows:

$$\mathsf{R}(f) = \{\sigma \in \Sigma^* \mid \text{for every } w \text{ with } w|_\Sigma = \sigma,\ f(w) \text{ is defined}\}.$$

So, the domain of a transduction is equal to

$$\mathrm{dom}(f) = \{w \in (\Sigma \times D)^* \mid w|_\Sigma \in \mathsf{R}(f)\}.$$

A transduction $f : (\Sigma \times D)^* \rightharpoonup D$ is said to be *value-oblivious* if for all input sequences u, v with $u|_\Sigma = v|_\Sigma$ it satisfies $f(u) = f(v)$. So, a value-oblivious transduction f can be represented by a partial function $\hat{f} : \Sigma^* \rightharpoonup D$ whose domain is equal to the rate of f and which satisfies $f(w) = \hat{f}(w|_\Sigma)$ for every input sequence w.

Now, we also fix a family $\mathcal{O}$ of constants and operations that are allowed on the data set D. We write $\mathcal{O}_n$ to denote the set of n-ary operations that are contained in $\mathcal{O}$. In particular, $\mathcal{O}_0$ is the set of constants in $\mathcal{O}$. We will consider below several combinators for composing data transductions using the allowed operations. Before we give the definitions of these combinators, we need to introduce some additional notation. The operation $\odot$ is *unambiguous concatenation* of strings and $^{\circledast}$ is *unambiguous iteration*, which are defined as follows:

$$A \odot B = \{w \mid \text{there are unique } u \in A \text{ and } v \in B \text{ with } w = uv\}$$

$$A^{\circledast} = \{w \mid \text{there are unique } n \geq 0 \text{ and } w_1, \ldots, w_n \in A \text{ with } w = w_1 \cdots w_n\}$$

The *left fold* combinator $\mathsf{fold} : (B \times A \to B) \times B \times A^* \to B$ is defined by

$$\mathsf{fold}(f, b, \varepsilon) = b \text{ and}$$

$$\mathsf{fold}(f, b, a \cdot w) = \mathsf{fold}(f, f(b, a), w)$$

Table 1
Some regular combinators for data transductions: output combination, choice, quantitative concatenation, and quantitative iteration. Here, $\odot$ is *unambiguous concatenation* and $^{\circledast}$ is *unambiguous iteration*, so that quantitative concatenation (resp., quantitative iteration) is defined only on input strings where the concatenation (resp., iteration) of the underlying rates $\mathsf{R}(-)$ is unambiguous.

$$\frac{op : D^n \to D \qquad f_i : (\Sigma \times D)^* \multimap D \text{ for all } i}{op(f_1, \ldots, f_n) : (\Sigma \times D)^* \multimap D} \text{ (output combination)}$$
$$\mathsf{R}(op(f_1, \ldots, f_n)) = \mathsf{R}(f_1) \cap \cdots \cap \mathsf{R}(f_n)$$
$$op(f_1, \ldots, f_n)(w) = op(f_1(w), \ldots, f_n(w))$$

$$\frac{f, g : (\Sigma \times D)^* \multimap D}{f \texttt{ else } g : (\Sigma \times D)^* \multimap D} \text{ (choice)}$$
$$\mathsf{R}(f \texttt{ else } g) = \mathsf{R}(f) \cup \mathsf{R}(g)$$
$$(f \texttt{ else } g)(w) = \begin{cases} f(w), & \text{if } w|_\Sigma \in \mathsf{R}(f) \\ g(w), & \text{otherwise} \end{cases}$$

$$\frac{f, g : (\Sigma \times D)^* \multimap D \qquad op : D \times D \to D}{\texttt{split}(f, g, op) : (\Sigma \times D)^* \multimap D} \text{ (quantitative concatenation)}$$
$$\mathsf{R}(\texttt{split}(f, g, op)) = \mathsf{R}(f) \odot \mathsf{R}(g)$$
$$\texttt{split}(f, g, op)(uv) = op(f(u), g(v)), \text{ if } u|_\Sigma \in \mathsf{R}(f) \text{ and } v|_\Sigma \in \mathsf{R}(g)$$

$$\frac{f : (\Sigma \times D)^* \multimap D \qquad c \in D \qquad op : D \times D \to D}{\texttt{iter}(f, c, op) : (\Sigma \times D)^* \multimap D} \text{ (quantitative iteration)}$$
$$\mathsf{R}(\texttt{iter}(f, c, op)) = \mathsf{R}(f)^{\circledast}$$
$$\texttt{iter}(f, c, op)(w_1 \cdots w_n) = \mathsf{fold}(op, c, [f(w_1), \ldots, f(w_n)]),$$
$$\text{if } w_i|_\Sigma \in \mathsf{R}(f) \text{ for all } i = 1, \ldots, n$$

for all $f : B \times A \to B$, $b \in B$, $a \in A$, and $w \in A^*$. For example:

$$\mathsf{fold}(f, b, a_1) = f(b, a_1)$$
$$\mathsf{fold}(f, b, a_1a_2) = f(f(b, a_1), a_2)$$
$$\mathsf{fold}(f, b, a_1a_2a_3) = f(f(f(b, a_1), a_2), a_3)$$

Table 1 contains the typing rules and the definitions of several combinators for data transductions: output combination, choice (`else`), quantitative concatenation (`split`), and quantitative iteration (`iter`). These combinators are quantitative analogs of the familiar combinators for string languages: intersection, union, concatenation, and iteration. These regular operations on data transductions are relevant for the design of query languages for stream processing, since they enable the modular description of complex streaming computations.

The work [1] introduces a family of regular combinators that captures a class of transductions that is defined in terms of CRAs whose registers use terms with variables instead of values. The StreamQRE language of [2,3] extends the combinators of Table 1 with additional constructs (such as streaming composition and key-based partitioning) that are useful in practical stream processing applications.

2.2. Syntax and semantics of CRAs

A Cost Register Automaton (CRA) is a machine that maps data words, i.e. strings over an input alphabet of tagged values, to output values. It uses a finite-state control and a finite set of registers that contain values. At each step, the machine reads an input tag and an input value, updates its control state, and updates its registers using a parallel assignment. The definition of the machine is parameterized by the set of expressions that can be used in the assignments. For example, a CRA with increments can use multiple registers to compute alternative costs, perform updates of the form $x := y + c$, where x and y are registers and c is a constant, at each step, and commit to the cost computed in one of the registers at the end.

Let X be a set of variables, and suppose that $\mathcal{O}$ is the family of available constants and operations on the data set D. The set of *expressions* over X, denoted $\mathbb{E}_{\mathcal{O}}[X]$ is defined by the following rules:

$$\frac{\text{constant } c \in \mathcal{O}}{c : \mathbb{E}_{\mathcal{O}}[X]} \qquad \frac{\text{variable } x \in X}{x : \mathbb{E}_{\mathcal{O}}[X]}$$
$$\frac{n\text{-ary operation } op \in \mathcal{O} \qquad t_i : \mathbb{E}_{\mathcal{O}}[X] \text{ for all } i = 1, \ldots, n}{op(t_1, \ldots, t_n) : \mathbb{E}_{\mathcal{O}}[X]}$$

A function $\alpha : X \to D$ is called a *variable assignment*, and it extends uniquely to a homomorphism (i.e., operation-preserving mapping) $\hat{\alpha} : \mathbb{E}_{\mathcal{O}}[X] \to D$. An expression $t \in \mathbb{E}_{\mathcal{O}}[X]$ denotes a function $[\![t]\!] : D^X \to D$, defined as follows: for a variable assignment $\alpha : X \to D$, we put $[\![t]\!](\alpha) = \hat{\alpha}(t)$. That is, $[\![t]\!](\alpha)$ is the result of evaluating the variable-free expression that is obtained from t by replacing each variable x by $\alpha(x)$.

In the development that follows, we will also consider a special symbol val, which should be assumed to be always distinct from all variables in X. The symbol val is meant to be used as a reference to the value of the current data item. An expression $t \in \mathbb{E}_{\mathcal{O}}[X \cup \{\mathsf{val}\}]$ that potentially contains occurrences of val denotes a function $[\![t]\!] : D^X \times D \to D$, given by

$$[\![t]\!](\alpha, d) = \hat{\beta}(t) \text{ where } \beta = \alpha[\mathsf{val} \mapsto d] : X \cup \{\mathsf{val}\} \to D$$

for every variable assignment $\alpha : X \to D$ and every value $d \in D$. We use the notation $\alpha[\mathsf{val} \mapsto d]$ to mean the extension of α that maps val to d.

Definition 1 *(CRA).* A (nondeterministic, copyful) *Cost Register Automaton* (*NCRA*) over the tag alphabet Σ, data values D, and data operations $\mathcal{O}$ is a tuple

$$\mathcal{A} = (Q, X, \Delta, I, F),$$

where

- Q is a finite set of states,
- X is a finite set of registers,
- $\Delta \subseteq Q \times \Sigma \times U_{\mathcal{O}} \times Q$ is the set of transitions with $U_{\mathcal{O}}$ the set of register updates $X \to \mathbb{E}_{\mathcal{O}}[X \cup \{\mathsf{val}\}]$,
- $I : Q \rightharpoonup (X \to \mathbb{E}_{\mathcal{O}}[\emptyset])$ is the initialization function, and
- $F : Q \rightharpoonup \mathbb{E}_{\mathcal{O}}[X]$ is the finalization function.

The domain of I is the set of *initial states*, and the domain of F is the set of *final* or *accepting states*. A CRA is said to be *unambiguous* (resp., deterministic) if the underlying NFA is unambiguous (resp., deterministic). We use the abbreviations NCRA, UCRA, and DCRA to indicate that an automaton is nondeterministic, unambiguous, and deterministic respectively.

A CRA is said to be *copyless* if it satisfies the following restrictions: (1)for every transition $(p, a, \theta, q) \in \Delta$ and every register x, there is at most one occurrence of x in the list of expressions $\theta(x_1), \ldots, \theta(x_n)$, where $x_1, \ldots, x_n$ is an enumeration of X, and (2) for every final state q and every register x, there is at most one occurrence of x in the expression $F(q)$. Note that we do *not* require that the special symbol val is only used once in each expression. This is reasonable because val represents the current value in the input data word, which changes at each computation step of the automaton. Therefore, a particular value in the input data word will only be used (copied) at most a constant number of times.

A CRA is said to be *trim* if the satisfies the properties: (1) every state of the automaton is reachable from an initial state, and (2) some final state is reachable from every state.

Semantics of CRAs A CRA computes like a classical finite-state automaton, but the configuration consists of both a current state and a register assignment $X \to D$. A transition specifies the next state, as well as the update of the registers using their current values and the current data value from the input. For an input sequence $w = (a_1, d_1)(a_2, d_2) \cdots (a_n, d_n)$ in $(\Sigma \times D)^*$, we define a *w-run* in $\mathcal{A}$ to be a sequence

$$(q_0, \alpha_0) \xrightarrow{(a_1,d_1)} (q_1, \alpha_1) \xrightarrow{(a_2,d_2)} (q_2, \alpha_2) \xrightarrow{(a_3,d_3)} \cdots \xrightarrow{(a_n,d_n)} (q_n, \alpha_n)$$

with $q_i \in Q$ and $\alpha_i : X \to D$ for all i so that the following hold:

1. *Initialization*: q_0 is an initial state and $\alpha_0(x) = [\![I(q_0)(x)]\!]$ for every register $x \in X$.
2. *Transition*: for every transition $(p, \alpha) \to^{(a,d)} (q, \beta)$ of the run there is a register update function $\theta \in U_{\mathcal{O}}$ with $(p, a, \theta, q) \in \Delta$ s.t.

$$\beta(x) = [\![\theta(x)]\!](\alpha, d) \text{ for every register } x \in X.$$

3. *Finalization*: q_n is a final state.

The *value* of the run is $[\![F(q_n)]\!](\alpha_n)$. A nondeterministic CRA implements a multi-valued transduction $(\Sigma \times D)^* \to \mathcal{P}(D)$, where $\mathcal{P}$ is the powerset operator. The value of this transduction on an input sequence w is the set of values of all w-runs. An unambiguous CRA has at most one run for each input sequence, and therefore implements a transduction $(\Sigma \times D)^* \rightharpoonup D$. We write $[\![\mathcal{A}]\!]$ to denote the transduction that an automaton $\mathcal{A}$ implements.

As for classical NFAs, we can think that the computation of a CRA proceeds by placing *tokens* on the states. A token on a state q indicates that there is a run from an initial state to q after consuming the given input prefix. When a state has a

Table 2
Classes of transductions computed by unambiguous CRAs.

Unambiguous CRA	Class of transductions
copyless	SLR: Streamable Linear Regular
copyful	SR: Streamable Regular

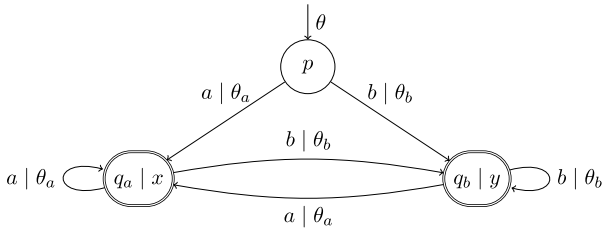
token on it we say that it is *active*. The computation starts by placing a token on each of the initial states. Every time an item is consumed, each token is transformed into a set of new tokens according to the transition relation. In the case of plain NFAs, a token carries no information. For CRAs, on the other hand, each token carries a register assignment $X \to D$. For this reason, it is possible to have more than one token per state in the case of nondeterministic (ambiguous) CRAs. For unambiguous trim CRAs, at any time during the computation there can be at most one token per state.

Definition 2 *(Streamable Regular Transductions).* Fix a tag alphabet Σ, a set D of values, and a family $\mathcal{O}$ of operations over D. We define the class of *streamable regular transductions* (over $\mathcal{O}$), denoted $\mathsf{SR}(\mathcal{O})$, to be the class of data transductions that can be computed by some UCRA (over $\mathcal{O}$). Similarly, we define the class of *streamable linear regular transductions*, denoted $\mathsf{SLR}(\mathcal{O})$, to be the class of data transductions that can be computed by some copyless UCRA. Table 2 summarizes these definitions.

2.3. Examples

We present in this subsection several examples that illustrate how CRAs compute. We consider both deterministic and unambiguous variants, and we give examples of copyless and copyful (i.e., unrestricted) register updates. Whenever convenient, we will allow the use of ε-transitions for the unambiguous variants. This is w.l.o.g. because ε-transitions can be eliminated with a straightforward variant of the ε-elimination procedure for classical finite-state automata (assuming there are no ε-cycles).

Example 3 *(Copyless DCRA).* Suppose the tag alphabet is $\Sigma = \{a, b\}$, the type of values is $D = \mathbb{N}$ (the set of nonnegative integers), and $\mathcal{O}$ consists of the constant 0 and the binary addition operation. Consider the following transduction $f : (\Sigma \times D)^* \rightharpoonup D$, which is defined on all nonempty sequences. If a sequence ends with an a-labeled value, then f outputs the sum of all a-labeled values in the sequence. Similarly, if a sequence ends with a b-labeled value, then f outputs the sum of all b-labeled values in the sequence. Then, f is implemented by the following copyless CRA:



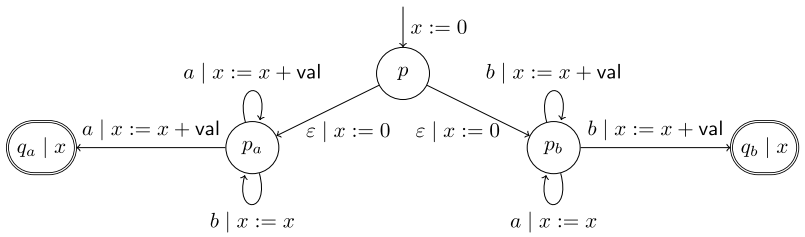
where the initialization θ and the register updates θ_a, θ_b are defined as follows:

$$\theta = \begin{cases} x := 0 \\ y := 0 \end{cases} \qquad \theta_a = \begin{cases} x := x + \mathsf{val} \\ y := y \end{cases} \qquad \theta_b = \begin{cases} x := x \\ y := y + \mathsf{val} \end{cases}$$

In the diagram, we have indicated the initialization function with the arrow labeled with θ, and the finalization function is given by the annotations x and y to the two accepting states (shown with a doubly circled border).

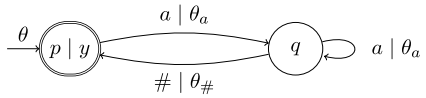
The register x holds the sum of all a-labeled values seen so far, and the register y holds the sum of all b-labeled values seen so far. The state p is active only upon initialization, and q_a (resp., q_b) is active when the input sequence ends with an a-labeled (resp., b-labeled) value. The CRA is deterministic. Moreover, it is copyless, since every register is used at most once in the right-hand side of the updates θ_a and θ_b.

Example 4 *(Copyless UCRA).* In Example 3, it was shown how to implement the transduction f using a copyless DCRA with two registers. We will show now how to implement the same transduction using a copyless UCRA with one register. Informally, the idea is to employ unambiguous nondeterminism in order to guess whether the input data word will end with the tag a or the tag b. We assume additionally that ε-transitions are allowed.



The register x holds the sum of all a-labeled values at the states p_a and q_a, and the sum of all b-labeled values at the states p_b and q_b. The automaton is copyless, since the register x is used at most once in the right-hand side of the updates.

Example 5 *(Copyless DCRA).* Suppose the tag alphabet is $\Sigma = \{a, \#\}$, the type of values is $D = \mathbb{N}$, and $\mathcal{O}$ consists of the constant 0, the binary addition operation, and max. Consider the following transduction $f : (\Sigma \times D)^* \rightharpoonup D$, whose rate is given by the regex $(a^+ \cdot \#)^*$. A maximal subsequence of the input that is of the form $aa \ldots a\#$ is called a *block*. The transduction f outputs the maximum cost over all input blocks, where the cost of a block is the sum of the a-labeled values. This is implemented by the following two-register copyless CRA:

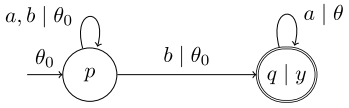


where the initialization θ and the register updates θ_a, $\theta_\#$ are given by

$$\theta = \begin{cases} x := 0 \\ y := 0 \end{cases} \qquad \theta_a = \begin{cases} x := x + \mathsf{val} \\ y := y \end{cases} \qquad \theta_\# = \begin{cases} x := 0 \\ y := \max(y, x) \end{cases}$$

The register x holds the sum of all a-labeled values in the current block, and the register y holds the maximum cost over all complete blocks seen so far. The CRA is deterministic and copyless (every register is used at most once in the right-hand side of the updates θ_a and $\theta_\#$).

Example 6 *(Copyful UCRA).* Suppose the tag alphabet is $\Sigma = \{a, b\}$, the type of values is $D = \mathbb{N}$, and $\mathcal{O}$ consists of the constant 0 and the binary operations max and $\ominus$, where the latter is natural number subtraction, sometimes called "monus", and defined as $x \ominus y = \max(x - y, 0)$ for all $x, y \in \mathbb{N}$. Consider the following transduction $f : (\Sigma \times D)^* \rightharpoonup D$, which is defined on sequences that contain at least one a-labeled value. For these sequences, the value of f is the *maximum drawdown* (the maximum loss from a peak to a trough) in the input signal after the last occurrence of a b-labeled value. The transduction f is implemented by the following copyful CRA:

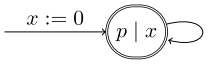


where the initialization θ_0 and the register update θ are defined as follows:

$$\theta_0 = \begin{cases} x := 0 \\ y := 0 \end{cases} \qquad \theta = \begin{cases} x := \max(x, \mathsf{val}) \\ y := \max(y, \max(x, \mathsf{val}) \ominus \mathsf{val}) \end{cases}$$

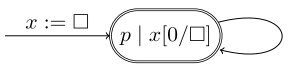
The numerical computation on the values concerns the part of the input after the last occurrence of a b-labeled value. The register x holds the maximum value, and the register y holds the maximum drawdown. The state p is always active, and q is active when at least one b-labeled value is seen in the input. The CRA is unambiguous. Moreover, it is not copyless, since the register x is used twice in the right-hand sides of the register update θ.

Example 7 *(CRAs with holes).* Suppose the tag alphabet is $\Sigma = \{a\}$, the type of values is $D = \mathbb{Q}$ (the set of rational numbers), and $\mathcal{O}$ consists of the constant 0 and the binary operation op given by $op(x, y) = \lambda \cdot x + y$, where λ is a fixed rational constant in the open interval $(0, 1)$. The transduction f maps a data word w with $w|_D = d_1 d_2 \ldots d_n \in D^*$ to $\lambda^{n-1} \cdot d_1 + \cdots + \lambda \cdot d_{n-1} + d_n$. We write $w|_D$ to denote the elementwise projection of the data word to the value component. The transduction can be computed by the following CRA:



$a \mid x := op(x, \mathsf{val}) = \lambda \cdot x + \mathsf{val}$

Define now the transduction g as $g(w_1 w_2 \dots w_n) = f(w_n \dots w_2 w_1)$. The transduction g cannot be computed by a CRA (over $\mathcal{O}$), because the registers can only hold values. However, if the registers were allowed to hold terms with a hole $\Box$ (a placeholder that can be substituted later in the computation), then g could be expressed.



$a \mid x := x[op(\Box, \mathsf{val})/\Box]$

In the machine shown above the register is meant to hold a term (with potential occurrences of $\Box$), and the operation $x[t/\Box]$ denotes the result of substituting t for $\Box$ in the term that x holds. For example, if the input is $w = (a, d_1)\,(a, d_2)\,(a, d_3)$, then the successive contents of the register x (after 3 steps of computation) are shown below:

$$\Box \qquad op(\Box, d_1) \qquad op(op(\Box, d_2), d_1) \qquad op(op(op(\Box, d_3), d_2), d_1)$$

So, the ability to use the holes $\Box$ and substitute terms into it provides additional computation power. The disadvantage of using terms is that in some cases their size can grow linearly in the size of the input.

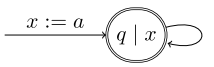
2.4. Choice of operations on data values

The definition of CRAs is parameterized by the choice of operations that are allowed when updating the registers. We now emphasize that seemingly inconsequential variations of the available operations can change what can be computed by a CRA. Suppose that $M = (D, \cdot, 1)$ is a monoid, i.e. $\cdot$ is an associative binary operation on D and 1 is a left and right identity for the $\cdot$ operation. Define $\mathcal{O}_M$ to be the family of operations that contains 1 and the binary operation $\cdot$, and $\mathcal{O}_M^{\mathsf{unary}}$ to contain the identity 1 and the unary operation $(- \cdot d)$ for every $d \in D$. (Note that we include only unary multiplication on the right, not on the left—this is not crucial for the present discussion, but will be necessary for a later result about $\mathcal{O}_M^{\mathsf{unary}}$, Theorem 31.) The following lemma applies to $\mathcal{O}_M^{\mathsf{unary}}$:

Lemma 8. *Suppose that $\mathcal{O}$ consists only of unary operations and constants. Then the class of transductions $\mathsf{SR}(\mathcal{O})$ is equal to $\mathsf{SLR}(\mathcal{O})$.*

Proof. It suffices to see that a copyful UCRA over $\mathcal{O}$ can be simulated by a copyless UCRA over $\mathcal{O}$. The key observation is that in a copyful UCRA with only unary operations, at most one register can contribute to the final output, because there is no way to combine multiple registers. So at every step of the computation, the machine can guess the register x that will contribute to the final output (using unambiguous nondeterminism), and proceed by updating only x. In particular, if the copyful UCRA over $\mathcal{O}$ has states Q and registers X, we construct a copyless UCRA over $\mathcal{O}$ with states $Q \times (X \cup \{\bot\})$ and only one register, where the single register holds the value of the register x that we guess will contribute to the final output, and the second component of the state tracks which register it is in the original machine—or $\bot$ if we guess that none of the current registers will contribute to the output. Given any UCRA over $\mathcal{O}$, this construction produces an equivalent copyless UCRA with only one register. □

The above lemma shows that if $M = (D, \cdot, 1)$ is a monoid, then the class of transductions $\mathsf{SR}(\mathcal{O}_M^{\mathsf{unary}})$ is equal to $\mathsf{SLR}(\mathcal{O}_M^{\mathsf{unary}})$. However, in general, the class of transductions $\mathsf{SR}(\mathcal{O})$ may be strictly larger than $\mathsf{SLR}(\mathcal{O})$. In particular, consider the family of operations $\mathcal{O}_M$, where $M = (D, \cdot, 1)$ is the free monoid generated by the alphabet Σ. That is, D is the set of finite words over Σ, $\cdot$ is word concatenation, and 1 is the empty word. For all transductions on $\mathsf{SLR}(\mathcal{O}_M)$, the output is of size linear in the size of the input. This property does not hold, however, for the transductions of $\mathsf{SR}(\mathcal{O}_M)$. The copyful single-state single-register CRA



$a, b \mid x := x \cdot x$

emits outputs that are of size exponential in the input. Lemma 8 and these observations establish that the precise choice of $\mathcal{O}$ can affect the class of transductions that can be computed.

Another important point about the set $\mathcal{O}$ of operations is that it can incorporate tests on values. The definition of CRAs does not allow tests on registers as guards on the transitions, which means that the finite control of the automaton depends only on the observed sequence of input tags. It is possible, however, to compute a transduction such as

$$f(w) = \begin{cases} \text{sum of } a\text{-tagged values}, & \text{if } |w|_a = |w|_b \\ \text{sum of } b\text{-tagged values}, & \text{if } |w|_a \neq |w|_b \end{cases}$$

(where $|w|_a$ is the number of occurrences of the tag a in the data word w) by including in $\mathcal{O}$ an *if-then-else* operation such as

$$\mathsf{ITE}(x, y, z, w) = \begin{cases} z, & \text{if } x = y \\ w, & \text{if } x \neq y \end{cases}$$

2.5. Unambiguity versus determinism

In this section we explore the relationship between unambiguous and deterministic CRAs. The first result is a kind of subset construction for transforming a copyful UCRA to an equivalent copyful DCRA. This means that unambiguous nondeterminism is not essential when the automata are allowed to copy values. In the case of copyless CRAs, however, we will see that unambiguity is essential. That is, there are data transductions that can be computed by copyless UCRAs but not by copyless DCRAs.

To get an intuitive understanding of why copyless DCRAs are weaker, suppose that a copyless DCRA has a register x which can contribute to the output in two different ways, depending on some regular property of the future input. Because the automaton is not allowed to copy the value of x, it cannot commit to any of the two different eventualities in order to update x appropriately. If this situation can only arise a fixed number of times in the computation, a solution can be given by blowing up the number of registers: create from the outset two copies of the register x, one for each possible eventuality and update them separately. We will see, however, that there are examples where this situation can arise an arbitrary number of times. Such examples witness the power of unambiguity in the model of copyless CRAs.

Theorem 9 *(**Copyful UCRA to Copyful DCRA**). For any $\mathcal{O}$, every copyful unambiguous CRA over $\mathcal{O}$ is equivalent to a copyful deterministic CRA over $\mathcal{O}$.*

Proof. The result holds trivially when $\mathcal{O}$ does not contain any constants. In this case, the initialization function then has to be undefined and hence the automaton can only implement the transduction that is undefined for every input, which can be implemented by a DCRA. So, we assume w.l.o.g. that $\mathcal{O}$ contains at least one constant. We will use this assumption in the rest of the proof in order to set registers to arbitrary constant values.

Let $\mathcal{A} = (Q, X, \Delta, I, F)$ be an arbitrary copyful unambiguous CRA. W.l.o.g. we assume that $\mathcal{A}$ is trim. We define the deterministic CRA $\mathcal{B}$ using a modified subset construction. The automaton $\mathcal{B}$ has registers $Q \times X$ and its state space is equal to

$$\{\Delta(\mathrm{dom}(I), w) \mid w \in \Sigma^*\},$$

where $\Delta(P, w)$ for $P \subseteq Q$ is the set of states reachable from a state in P via a path that is labeled with the word w.

The initial state of $\mathcal{B}$ is equal to the set of initial states of $\mathcal{A}$, and a register (q, x) of $\mathcal{B}$ is initialized to $I(q)(x)$ if q is initial. The initialization of the rest of the registers can be chosen arbitrarily.

A state P of $\mathcal{B}$ is final if P contains a final state q of $\mathcal{A}$. Since $\mathcal{A}$ is unambiguous, each state of $\mathcal{B}$ contains at most one final state of $\mathcal{A}$. The finalization term for P is defined to be the term that results from $F(q)$ by substituting (q, x) for every register x.

Consider now a state P of $\mathcal{B}$ and a tag $a \in \Sigma$. Then, $R = \Delta(P, a) = \bigcup_{p \in P} \Delta(p, a)$ is also a state of $\mathcal{B}$. We include a unique transition $P \to^a R$ in $\mathcal{B}$, which ensures that $\mathcal{B}$ is deterministic. It remains to specify the register update function η for this transition. Consider an arbitrary $q \in R$. Since $\mathcal{A}$ is unambiguous and trim, there is a unique $p \in P$ with $q \in \Delta(p, a)$. For a register (q, x) of $\mathcal{B}$ we put

$$\eta(q, x) = \text{the term resulting from } \theta(x) \text{ by replacing every register } y \text{ by } (p, y),$$

where θ is the register update function of the unique transition $p \to^{a,\theta} q$ in the automaton $\mathcal{A}$. The update for registers not covered by the above description can be set arbitrarily.

In order to show that $\mathcal{B}$ is equivalent to $\mathcal{A}$, it suffices to establish that on any input, the sole active state of $\mathcal{B}$ encodes a copy of the registers of $\mathcal{A}$ for every active state of $\mathcal{A}$. □

Theorem 9 establishes the expressive equivalence of copyful UCRAs and copyful DCRAs. Now, we will show (Theorem 13) that copyless UCRAs are strictly more expressive than copyless DCRAs (for a fixed family of data operations). To establish this separation result, it suffices to focus on CRAs that do not make use of the input data values in their computations. We say that a CRA is *value-oblivious* if its register update function contains no occurrence of the val symbol. In this case the denotation of the CRA is a value-oblivious transduction, since only the input tags are used and the input data values do not affect the computation. For the rest of this subsection, we will be dealing exclusively with value-oblivious CRAs. For this reason, a data transduction is taken here to be a function of type $\Sigma^* \rightharpoonup D$, and a CRA is implicitly assumed to not contain any occurrence of the val symbol. For the special case where $D = \Gamma^*$ for some finite alphabet Γ, a transduction $\Sigma^* \rightharpoonup D$ is also called a *string transduction* (the term *word transduction* is often used in the relevant literature).

As an example of this, consider the following string transduction $\mathsf{f} : \Sigma^* \rightharpoonup \Sigma^*$ with domain $\mathrm{dom}(\mathsf{f}) = \{a, b\}^+\#$, where the alphabet is $\Sigma = \{a, b, \#\}$:

$$\mathsf{f}(u\#) = \begin{cases} a^i\#, & \text{if } |u| = i \text{ and } u \text{ ends with } a \\ b^i\#, & \text{if } |u| = i \text{ and } u \text{ ends with } b \end{cases}$$

for all $u \in \{a, b\}^+$. We assume that the only data operation that is allowed is to concatenate string constants to the registers. An unambiguous CRA can compute this transduction with a single register by guessing the letter that will precede the # symbol. A deterministic CRA, on the other hand, requires at least two registers: one register for the case where the input ends in $a\#$, and one register for the case where the input ends in $b\#$.

This situation extends to the case where the computation of f is iterated several times. That is, let us define the string transduction $\mathsf{f}^k : \Sigma^* \rightharpoonup \Sigma^*$ that iterates f sequentially k times. The domain of f^k is equal to $\mathrm{dom}(\mathsf{f}^k) = (\{a, b\}^+\#)^k$, and f^k is given by

$$\mathsf{f}^k(u_1\# \cdots u_k\#) = \mathsf{f}(u_1\#) \cdots \mathsf{f}(u_k\#)$$

for all $u_1, \ldots, u_k \in \{a, b\}^+$. In order to compute f^k, an unambiguous CRA requires just one register: for every block of the input (where a block is a sequence of letters that ends with a # symbol), the automaton guesses the last letter of the block and extends the unique register with the appropriate letter. As we will show formally, however, a copyless DCRA requires at least 2^k registers: a separate register is needed for each possible combination of ending letters for the k input blocks. For example, if $k = 2$ then there are $2^k = 4$ different cases for the form of the input strings in the domain of f^k:

$$\ldots a\# \ldots a\# \qquad \ldots a\# \ldots b\# \qquad \ldots b\# \ldots a\# \qquad \ldots b\# \ldots b\#$$

A copyless DCRA can only compute f^k by using one separate register for each one of these 2^k possibilities.

In the previous paragraph, we discussed how the iteration of f a fixed number k of times creates the need for 2^k registers in the model of copyless DCRAs. An immediate consequence of that is that in this model, the iteration of f a non-constant number of times cannot be computed with a finite number of registers. This will establish the result that UCRAs are more expressive than DCRAs in the copyless case. Formally, the result is rather involved and requires a sequence of constructions on CRAs:

1. Every CRA over *unary* data operations is equivalent to the disjoint union of single-register CRAs.
2. A copyless DCRA can be modified in a way that suppresses a prefix of the output.
3. If a string transduction g requires at least k registers to be computed by a copyless DCRA, then the string transduction that computes f (defined previously) and then g in sequence requires at least $2k$ registers in this model.

Lemma 12 is the heart of the argument. It establishes that the computation of the sequential iteration f^k (defined previously) requires at least 2^k registers in the model of copyless DCRAs.

We start the technical development with Lemma 10 below, which describes a construction for simplifying CRAs that use only unary operation on the values. For this special case, it is shown that the register updates are w.l.o.g. of the form $x := \delta(c)$ or $x := \delta(x)$, where x is a register, c is a constant, and δ is a unary operation. In other words, the CRA can be decomposed into the disjoint union of several single-register CRAs.

Lemma 10 *(Separable Updates). Suppose that the collection of operations $\mathcal{O}$ consists only of unary operations and constants. Then, every copyless NCRA (resp., UCRA/DCRA) over $\mathcal{O}$ is equivalent to some copyless NCRA (resp., UCRA/DCRA) over $\mathcal{O}$ (with the same set of registers) whose register updates are of one of the following forms: $x := \delta(c)$ or $x := \delta(x)$, where c is a constant and δ is the composition of unary operations.*

Proof. We use the symbol δ in order to indicate an arbitrary composition of unary operation symbols. In a CRA $\mathcal{A} = (Q, X, \Delta, I, F)$ over $\mathcal{O}$ the register updates can be of any of the forms: $x := \delta(c)$, or $x := \delta(x)$, or $x := \delta(y)$ for $x \neq y$. We will employ a product construction in order to eliminate updates of the form $x := \delta(y)$ with $x \neq y$. If the assignment $x := \delta(y)$ is part of a parallel update, then y cannot flow into another register due to the copyless restriction.

We will describe now the construction of the CRA $\mathcal{B}$, which has the same registers as $\mathcal{A}$. The idea is that $\mathcal{B}$ computes like $\mathcal{A}$, but also maintains some extra information in the finite control that allows to "rename" the registers. A renaming function σ is a bijection on the set X of registers, and the state space of $\mathcal{B}$ is the cartesian product of the state space of $\mathcal{A}$ with the function space of bijections $X \to X$. The intuition is that when the computation is in a state (q, σ) of $\mathcal{B}$, the value that $\mathcal{A}$ would hold in the register x is contained in the register $\sigma(x)$ of $\mathcal{B}$.

If q is an initial state of $\mathcal{A}$, then (q, id_X) is an initial state of $\mathcal{B}$ with the same initialization. The renaming function id_X is the identity, which essentially means that the registers are not renamed.

If q is a final state of $\mathcal{A}$, then (q, σ) is a final state of $\mathcal{B}$. The finalization term for (q, σ) in $\mathcal{B}$ is defined as $\sigma(F(q))$.

Since $\mathcal{A}$ is copyless, every register update function θ induces *at least one* bijection $\upsilon : X \to X$, where $\upsilon(x)$ is the register that appears on the right-hand side of the assignment to x (if such a register exists). If no register is used (i.e., x is set to $\delta(c)$ for a constant c), then $\upsilon(x)$ can be set arbitrarily to one of the registers that does not appear in the right-hand side of θ at all. Thus, there will be multiple possible bijections υ, and we choose one arbitrarily. For a transition from state p to state q in $\mathcal{A}$ with label a and register update θ, we put for all renaming functions σ a transition from (p, σ) to (q, τ) in $\mathcal{B}$ with label a and register update η, where:

$$\tau = \upsilon; \sigma \qquad \eta(x) = \sigma(\theta(\tau^{-1}(x)))$$

and ; is function composition (in diagrammatic order). In other words, if $x := \delta(y)$ is a register update of $\mathcal{A}$, then we put $\tau(x) := \delta(\sigma(y))$ for the corresponding register update in $\mathcal{B}$. Moreover, notice that $\upsilon(x) = y$ and therefore $\tau(x) = (\upsilon; \sigma)(x) = \sigma(\upsilon(x)) = \sigma(y)$. So, the update in $\mathcal{B}$ has the same register appearing in both the left-hand and right-hand side.

The main claim is that the possible runs in $\mathcal{B}$ correspond to the runs of $\mathcal{A}$ modulo the variable renaming described by the renaming functions. □

For an alphabet Σ that contains the symbol #, we define the *prefix removal* operation $\partial_{\#} : \Sigma^* \to \Sigma^*$ as follows:

$$\partial_{\#}(w) = \begin{cases} \varepsilon, & \text{if \# does not appear in } w; \\ v, & \text{if } w = u\#v \text{ where } u \text{ has no occurrence of \#.} \end{cases}$$

That is, $\partial_{\#}$ removes the prefix of the string until the first occurrence of a #.

Suppose now that Σ and Γ are finite alphabets, and that Γ contains the # symbol. For a string transduction of type $\Sigma^* \rightharpoonup \Gamma^*$, we define the *output prefix removal* operation $\partial_{\#}$ as follows:

$$\partial_{\#}(f)(w) = \begin{cases} \text{undefined}, & \text{if } f(w) \text{ is undefined;} \\ \partial_{\#}(f(w)), & \text{if } f(w) \text{ is defined.} \end{cases}$$

We show below that there is a construction on CRAs that corresponds to the output prefix removal operation on string transductions.

Lemma 11 *(Remove Output Prefix). Suppose that Σ and Γ are tag alphabets with $\# \in \Gamma$, Γ^* is the set of values, and $\mathcal{O}$ consists of the constant ε and the unary operations $(- \cdot a)$ for every $a \in \Gamma$. If the string transduction $f : \Sigma^* \rightharpoonup \Gamma^*$ is implementable by a UCRA (resp., DCRA) over $\mathcal{O}$, then $\partial_{\#}(f)$ is implementable by a UCRA (resp. DCRA) with the same number of registers.*

Proof. Suppose that the UCRA $\mathcal{A} = (Q, X, \Delta, I, F)$ implements f. We will describe a UCRA $\mathcal{B}$ that implements $\partial_{\#}(f)$. The idea is to employ a product construction that records some additional information in the finite control: whether the first # symbol would have already been added to a register in the execution of $\mathcal{A}$ or not. Additionally, we modify the updates of $\mathcal{A}$ so that no tags are appended to registers until after the first # would have been appended in the execution of $\mathcal{A}$.

The state space of $\mathcal{B}$ is $Q \times (X \to \{0, 1\})$. A run of $\mathcal{A}$ that ends in a configuration $(q, \alpha) \in Q \times (X \to \Gamma^*)$ corresponds to a run of $\mathcal{B}$ that ends in a configuration $(q, \rho, \beta) \in Q \times (X \to \{0, 1\}) \times (X \to \Gamma^*)$ that satisfies

$$\beta(x) = \partial_{\#}(\alpha(x)) \quad \text{and} \quad \rho(x) = \begin{cases} 0, & \text{if } \alpha(x) \text{ does not contain \#} \\ 1, & \text{if } \alpha(x) \text{ contains \#} \end{cases}$$

for every register x. Notice that the construction that is outlined here preserves determinism. □

Lemma 12. *Suppose that $\Sigma = \{a, b, \#\}$ is the tag alphabet, Σ^* is the set of values, and $\mathcal{O}$ consists of the constant ε and the unary operations $(- \cdot a)$, $(- \cdot b)$, and $(- \cdot \#)$. Let $f : \Sigma^* \rightharpoonup \Sigma^*$ be a string transduction, and define the string transduction $g : \Sigma^* \rightharpoonup \Sigma^*$ as follows:*

$$\mathrm{dom}(g) = \{a, b\}^+ \cdot \# \cdot \mathrm{dom}(f)$$

$$g(u\#v) = \begin{cases} a^i \cdot \# \cdot f(v) \text{ where } i = |u|, & \text{if } u \text{ ends with } a \\ b^i \cdot \# \cdot f(v) \text{ where } i = |u|, & \text{if } u \text{ ends with } b \end{cases}$$

for all $u \in \{a, b\}^+$ and $v \in \mathrm{dom}(f)$. If the transduction g can be implemented by a copyless DCRA with k registers, then f can be implemented by a copyless DCRA with at most $k/2$ registers.

Proof. Let $\mathcal{A}$ be a DCRA with k registers that implements g. By Lemma 10, we can assume without loss of generality that every register update of $\mathcal{A}$ is of the form $x := u$ or $x := x \cdot u$ for some $u \in \Sigma^*$. Consider all register updates that are enabled when the automaton $\mathcal{A}$ consumes the input sequences $\{a, b\}^*$. Let R_a (resp., R_b) be the set of registers that eventually only contain and are extended with a tags (resp., b tags) in these updates. The registers that are not among R_a and R_b cannot contribute to the output of the computation for sufficiently long input prefixes before the first occurrence of #. One of R_a, R_b is of cardinality at most $k/2$. W.l.o.g. we can assume that $|R_a|$ is at most $k/2$. Then, consider the product $\mathcal{B}$ of $\mathcal{A}$ with an automaton that accepts $a^+ \cdot \# \cdot \mathrm{dom}(f)$. In the automaton $\mathcal{B}$, only the registers of R_a contribute nontrivially to the output, so we can assume that R_a is the register set of $\mathcal{B}$. Notice that $\mathcal{B}$ computes the transduction $h : \Sigma^* \rightharpoonup \Sigma^*$, given by

$$\mathrm{dom}(h) = a^+ \cdot \# \cdot \mathrm{dom}(f) \qquad h(a^i\#v) = a^i \cdot \# \cdot f(v)$$

for all $i \geq 1$ and $v \in \mathrm{dom}(f)$. Now, Lemma 11 implies that a DCRA $\mathcal{C}$ with at most $k/2$ registers can compute the transduction $k : \Sigma^* \rightharpoonup \Sigma^*$, given by

$$\mathrm{dom}(k) = a^+ \cdot \# \cdot \mathrm{dom}(f) \qquad k(a^i \# v) = f(v)$$

for all $i \geq 1$ and $v \in \mathrm{dom}(f)$. Let (q, α) be the configuration of $\mathcal{C}$ after consuming $a\#$. The DCRA that results from $\mathcal{C}$ by setting the start state to be q and the initialization function to α computes the transduction f and has at most $k/2$ registers. □

Theorem 13. *For some $\mathcal{O}$, there is a string transduction that can be computed by a copyless UCRA over $\mathcal{O}$ but not by a copyless DCRA over $\mathcal{O}$.*

Proof. Suppose that $\Sigma = \{a, b, \#\}$ is the tag alphabet, Σ^* is the set of values, and $\mathcal{O}$ consists of the constant ε and the unary operations $(- \cdot a)$, $(- \cdot b)$, and $(- \cdot \#)$. Suppose that $\Sigma = \{a, b, \#\}$ and $f : \Sigma^* \rightharpoonup \Sigma^*$ is the string transduction with domain $\mathrm{dom}(f) = \{a, b\}^+ \cdot \#$ that is defined as follows:

$$f(u\#) = \begin{cases} a^i\#, & \text{if } |u| = i \text{ and } u \text{ ends with } a \\ b^i\#, & \text{if } |u| = i \text{ and } u \text{ ends with } b \end{cases}$$

for all $u \in \{a, b\}^+$. Any copyless DCRA that computes f needs at least two registers. Indeed, if there is only one register, then on any input string it either contains an a or it contains a b or it contains neither; if we take the input string to be sufficiently large and then read in a b or an a, there is not enough information stored in the state to know the value of $|u| = i$ and output b^i or a^i. Define f^n to be the string transduction with domain $\mathrm{dom}(f^n) = (\{a, b\}^+ \cdot \#)^n$ and

$$f^n(u_1\#u_2\# \cdots u_n\#) = f(u_1\#)f(u_2\#) \cdots f(u_n\#)$$

where every u_i is an element of $\{a, b\}^+$. A consequence of Lemma 12 is that any copyless DCRA that computes f^n requires at least 2^n registers.

Now, consider the string transduction f^* with domain $(\{a, b\}^+ \cdot \#)^*$ defined as $f^*(u_1\# \cdots u_n\#) = f^n(u_1\# \cdots u_n\#)$, where $u_i \in \{a, b\}^+$ for all $1 \leq i \leq n$. The transduction f^* can be computed by a copyless UCRA that guesses the last letter of each block (maximal segment of letters that ends with a # symbol).

Assume for the sake of contradiction that there exists a copyless DCRA $\mathcal{A}$ with $m \geq 1$ registers that computes f^*. The product of $\mathcal{A}$ with a DFA that accepts the language $(\{a, b\}^+ \cdot \#)^k$ has m registers and computes the transduction f^k. But we have already discussed that such a DCRA would need at least 2^k registers, hence taking k large enough that $2^k > m$ gives us the desired contradiction. □

2.6. Closure under regular combinators

We will now show that both classes SR (streamable regular) and SLR (streamable linear regular) of transductions are closed under operations that are relevant for the design of streaming query languages. In particular, we will consider the collection of the regular combinators of Table 1, which were defined for data transductions.

Theorem 14. *For every $\mathcal{O}$, the classes of transductions* SR *and* SLR *(see Definition 2 and Table 2) are closed under the regular combinators of Table 1.*

Proof. The proof of the theorem relies on constructions that are variants of well-known constructions on classical finite-state automata. We will present the proofs only for the cases $op(f, g)$ and $\mathtt{iter}(f, c, op)$ and leave the rest of the cases as exercises for the reader. For the cases that we present, we will only give the main ingredients of the construction.

Suppose that $\mathcal{A}$ and $\mathcal{B}$ are CRAs. We will construct a CRA $\mathcal{C}$ that computes $op([\![\mathcal{A}]\!], [\![\mathcal{B}]\!])$, where op is a binary operation on D. The state space of $\mathcal{C}$ is the product of the state space of $\mathcal{A}$ and the state space of $\mathcal{B}$. The set of registers of $\mathcal{C}$ is the union of the registers of $\mathcal{A}$ and the registers of $\mathcal{B}$. The automaton $\mathcal{C}$ simulates the parallel execution of $\mathcal{A}$ and $\mathcal{B}$, where the registers of the two machines are updated separately. When a state (p, q) of $\mathcal{C}$ consists of a final state p of $\mathcal{A}$ and a final state q of $\mathcal{B}$, the finalization term is defined so that it combines the output values of the two automata $\mathcal{A}$ and $\mathcal{B}$. This construction preserves the "copylessness" property: if the automata $\mathcal{A}$ and $\mathcal{B}$ are copyless, then so is $\mathcal{C}$.

Suppose that $\mathcal{A}$ is a CRA, c is a data value, and op is a binary operation on values. We will construct a CRA $\mathcal{C}$ that computes $\mathtt{iter}([\![\mathcal{A}]\!], c, op)$. W.l.o.g. we assume that we can use ε-transitions, and therefore $\mathcal{A}$ has a unique initial state and a unique final state. We will also assume that $\varepsilon \notin \mathsf{R}([\![A]\!])$, because this special case can be handled separately to avoid ε-cycles. We will describe the construction of $\mathcal{C}$ as a sequence of modifications on $\mathcal{A}$.

- First, add a new register y to $\mathcal{A}$ that is meant to hold the aggregate value computed in the iteration. The register y stays unchanged in the old transitions of $\mathcal{A}$.

- Add an ε-transition from the unique final state to the unique initial state. In this transition, update the aggregate value of y to reflect one more step of the iteration.
- Add a new initial and final state for computing the initial aggregate c.
- Since the NCRA that has resulted from the previous steps could potentially be *ambiguous*, take the product with a (register-free) DFA that accepts the language $\mathsf{R}([\![A]\!])^{\circledast}$.

The last step of the construction removes the ambiguity. So, the resulting CRA is unambiguous and computes the transduction $\texttt{iter}([\![\mathcal{A}]\!], c, op)$. This construction preserves the property of "copylessness": if $\mathcal{A}$ is copyless then so is $\mathcal{C}$. □

In contrast to Theorem 14, we can show that the class of *all streaming algorithms* with an appropriate streaming complexity bound is not closed under the regular combinators of Table 1. For instance, the following theorem shows that this is true if by "streaming algorithm" we require that the algorithm must use only $O(\log(n))$ space in the length of the input stream. This justifies the role of regularity in guaranteeing modular composition, as we mentioned in the introduction.

Theorem 15. *Let $\Sigma = \{a, b\}$ and $D = \mathbb{N}$. There exist data transductions f and g (*not *in* SR*) such that f and g can be implemented as streaming algorithms which use at most $O(\log(n))$ space in the length of the input stream (n), but $\texttt{split}(f, g, op)$ cannot.*

Proof. This example is adapted from Theorem 5.3 in [10]. Define $f, g : (\Sigma \times \mathbb{N})^* \rightharpoonup \mathbb{N}$ as follows: $f(w) = 1$ if $|w|_a = 2 \cdot |w|_b$, and undefined otherwise, and $g(w) = 1$ if $|w|_a = |w|_b$, and undefined otherwise. Here $|w|_a$ is the number of occurrences of a in w. Note that f and g are value-oblivious, and neither $\mathsf{R}(f)$ nor $\mathsf{R}(g)$ is regular. The rate of the transduction $\texttt{split}(f, g, op)$ (the choice of op doesn't matter) is the concatenation of these two languages, which is an unambiguous concatenation.

Both f and g can be implemented by maintaining two counters for the number of a's and the number of b's seen so far. This requires $O(\log n)$ space. On the other hand, any streaming algorithm that computes $h = \texttt{split}(f, g, op)$ requires at least a linear number of bits. Specifically, consider the behavior of such a streaming algorithm on inputs of the form $a(aab|aba)^n ab$. On these 2^n distinct inputs, each of length $3n + 3$, the streaming algorithm would have to reach 2^n different internal states, because the inputs are pairwise distinguished by reading in a further string of the form b^k. For example, the inputs $w = a(aab)ab$ and $w' = a(aba)ab$ are distinguished by reading in b, and in general if we have two inputs

$$w \in a(aab|aba)^{n-k}(aab)(aab|aba)^{k-1}ab$$
$$\text{and} \quad w' \in a(aab|aba)^{n-k}(aba)(aab|aba)^{k-1}ab,$$

they are distinguished by reading in the string b^k.

Thus on inputs of size $3n + 3 = \Theta(n)$ the streaming algorithm requires at least n bits to store the state, which is not $O(\log n)$. □

Finally, we note as another contrast to Theorem 14 that there are interesting and natural operations on data transductions under which SR is closed, but SLR is not closed. In particular, let $f : (\Sigma \times D)^* \rightharpoonup D$ be a data transduction such that $\mathsf{R}(f) = \Sigma^*$, i.e. f is *total*, let c be a constant in $\mathcal{O}$ and op a binary operation in $\mathcal{O}$, and define the *prefix sum* of f by

$$\texttt{prefix-sum}(f, c, op) : (\Sigma \times D)^* \rightharpoonup D$$
$$\mathsf{R}(\texttt{prefix-sum}(f, c, op)) = \Sigma^*$$
$$\texttt{prefix-sum}(f, c, op)(w) = \mathsf{fold}(op, c, [f(\varepsilon), f(w_1), \ldots, f(w_1 w_2 \ldots w_n)]),$$

where $w = w_1, \ldots, w_n$ and each $w_i \in \Sigma \times D$. Then:

Theorem 16. *For every $\mathcal{O}$, the class of data transductions $\mathsf{SR}(\mathcal{O})$ is closed under the operation $\texttt{prefix-sum}$. For some $\mathcal{O}$, the class $\mathsf{SLR}(\mathcal{O})$ is not closed under $\texttt{prefix-sum}$.*

Proof. First we show SR is closed. Let $c \in \mathcal{O}$ be a constant, $op \in \mathcal{O}$ a binary operation, and $\mathcal{A} = (Q, X, \Delta, I, F)$ a copyful UCRA which denotes a total transduction. By Theorem 9, assume w.l.o.g. that $\mathcal{A}$ is deterministic; additionally, assume it is trim. Then, we construct a DCRA for $\texttt{prefix-sum}(f, c, op)$ by adding one register total, and updating it along every transition by applying op to the previous total and the new output of $\mathcal{A}$. We always know the new output of $\mathcal{A}$, because in a deterministic, trim automaton which denotes a total transduction, every state is final.

Next we show that SLR is *not* closed. Let $\Sigma = \{a\}$ be the tag alphabet, let $M = (D, \cdot, \varepsilon)$ be the free monoid generated by Σ, and consider the family of operations $\mathcal{O}_M$. Let $f : (\Sigma \times D)^* \rightharpoonup D$ be the transduction which outputs the same number of a's in the input: $f((a, d_1) \ldots (a, d_n)) = a^n$. Then $f \in \mathsf{SLR}(\mathcal{O})$ because it can be implemented by a copyless UCRA. However, $\texttt{prefix-sum}(f, \varepsilon, \cdot)$ has the property that on inputs of length n, the output $a^{n(n+1)/2}$ is of quadratic size. As we have observed in the comments following Lemma 8, the output of a copyless transduction over $\mathcal{O}_M$ must be linear in the input, so this means that $\texttt{prefix-sum}(f, \varepsilon, \cdot) \notin \mathsf{SLR}(\mathcal{O})$. □

2.7. Complexity of evaluation

We are interested in evaluation in the streaming model of computation. The main resources in this model are: 1) the total space required by the algorithm, and 2) the time needed to process each data item. Both these resources are given as a function of the length of the stream that has been seen so far. Ideally, both these key parameters should be constant or logarithmic in the length of the stream.

Suppose that $\mathcal{A} = (Q, X, \Delta, I, F)$ is a trim UCRA. The computation of the semantics of $\mathcal{A}$ in the streaming model of computation is similar to the deterministic simulation of classical NFAs, in which the set of *active states* is maintained and updated at every step of the computation. The difference for CRAs is that we need to maintain for every active state q a register assignment $\alpha_q : X \to D$. Unambiguity guarantees that at every step of the computation we need to store *at most one* register assignment for every state. This property does not hold for ambiguous machines, as there are several different paths that could lead to the same state. So, the total number of values that are needed to store is bounded above by $|Q| \cdot |X|$. For deterministic CRAs, the corresponding upper bound is $|X|$. Analogous upper bounds can be given on the *number of operations* in $\mathcal{O}$ that need to be executed (roughly, the time needed) per element of the input stream. For both copyless and copyful cases, these bounds are the same.

However, a more precise accounting of the used computational resources requires looking at the number of *bits* that are needed to store each value during the computation (rather than just the number of values that need to be stored and the number of operations that need to be computed). In order to make such finer distinctions, we need to take into account the operations that are allowed in $\mathcal{O}$. For example, suppose that the values are the natural numbers and that $\mathcal{O}$ contains numerical constants, binary addition, max and min. In the copyless case, the contents of registers can grow only linearly in the length of the input and therefore a logarithmic numbers of bits is required for each register. In the copyful case, however, even if we restrict $\mathcal{O}$ to contain only 0 and binary addition, the contents of registers can grow exponentially and therefore a linear number of bits may be needed.

In some applications, the data values range over a finite domain (e.g., 32-bit integers, 64-bit floating-point numbers, and so on), and therefore the space needed to store each register is constant. Moreover, the operations on such bounded values require constant time. In this setting, the space and time-per-element resource requirements for evaluating CRAs is constant.

3. MSO-definable transductions

In this section, we give logical characterizations of the classes of transductions defined by the Cost Register Automata of Definition 1:

1. streamable regular or SR (computed by copyful UCRAs),
2. streamable linear regular or SLR (computed by copyless UCRAs).

These logical characterizations are in terms of word-to-DAG transformations which are given by formulas written in the language of monadic second-order logic (MSO).

The MSO-definable transductions that we consider here specify output graphs *with no backward edges*. This restriction for the output DAGs to only have forward edges is crucial so that the function is implementable by an automaton which maintains values (as opposed to maintaining terms which may have "holes", as in [11]). Indeed, we will show that the forward-only MSO-definable transductions are exactly the transformations that are implementable by the value-based UCRAs of Definition 1.

This class of transductions has previously arisen in the study of attribute grammars [6] under the name of *direction preserving* MSO graph transductions (page 11 in [6]). An important difference here compared to [6] is the characterization of the transduction classes in terms of automata ([6] uses attribute grammars) whose efficient evaluation in the streaming model of computation is obvious. Another technical difference is that the transductions in the present paper involve data values, and we use a special symbol val and the position of a val-labeled vertex in order to define the semantics of an MSO transduction. Our proof is self-contained, and exhibits a pleasant uniformity due to the translation from forward-only MSO transductions to *unambiguous* CRAs. Recall that unambiguity is essential in order to capture the class SLR in the copyless CRA model (Theorem 13). In the word-to-DAG case, the output graphs can have more than one edge emanating from a vertex, which corresponds to copying a register in the machine model. In the word-to-tree case, each vertex of the output graph has at most one edge emanating from it, which implies that registers are used in a copyless manner in the machine model.

An *MSO transduction* specifies the output DAG using MSO formulas for the vertices and edges that are interpreted over the entire input stream. The challenge is to convert this global description of the output into local computation rules for the automaton. In particular, the existence of a vertex or edge depends on a regular property of the *future* of the input (regular look-ahead). To deal with this regular look-ahead, we make use of *unambiguously nondeterministic* CRAs. The unambiguous nondeterminism conveniently allows us to make guesses regarding a regular property of the future of the input.

The crucial intermediate result for obtaining equivalence between MSO transductions and CRAs (Corollary 29 and Theorem 30) asserts that every MSO transduction can be transformed equivalently into a *single-step* MSO transduction, namely one where the edges of the output DAG extend over at most one input element (Lemma 23). To implement a single-step

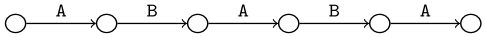
MSO transduction it is then sufficient to maintain as many data registers as the maximum number of DAG vertices that appear in some position. As before, w.l.o.g. we allow the use of ε-transitions for unambiguous CRAs.

Let Σ be the finite alphabet of tags, D a (possibly infinite) set of data values, and $\mathcal{O}$ a collection of constants and operations on D. Recall that we only consider transductions $f : (\Sigma \times D)^* \rightharpoonup D$ that satisfy the following condition: whether $f(w)$ is defined or not depends only on the sequence of tags $w|_\Sigma$.

3.1. MSO transductions

An MSO-definable graph transduction [12] specifies a partial function from labeled graphs to labeled graphs. The nodes, edges and labels of the output graph are described in terms of MSO formulas over the nodes, edges and labels of the input graph.

The tag projection of a data word $w = (a_1, d_1)\,(a_2, d_2)\ldots(a_n, d_n)$ of length n over $\Sigma \times D$ can be represented as a labeled directed graph $\mathsf{G}(w)$ with $n+1$ vertices $\{0, 1, \ldots, n\}$ and n Σ-labeled edges $(i-1, a_i, i)$ for all $i = 1, \ldots, n$. For example, we visualize the graph $\mathsf{G}(w)$ of the word $w = (\mathtt{A}, 1)\,(\mathtt{B}, 2)\,(\mathtt{A}, 3)\,(\mathtt{B}, 4)\,(\mathtt{A}, 1)$ over the alphabet with tags $\Sigma = \{\mathtt{A}, \mathtt{B}\}$ and values that are natural numbers as follows:



Note that the values in the input word are not represented in the graph $\mathsf{G}(w)$; they will be represented later using the special symbol val.

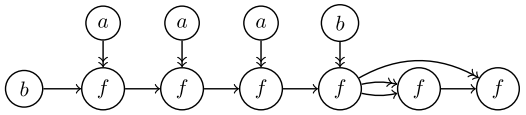
A graph $\mathsf{G}(w)$ specifies a relational structure whose domain is the set of vertices and which has a binary predicate $\mathsf{edge}_\mathtt{a}^w$ for every tag $\mathtt{a} \in \Sigma$ consisting of the $\mathtt{a}$-labeled edges. We use the *monadic second-order language* $\mathsf{MSO}(\Sigma)$ to express properties of words when represented as graphs. For every tag $\mathtt{a}$ of the alphabet Σ, the language has a binary predicate symbol $\mathsf{edge}_\mathtt{a}$. Informally, $\mathsf{edge}_\mathtt{a}(x, y)$ means that there is an edge from x to y labeled with the tag $\mathtt{a}$. The language also contains the equality predicate $=$ (i.e., $x = y$ means that the vertices x and y are equal) and the containment predicate $\in$ (i.e., $x \in X$ means that the vertex x belongs to the set of vertices X). The formulas of $\mathsf{MSO}(\Sigma)$ are built from atomic formulas using the Boolean connectives, first-order quantification (over vertices of $\mathsf{G}(w)$), and second-order monadic quantification (over sets of vertices of $\mathsf{G}(w)$). We do not need to include $\leq$, because we will shortly introduce it as an abbreviation.

Note that we use binary predicates $\mathsf{edge}_\mathtt{a}(x, y)$ to encode tags, instead of the more commonly used unary predicates [13]. That is, $\mathsf{G}(w)$ has labeled edges instead of labeled vertices. The reason for this encoding is that we want to allow transductions that map the empty input word into some output graph (as is done, for example, in page 232 of [14]). In any case, this difference is inconsequential for the definable class of functions.

We write $\mathsf{Word}(\Sigma)$ for the class of graph structures over the signature of $\mathsf{MSO}(\Sigma)$ that correspond to finite words over Σ. If ϕ is a sentence of $\mathsf{MSO}(\Sigma)$ and w is a word over Σ, then we write $w \models \phi$ to denote that the graph $\mathsf{G}(w)$ satisfies ϕ. The abbreviation $\mathsf{edge}(x, y) := \bigvee_{\mathtt{a}\in\Sigma} \mathsf{edge}_\mathtt{a}(x, y)$ says there is an edge from x to y. Moreover, we encode $\leq$ as follows:

$$(x \leq y) := \forall X.x \in X \wedge (\forall u \forall v.u \in X \wedge \mathsf{edge}(u, v) \rightarrow v \in X) \rightarrow y \in X.$$

A *ranked alphabet* Γ is a finite set with assignment of a natural number to each element, called its *arity*. A *directed acyclic graph* (DAG) over a set of ranked symbols Γ has Γ-labeled vertices, a single sink vertex (its *root*), and edges indexed between 1 and the maximum arity, such that the graph respects the arity of the symbols of Γ. For example, suppose $\Gamma = \{a, b, f\}$, where a and b are constants (arity 0) and f is binary (arity 2). The following graph is a DAG over Γ, where we use arrows $\rightarrow$ for the first argument of f, and arrows $\rightarrow\!\!\!\rightarrow$ for the second argument of f:



Definition 17 *(Forward-only word-to-DAG MSO transductions).* Let $\Gamma = \mathcal{O} \cup \{\mathsf{val}\}$ be the ranked alphabet that includes the operations $\mathcal{O}$ on the data values D as well as a fresh constant symbol val. Let i_{max} denote the maximum arity of a symbol in Γ. A *forward-only MSO-definable word-to-DAG transduction* over $\mathcal{O}$ from $(\Sigma \times D)^*$ to D consists of:

1. An $\mathsf{MSO}(\Sigma)$ sentence ϕ_{dom}, called the *domain sentence*.
2. A finite *copy set* C.
3. For every copy element $c \in C$ and every symbol $\gamma \in \Gamma$, an $\mathsf{MSO}(\Sigma)$ *vertex formula* $\phi_\gamma^c(x)$ with one free variable x.
4. For all copy elements $c, d \in C$ and for all argument indices i from 1 to i_{max}, an $\mathsf{MSO}(\Sigma)$ *edge formula* (for the i-th argument) $\psi_i^{c,d}(x, y)$ with two free variables x and y s.t. $\models \forall x \forall y.\, \psi_i^{c,d}(x, y) \rightarrow x \leq y$.

such that all of the well-formedness conditions shown in Observation 18 hold. These conditions assert that the formulas define a properly structured DAG over the symbols Γ in the output structure (which consists of C copies of the vertices of $\mathsf{G}(w)$).

Semantics: The MSO transduction *denotes* a partial function $\tau : (\Sigma \times D)^* \rightharpoonup D$ whose domain is the set of models $[\![\phi_{\text{dom}}]\!]$ of the domain sentence. For every data word w satisfying $[\![\phi_{\text{dom}}]\!]$, the transduction specifies a single DAG over the symbols $\mathcal{O} \cup \{\text{val}\}$. Every symbol in $\mathcal{O}$ is interpreted as an operation on D, and val is a special symbol for the data value of the input string at the corresponding position. The DAG therefore can be evaluated to a value in D, which is defined to be the value of τ on w.

Note that while the formulas above *define* an output graph, they are only logically *interpreted over* an input graph $\mathsf{G}(w)$. This is why they define a single output structure (not, e.g., a set of output structures or a relation between input and output structures). The well-formedness conditions of Observation 18 then guarantee this output structure has the correct form. These conditions are stated as MSO formulas, but it is not actually necessary for our results that they are expressible in MSO; rather, we only use the fact that the output of every forward-only MSO transduction is a properly-structured DAG, and we have baked this condition into the definition of such transductions. However, the fact that they are $\mathsf{MSO}(\Sigma)$ formulas means the conditions are effectively checkable (decidable).

Observation 18 *(Well-formedness).* We list here the well-formedness conditions for the forward-only MSO-definable transductions of Definition 17. Note that all conjunctions and disjunctions ($\bigvee, \bigwedge$) are finite, and there are finitely many formulas overall. Logical validity is over all structures in $\mathsf{Word}(\Sigma)$. I.e., for every input graph $\mathsf{G}(w) \in \mathsf{Word}(\Sigma)$ (and all interpretations of the free variables), the formula should be true.

(1) **At most one vertex label**: $\bigwedge_{\gamma\in\Gamma}\bigwedge_{\delta\in\Gamma,\delta\neq\gamma} \neg\phi^c_\gamma(x) \vee \neg\phi^c_\delta(x)$ is valid for every $c \in C$.

(2) **Edges connect active vertices**: $\psi_i^{c,d}(x,y) \to \phi^c(x) \wedge \phi^d(y)$ is valid for all $c, d \in C$ and $i = 1, 2, \ldots, i_{\text{max}}$, where $\phi^c(x) := \bigvee_{\gamma\in\Gamma}\phi^c_\gamma(x)$ abbreviates that the c-vertex at position x is active. Informally, a vertex (c, x) of the output graph (where c is an element of the copy set C and x is a position) is *active* if it is labeled with some symbol, i.e. $\phi^c(x)$ is true (when interpreted in the input graph).

(3) **(Non)existence of output**: The formulas $\phi_{\text{dom}} \to \exists x \bigvee_{c\in C}\phi^c(x)$ and $\neg\phi_{\text{dom}} \to \forall x \bigwedge_{c\in C}\neg\phi^c(x)$ are valid.

(4) **No local cycle at any position**: For every potential cycle $c_1 \to^{i_1} c_2 \to^{i_2} \cdots \to^{i_{k-1}} c_k \to^{i_k} c_1$ of length $k \leq |C|$, where $c_j \in C$ and $1 \leq i_j \leq i_{\text{max}}$ are edge labels, the following formula is valid:

$$\neg\exists x.\, \psi_{i_1}^{c_1,c_2}(x,x) \wedge \psi_{i_2}^{c_2,c_3}(x,x) \wedge \cdots \wedge \psi_{i_k}^{c_k,c_1}(x,x).$$

(5) **The arity of labels is respected**: For every symbol $\gamma \in \Gamma$ of arity n, every $i = 1, \ldots, n$, and every $c \in C$, the following formula is valid:

$$\phi^d_\gamma(y) \to \exists x.\, \Big(\bigvee\nolimits_c(\psi_i^{c,d}(x,y) \wedge \bigwedge\nolimits_{\hat{c}\neq c}\neg\psi_i^{\hat{c},d}(x,y)) \wedge \forall\hat{x}\bigwedge\nolimits_{\hat{c}}(\psi_i^{\hat{c},d}(\hat{x},y) \to \hat{x} = x)\Big).$$

(6) **Global uniqueness of sink**: For all $c \in C$, the formula

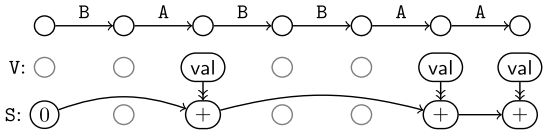
$$\mathsf{sink}_c(x) \to \bigwedge\nolimits_{d\neq c}\neg\mathsf{sink}_c(x) \wedge \forall y.(y \neq x \to \bigwedge\nolimits_d \neg\mathsf{sink}_d(y))$$

is valid, where $\mathsf{sink}_c(x) := \forall y \bigwedge_d \bigwedge_i \neg\psi_i^{c,d}(x,y)$ abbreviates that there is no outgoing edge from copy c at position x.

(7) **No value at first position**: $\mathsf{first}(x) \to \bigwedge_c \neg\phi^c_{\text{val}}(x)$ is valid, where $\mathsf{first}(x) := \forall y.\, x \leq y$ abbreviates the very first position. The constant symbol val, which is meant to refer to the input data value immediately before the point of occurrence, should not appear in this position.

Example 19 *(MSO Transduction).*

The tag alphabet is $\Sigma = \{\mathtt{A}, \mathtt{B}\}$ and the data domain D is the set of natural numbers. The collection $\mathcal{O}$ of operations on D contains the constant 0 and addition.



We will describe the transduction that sums up all data values that are tagged with $\mathtt{A}$. The copy set is taken to be $C = \{\mathtt{V}, \mathtt{S}\}$. The domain sentence is $\phi_{\text{dom}} = \text{true}$ and the vertex formulas are:

$$\phi^{\mathtt{V}}_{\text{val}}(x) = \phi^{\mathtt{S}}_{+}(x) = \mathsf{edge}_{\mathtt{A}}(x{-}1, x) \qquad \phi^{\mathtt{S}}_0(x) = \mathsf{first}(x)$$

where $\mathsf{edge}_{\mathtt{A}}(x-1,x)$ is notation for $\exists y.\mathsf{edge}_{\mathtt{A}}(y,x)$. The edge formulas are given as follows:

Table 3
Classes of MSO-definable transductions with forward-only edges.

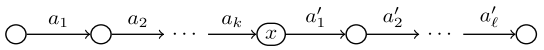
Output graph	Class of transductions
Tree	Streamable Linear Regular
DAG	Streamable Regular

$$\psi_2^{\mathrm{V},\mathrm{S}}(x,y) = (x = y) \wedge \mathsf{edge}_{\mathbb{A}}(x{-}1, x)$$
$$\psi_1^{\mathrm{S},\mathrm{S}}(x,y) = (\mathsf{first}(x) \vee \mathsf{edge}_{\mathbb{A}}(x{-}1, x)) \wedge \mathsf{edge}_{\mathbb{A}}(y{-}1, y) \wedge$$
$$\forall z.\, x < z < y \rightarrow \neg \mathsf{edge}_{\mathbb{A}}(z{-}1, z)$$

All omitted vertex and edge formulas are equal to false.

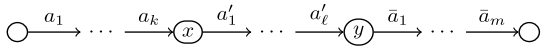
As we will see later, the data transductions that are definable by forward-only word-to-DAG MSO-transductions are exactly the streamable regular transductions. Similarly, the data transductions that are definable by forward-only word-to-tree MSO-transductions will be shown to be exactly the streamable linear regular transductions. See Table 3. The condition that the output graph is a tree can be guaranteed by requiring that every vertex of the graph has at most one outgoing edge. This condition can be expressed by an MSO formula, and therefore its validity can be effectively checked.

Definition 20 *(Models of Open Formulas).* Let $\phi(x)$ be an $\mathsf{MSO}(\Sigma)$ formula with one free individual variable. The set of *models* of $\phi(x)$, denoted $[\![\phi(x)]\!]$,



is defined to be the set of pairs $(u, v) \in \Sigma^* \times \Sigma^*$ such that the word $u \cdot v$ satisfies $\phi(x)$ under the variable assignment $x \mapsto |u|$.

Let $\psi(x, y)$ be an $\mathsf{MSO}(\Sigma)$ formula with two free individual variables such that $\models \psi(x, y) \rightarrow x \leq y$. The set of *models* of $\psi(x, y)$, denoted $[\![\psi(x, y)]\!]$,



is defined to be the set of triples $(u, v, w) \in \Sigma^* \times \Sigma^* \times \Sigma^*$ such that the word $u \cdot v \cdot w$ satisfies $\psi(x, y)$ under the variable assignment $x \mapsto |u|$, $y \mapsto |uv|$.

Lemma 21 *([14], Lemma 5). Let $\phi(x)$ be an $\mathsf{MSO}(\Sigma)$ formula with one free individual variable. The set of models $[\![\phi(x)]\!]$ is a finite disjoint union of sets of the form $[\![r_1]\!] \times [\![r_2]\!]$, where r_1 and r_2 are regular expressions over Σ.*

Similarly, suppose that $\psi(x, y)$ is an $\mathsf{MSO}(\Sigma)$ formula with two free individual variables such that $\models \psi(x, y) \rightarrow x \leq y$. The set of models $[\![\psi(x, y)]\!]$ is a finite disjoint union of sets of the form $[\![r_1]\!] \times [\![r_2]\!] \times [\![r_3]\!]$, where r_1, r_2 and r_3 are regular expressions over Σ.

Proof sketch. This is a folklore result for regular languages (see, for example, Lemma 5 of [14]). For the first part of the lemma, the idea is to construct the DFA $\mathcal{A}$ that accepts the models of $\phi(x)$. The annotation with x is performed by the automaton by allowing some states to be labeled with x. For every state q of $\mathcal{A}$ that is labeled with x, we include the regular expression for the set $L_q \cdot R_q$, where L_q is the set of strings that lead from the initial state to q, and R_q is the set of strings that lead from q to a final state of $\mathcal{A}$. The construction for the second part of the lemma is similar. □

Using Lemma 21 an MSO transduction can be transformed into a set of rules that are expressed equivalently with regular expressions instead of MSO formulas. The domain sentence ϕ_{dom} can be given as a regular expression. A vertex formula $\phi_\gamma^c(x)$ can be given as a finite set of pairwise disjoint *vertex rules* of the form

$$\text{label } \gamma \text{ at } c : r_1;\ r_2,$$

where r_1 and r_2 are regular expressions. The rule specifies a prefix (from start to x) with r_1 and a suffix (from x to end) with r_2 so that the copy c of the output DAG is active at position x. Similarly, an edge formula $\psi_i^{c,d}(x, y)$ of the transduction can be presented as a finite set of pairwise disjoint *edge rules* of the forms

$$\text{edge } c \rightarrow_i d : r_1;\ r_2;\ r_3 \qquad \text{[forward edges]}$$

where r_1, r_2 and r_3 are regular expressions. The rule specifies a prefix (from start to x) with r_1, a middle part (from x to y) with r_2, and a suffix (from y to end) with r_3 so that the output DAG has an edge from copy c at position x to copy d at position y. We have required in the definition that edges be forward-only, so there are no models in which y comes before x in $\psi_i^{c,d}(x, y)$.

Example 22. Using Lemma 21 we will present the MSO transduction of Example 19 equivalently with regular expressions. We have: $\phi_{\text{dom}} = \Sigma^*$ and

$$\phi^{\mathrm{V}}_{\mathsf{val}} = \phi^{\mathrm{S}}_{+} = (\Sigma^*\mathtt{A});\ \Sigma^* \qquad \psi^{\mathrm{V},\mathrm{S}}_{2} = (\Sigma^*\mathtt{A});\ \varepsilon;\ \Sigma^*$$
$$\phi^{\mathrm{S}}_{0} = \varepsilon;\ \Sigma^* \qquad \psi^{\mathrm{S},\mathrm{S}}_{1} = (\varepsilon \cup \Sigma^*\mathtt{A});\ (\mathtt{B}^*\mathtt{A});\ \Sigma^*$$

The vertex rule ϕ^{S}_{+}, for example, says that the copy $\mathtt{S}$ is active with label $+$ at every position immediately after the occurrence of a letter $\mathtt{A}$ in the input string. Note that here, each formula becomes a single rule; in general, each formula will become a set of pairwise disjoint rules.

A MSO transduction is said to be ***single-step*** if the edges that it specifies are restricted so that they refer to positions that are at most 1 apart, that is: the formula

$$\psi_i^{c,d}(x, y) \to (x = y) \vee \mathsf{edge}(x, y)$$

is valid for all $c, d \in C$ and every argument index i.

Lemma 23 *(Single-Step Decomposition). Let Σ be a finite tag alphabet, and D be a set of data values with operations $\mathcal{O}$. A forward-only MSO-definable word-to-DAG transduction over $\Sigma, D, \mathcal{O}$ can be transformed into an equivalent single-step transduction over $\Sigma, D, \mathcal{O} \cup \{\mathrm{id}\}$, where $\mathrm{id} : D \to D$ is the identity function on D.*

Proof. Suppose that the transduction is presented in the form using regular expressions instead of MSO formulas, as described earlier. We will show how to eliminate rules that violate the single-step constraint by replacing them with rules that are either ε-transitions or single-letter transitions.

Consider a rule $\psi_i^{c,d} = c \to_i d : r_1;\ r_2;\ r_3$ for i-edges from copy c to copy d which is not an ε-transition, i.e. $r_2 \neq \varepsilon$. The idea is to decompose an edge specified by the rule, which spans a substring matching r_2, by simulating the execution of an automaton for the expression r_2. Take the minimal DFA for r_2 and prune it into an equivalent partial DFA so that every state has a path to some final state. Suppose this automaton is $(Q, \Sigma, \Delta, q_0, F)$. For a state $q \in Q$ define the regular set

$$L_q = \{w \in \Sigma^* \mid \Delta(q_0, w) = q\},$$

that is, L_q is the set of strings over Σ that lead from the initial state q_0 to q. Moreover, define the regular set

$$R_q = \{w \in \Sigma^* \mid \Delta(q, w) \in F\},$$

that is, R_q is the set of strings over Σ that lead from q to some final state. Notice that $L_q \cdot R_q \subseteq r_2$ for every state q, and additionally $r_2 = \bigcup\{L_q \mid q \in F\}$.

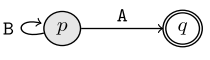
For every state q of the automaton, we introduce a fresh copy element v_q. We can replace the rule $\psi_i^{c,d}$ by the following set of vertex and edge rules:

$$\text{label id at } v_q : (r_1 \cdot L_q);\ (R_q \cdot r_3)$$
$$\text{edge } c \to_1 v_I : r_1;\ \varepsilon;\ (r_2 \cdot r_3)$$
$$\text{edge } v_q \to_i d : (r_1 \cdot L_q);\ \varepsilon;\ r_3, \text{ for every } q \in F$$
$$\text{edge } v_p \to_1 v_q : (r_1 \cdot L_p);\ \mathtt{a};\ (R_q \cdot r_3), \text{ for every } (p, \mathtt{a}, q) \in \Delta.$$

Since the automaton is deterministic, we are guaranteed that all of these rules are disjoint, and moreover that the modified transduction satisfies all the well-formedness conditions. In addition, it preserves the property of the output DAG being a tree.

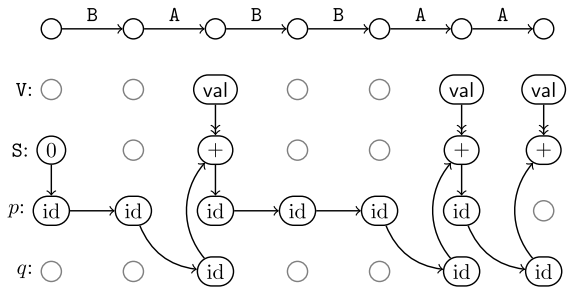
The construction is iterated until there is no edge rule left that violates the single-step condition. The resulting transduction can then be equivalently presented using MSO formulas. □

Example 24 *(Single-step).* We will apply now the construction of the proof of Lemma 23 to the transduction of Example 19. Using the presentation of Example 22 we see that the only edge rule that violates the single-step condition is $\psi_1^{\mathrm{S},\mathrm{S}}$. Now, the pruned partial DFA for $\mathtt{B}^*\mathtt{A}$ is the following:



$$L_p = \mathtt{B}^* \quad R_p = \mathtt{B}^*\mathtt{A}$$
$$L_q = \mathtt{B}^*\mathtt{A} \quad R_q = \varepsilon$$

The initial state is indicated with a gray background. We introduce fresh copy elements p and q, one for every state of the automaton for $\mathtt{B}^*\mathtt{A}$. So, the new copy set is $C' = \{\mathrm{V}, \mathrm{S}, p, q\}$. The following figure illustrates how the edges defined by $\psi_1^{\mathrm{S},\mathrm{S}}$ are decomposed into ε-transitions and single-letter transitions.



So, the transduction is specified by the following rules:

$$\phi_{\text{dom}} = \Sigma^* \qquad \psi_2^{\mathrm{V},\mathrm{S}} = (\Sigma^*\mathtt{A});\ \varepsilon;\ \Sigma^*$$
$$\phi_{\mathsf{val}}^{\mathrm{V}} = (\Sigma^*\mathtt{A});\ \Sigma^* \qquad \psi_1^{\mathrm{S},p} = (\varepsilon \cup \Sigma^*\mathtt{A});\ \varepsilon;\ \mathtt{B}^*\mathtt{A}\Sigma^*$$
$$\phi_0^{\mathrm{S}} = \varepsilon;\ \Sigma^* \qquad \psi_1^{p,p} = (\varepsilon \cup \Sigma^*\mathtt{A})\mathtt{B}^*;\ \mathtt{B};\ \mathtt{B}^*\mathtt{A}\Sigma^*$$
$$\phi_+^{\mathrm{S}} = (\Sigma^*\mathtt{A});\ \Sigma^* \qquad \psi_1^{p,q} = (\varepsilon \cup \Sigma^*\mathtt{A})\mathtt{B}^*;\ \mathtt{A};\ \Sigma^*$$
$$\phi_{\mathsf{id}}^{p} = (\varepsilon \cup \Sigma^*\mathtt{A})\mathtt{B}^*;\ \mathtt{B}^*\mathtt{A}\Sigma^* \qquad \psi_1^{q,\mathrm{S}} = (\varepsilon \cup \Sigma^*\mathtt{A})\mathtt{B}^*\mathtt{A};\ \varepsilon;\ \Sigma^*$$
$$\phi_{\mathsf{id}}^{q} = (\varepsilon \cup \Sigma^*\mathtt{A})\mathtt{B}^*\mathtt{A};\ \Sigma^*$$

The ε-transitions $\psi_1^{\mathrm{S},p}$, $\psi^{q,\mathrm{S}}$ and the single-letter transitions $\psi_1^{p,p}$, $\psi_1^{q,\mathrm{S}}$ expand the edges of $\psi_1^{\mathrm{S},\mathrm{S}}$ according to the automaton for $\mathtt{B}^*\mathtt{A}$.

Theorem 25. *For any $\mathcal{O}$, every single-step word-to-dag (resp., word-to-tree) forward-only MSO transduction over $\mathcal{O}$ can be implemented by a copyful (resp., copyless) UCRA over $\mathcal{O}$.*

Proof. Suppose we are given an MSO transduction, presented in the form with regular expressions instead of formulas. Given an input string, a position x in the string and a copy element c, in order to decide whether the vertex (c, x) of the output DAG is labeled with f we have to check if the suffix (from x to the end) satisfies the regular expression r_2, where $\phi_f^c = r_1;\ r_2$ is a vertex rule of the transduction. Similarly, to decide whether an edge $c \to_i d$ emanates from copy c at position x we have to check if the suffix (from x to end) satisfies the expressions $r_2 \cdot r_3$, where $\psi_i^{c,d} = r_1;\ r_2;\ r_3$ is an edge rule of the transduction. Finally, to decide whether an edge $d \to_i c$ has the vertex (c, x) as destination we have to check if the suffix (from x to end) satisfies the expression r_3, where $\psi_i^{d,c} = r_1;\ r_2;\ r_3$ is an edge rule of the transduction.

For a regular expression r over Σ, we write $[\![r]\!] \subseteq \Sigma^*$ to mean the regular set that r denotes. The *derivative* of a language $L \subseteq \Sigma^*$ w.r.t. a word $u \in \Sigma^*$ is the language $D_u(L) = \{v \in \Sigma^* \mid uv \in L\}$. Observe that $D_v(D_u(L)) = D_{uv}(L)$ for all $L \subseteq \Sigma^*$ and $u, v \in \Sigma^*$.

Let $\mathcal{R}$ be the collection of all regular sets needed for these suffix tests, that is, $[\![r_2]\!]$ from every vertex rule $r_1;\ r_2$, as well as both $[\![r_2 \cdot r_3]\!]$ and $[\![r_3]\!]$ from every edge rule $r_1;\ r_2;\ r_3$. Now, define $\mathcal{R}'$ to be the smallest collection of regular sets that contains $\mathcal{R}$ and is closed under derivatives. It can be seen that $\mathcal{R}' = \bigcup_{L \in \mathcal{R}} \mathcal{R}'_L$, where $\mathcal{R}'_L$ is the collection of regular sets obtained in the following way: take the collection of all derivatives of L, which correspond to the unique minimal automaton for L. Finally, define $\mathsf{At}_{\mathcal{R}}$ to be the collection of atoms (nonempty minimal elements) of the Boolean algebra generated by $\mathcal{R}'$. The elements of $\mathsf{At}_{\mathcal{R}}$ partition Σ^* (they are pairwise disjoint and their union is equal to Σ^*), and they define an *unambiguous NFA*, where the language accepted from every state $T \in \mathsf{At}_{\mathcal{R}}$ is equal to T. To see how the transitions of the NFA $\mathsf{At}_{\mathcal{R}}$ are calculated, consider a state (atom) $T = L_1 \cap {\sim}L_2 \cap L_3$ where each of L_i is an element of $\mathcal{R}'$. For a letter $\mathtt{a} \in \Sigma$ the derivative of T is

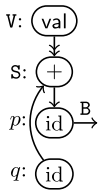
$$\begin{aligned} D_{\mathtt{a}}(T) &= D_{\mathtt{a}}(L_1) \cap D_{\mathtt{a}}({\sim}L_2) \cap D_{\mathtt{a}}(L_3) \\ &= D_{\mathtt{a}}(L_1) \cap {\sim}D_{\mathtt{a}}(L_2) \cap D_{\mathtt{a}}(L_3), \end{aligned}$$

and it is partitioned by some atoms $T'_1, \ldots, T'_\ell$ of $\mathsf{At}_{\mathcal{R}}$ because every $D_{\mathtt{a}}(L_i)$ belongs to $\mathcal{R}'$. So, $T \to^{\mathtt{a}} T'_i$ are all the transitions from T on letter $\mathtt{a}$ in the unambiguous NFA $\mathsf{At}_{\mathcal{R}}$. The automaton $\mathsf{At}_{\mathcal{R}}$ thus constructed is called the *future automaton* for the transduction, because every state specifies the possible future suffixes.

Similarly, the application of the rules of the MSO transduction depends on regular properties of the input prefix seen so far. For a vertex rule $r_1;\ r_2$, an ε-transition rule $r_1;\ \varepsilon;\ r_2$, and a letter transition rule $r_1;\ \mathtt{a};\ r_2$, we need to check whether the input prefix satisfies the pattern r_1. To perform all these tests we construct the *past automaton*, which simulates the parallel execution of automata for all these prefix patterns. The past automaton is always a DFA.

The product of the future automaton and the past automaton is an unambiguous NFA, which we call the *future-past automaton* where every state specifies exactly which rules of the MSO transduction apply at the current position. So, every

state of the future-past automaton specifies precisely the *shape* of a position: which copy elements are active, which ε-edges are enabled, and which outgoing letter-edges are enabled. A state only needs to specify values for variables whose values are going to be used in the next step of the computation. The expressions for these outgoing variables can be read off immediately from the shape. For example, suppose the state T' of the future-past automaton specifies the following active vertices and edges:



We know that any incoming transition $T \to T'$ from some state T (e.g., states q_1, q_3 and q_6 in Fig. 2) must specify a value for the variable x_q (for the copy q), because it has to be used. The state T' has to specify a new value for x_p, because the only outgoing letter-edge points to the copy p. So, the variable update for the transition $T \to T'$ must be $x_p := x_q + \mathrm{val}$ (see edges in Fig. 3). If a state is initial, then it must be possible to calculate the values for the variables without relying on any incoming value from a previous state (e.g., states q_1, q_8 and q_9 in Fig. 2). If the shape has an active copy without an outgoing edge, then its value is a potential final value for the computation (e.g., copy $\mathtt{S}$ in states q_4 and q_9 of Fig. 2). For the particular case where no vertex is active (e.g., state q_7 in Fig. 2), we have to propagate the final value of the computation (the variable where it is stored is uniquely determined).

It is easy to check that if the transduction is word-to-tree (that is, every vertex in a shape has at most one outgoing edge), then the constructed CRA is copyless. □

Example 26 *(Future Automaton).* For the single-step MSO transduction of Example 24 we will construct the future automaton, as described in the proof of Theorem 25. The collection of regular sets needed for suffix tests is:

$$\mathcal{R} = \{\Sigma^*,\ \mathtt{B}^*\mathtt{A}\Sigma^*,\ \mathtt{B}^+\mathtt{A}\Sigma^*,\ \mathtt{A}\Sigma^*\}.$$

After closing under derivatives we obtain the collection

$$\mathcal{R}' = \{\Sigma^*,\ \mathtt{B}^*\mathtt{A}\Sigma^*,\ \mathtt{B}^+\mathtt{A}\Sigma^*,\ \mathtt{A}\Sigma^*,\ \emptyset\}.$$

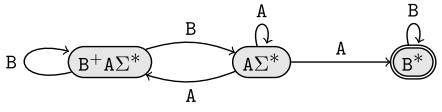
The set $\mathtt{B}^*\mathtt{A}\Sigma^*$ consists of all words that contain at least one $\mathtt{A}$, so its complement is $\mathtt{B}^*$. The set $\mathtt{B}^+\mathtt{A}\Sigma^*$ consists of all words that start with $\mathtt{B}$ and contain at least one $\mathtt{A}$, so its complement is $\mathtt{A}\Sigma^* \cup \mathtt{B}^*$. The complement of $\mathtt{A}\Sigma^*$ is $\varepsilon \cup \mathtt{B}\Sigma^*$. The elements of $\mathsf{At}_{\mathcal{R}}$ are therefore the minimal nonempty intersections of elements from $\mathtt{B}^*\mathtt{A}\Sigma^*$, $\mathtt{B}^+\mathtt{A}\Sigma^*$, $\mathtt{A}\Sigma^*$, $\mathtt{B}^*$, and $\varepsilon \cup \mathtt{B}\Sigma^*$. We claim that the Boolean atoms generated by this list are

$$\mathsf{At}_{\mathcal{R}} = \{\mathtt{A}\Sigma^*,\ \mathtt{B}^+\mathtt{A}\Sigma^*,\ \mathtt{B}^*\}.$$

Indeed, $\mathsf{At}_{\mathcal{R}}$ partitions Σ^* and moreover we have:

$$\mathtt{B}^*\mathtt{A}\Sigma^* = \mathtt{A}\Sigma^* \cup \mathtt{B}^+\mathtt{A}\Sigma^*$$
$$\varepsilon \cup \mathtt{B}\Sigma^* = \mathtt{B}^* \cup \mathtt{B}^+\mathtt{A}\Sigma^*$$

The unambiguous NFA $\mathsf{At}_{\mathcal{R}}$ is illustrated below:



All the states are marked as initial (with the gray background) because the domain of the transduction is Σ^*.

Example 27 *(Past Automaton).* The expression $(\varepsilon \cup \Sigma^*\mathtt{A})\mathtt{B}^*$ is equivalent to Σ^*, hence the rules of the single-step transduction of Example 24 can be simplified:

$$\phi^{\mathrm{V}}_{\mathsf{val}} = \Sigma^*\mathtt{A};\ \Sigma^* \qquad \psi^{\mathrm{V},\mathrm{S}}_{2} = \Sigma^*\mathtt{A};\ \varepsilon;\ \Sigma^*$$
$$\phi^{\mathrm{S}}_{0} = \varepsilon;\ \Sigma^* \qquad \psi^{\mathrm{S},p}_{1} = (\varepsilon \cup \Sigma^*\mathtt{A});\ \varepsilon;\ \mathtt{B}^*\mathtt{A}\Sigma^*$$
$$\phi^{\mathrm{S}}_{+} = \Sigma^*\mathtt{A};\ \Sigma^* \qquad \psi^{p,p}_{1} = \Sigma^*;\ \mathtt{B};\ \mathtt{B}^*\mathtt{A}\Sigma^*$$
$$\phi^{p}_{\mathsf{id}} = \Sigma^*;\ \mathtt{B}^*\mathtt{A}\Sigma^* \qquad \psi^{p,q}_{1} = \Sigma^*;\ \mathtt{A};\ \Sigma^*$$
$$\phi^{q}_{\mathsf{id}} = \Sigma^*\mathtt{A};\ \Sigma^* \qquad \psi^{q,\mathrm{S}}_{1} = \Sigma^*\mathtt{A};\ \varepsilon;\ \Sigma^*$$

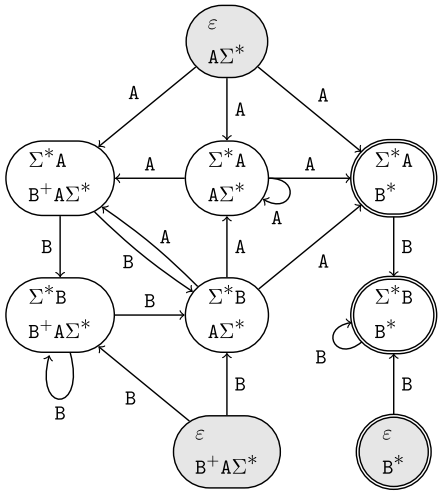
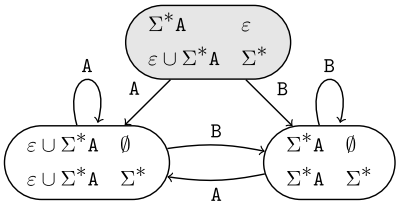
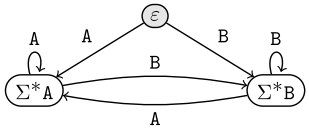


Fig. 1. The future-past automaton for the single-step transduction of Example 24.

The regular expressions $\Sigma^*\mathtt{A}$, ε, Σ^*, $\varepsilon \cup \Sigma^*\mathtt{A}$ are relevant for checking whether the input prefix seen so far satisfies the left-hand expression of a rule. The automaton that simulates the parallel execution of DFAs for these expressions is the following:



The labels are computed by taking derivatives of the expressions. It is more convenient to label a state q with the regular expression that denotes the set of strings that lead from the initial state to q.



The label of each state determines if the automaton accepts for each one of the "past" expressions $\Sigma^*\mathtt{A}$, ε, Σ^*, $\varepsilon \cup \Sigma^*\mathtt{A}$.

Example 28 *(Future-Past Automaton, CRA).* The product of the future automaton from Example 26 and the past automaton from Example 27 is the *future-past* automaton of Fig. 1. Every state q is labeled with L_q (top), the set of strings that lead from some initial state to q, and with R_q (bottom), the set of strings that lead from q to some final state. The purpose of these labelings is that they specify exactly which rules of the transaction are enabled at a particular state. In Fig. 2 every state of the future-past automaton is annotated with a shape that indicates the active copies and the edges (ε-edges and letter-edges) that are enabled. From these shapes we can immediately read off the variable updates for the transitions, and the initialization of the variables in the initial states. The resulting CRA is shown in Fig. 3. Every state q_i of the CRA is annotated with the set of variables that have to be set when the computation reaches q_i.

Corollary 29. *For any $\mathcal{O}$, every forward-only word-to-DAG (resp., word-to-tree) MSO transduction over $\mathcal{O}$ can be implemented by a copyful (resp., copyless) UCRA over $\mathcal{O}$.*

Proof. Suppose we are given an MSO transduction. By virtue of Lemma 21 we can present it equivalently using regular expressions instead of MSO formulas. Lemma 23 says that the transduction can be transformed into an equivalent one, with potentially more copy elements, in which edges are either ε-edges or single-letter edges. From this modified transduction we can construct an unambiguous CRA that computes the transduction, as shown in Theorem 25. □

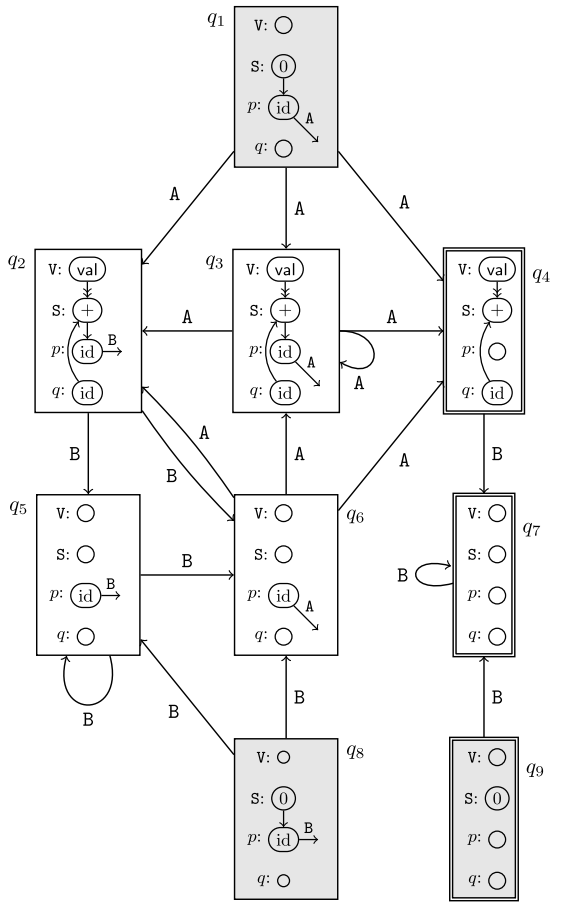


Fig. 2. The future-past automaton for the single-step transduction of Example 24, annotated with "shapes" that show the enabled rules.

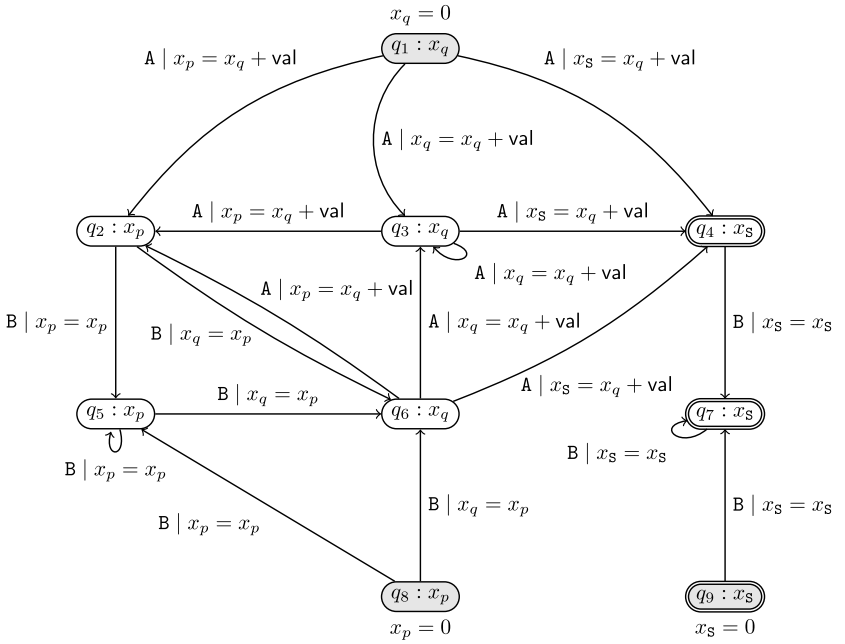


Fig. 3. Unambiguous CRA for the single-step transduction of Example 24.

Theorem 30. *For every $\mathcal{O}$, the transductions of the class $\mathsf{SR}(\mathcal{O})$ (resp., $\mathsf{SLR}(\mathcal{O})$) can be defined in forward-only word-to-DAG (resp., word-to-tree) MSO.*

Proof. With a construction that is a simple variant of the classical transformation of an automaton into a MSO formula defining the same language, we can convert an unambiguous CRA into a single-step MSO transduction. We need one copy element per register, as well as sufficient copies to construct the algebraic terms that appear in all variable updates of the CRA. The resulting MSO transduction may specify vertices labeled with the identity function id. Sequences of edges of the form $f \to_1 \mathrm{id} \to_1 \cdots \to_1 \mathrm{id} \to_i g$ can be specified with an MSO formula, and can therefore be replaced by a single edge of the form $f \to_i g$. This results in a (not necessarily single-step) MSO transduction that specifies the desired function. This construction handles copyful and copyless UCRAs uniformly: i.e., on a copyless UCRA it produces a word-to-tree MSO transduction. The well-formedness conditions for MSO transductions hold, including that all edges are forward-only. □

4. Relation to weighted automata

In this section we investigate the relationship between weighted automata and CRAs. We will show, in particular, that there are CRAs for suitable choice of $\mathcal{O}$ that correspond precisely to unambiguous and nondeterministic weighted automata respectively. This means that CRAs are a more general model of quantitative automata. In section 5 we give a brief overview of relevant literature from the line of work on weighted automata.

The model of weighted automata is a widely studied quantitative extension of nondeterministic finite-state automata [9] that describe (total) functions from Σ^* to a set of data values D. The set D is typically equipped with two operations, product and sum, that form a semiring. In a weighted automaton every transition is labeled with a tag in Σ and a weight in D. The weight of a run of the automaton is the product of all transition weights in the run. The output weight associated with a input string w is the sum of the weights of all the runs that are labeled with w. Unlike CRAs, weighted automata define total functions: if there are no such runs, the output weight would be the zero of the semiring.

Formally, a *semiring* (also called a *rig*) is an algebraic structure $\mathcal{D} = (D, +, \cdot, 0, 1)$ where $+$ is a commutative, associative binary operation on D with identity 0, $\cdot$ is an associative binary operation on D with identity 1 and absorbing element 0, and $\cdot$ distributes over $+$. A *weighted automaton* over a semiring $\mathcal{D}$ is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$, where Q is the set of states, Σ is the input alphabet, $\Delta : Q \times \Sigma \times Q \to D$ is the transition function, $I : Q \to D$ is the initial weight function, and $F : Q \to D$ is the final weight function. For a word $u = a_1 a_2 \cdots a_n \in \Sigma^*$, a *$u$-path* in $\mathcal{A}$ is a sequence

$$\pi = q_0 \xrightarrow{a_1/d_1} q_1 \xrightarrow{a_2/d_2} q_2 \cdots \xrightarrow{a_n/d_n} q_n,$$

where $\Delta(q_{i-1}, a_i, q_i) = d_i$ for all $i = 1, 2, \ldots, n$. The *weight* of the path π is $\mathsf{weight}(\pi) = d_1 \cdot d_2 \cdots d_n$. We denote by $\mathsf{first}(\pi)$ and $\mathsf{last}(\pi)$ the first state q_0 and the last state q_n of the path π respectively. Finally, the *weight* of a word $u \in \Sigma^*$ in $\mathcal{A}$ is given by

$$\mathsf{weight}(u) = \textstyle\sum_{u\text{-path } \pi} I(\mathsf{first}(\pi)) \cdot \mathsf{weight}(\pi) \cdot F(\mathsf{last}(\pi)).$$

The weighted automaton $\mathcal{A}$ computes the function $[\![\mathcal{A}]\!] : \Sigma^* \to D$, where $[\![\mathcal{A}]\!](u)$ is the weight of u in $\mathcal{A}$.

The set of *initial states* of $\mathcal{A}$ is $\{q \in Q \mid I(q) \neq 0\}$, and similarly the set of *final states* of $\mathcal{A}$ is $\{q \in Q \mid F(q) \neq 0\}$. A path π in $\mathcal{A}$ is *successful* if $\mathsf{first}(\pi)$ is initial and $\mathsf{last}(\pi)$ is final. A path is *unsuccessful* if it is not successful. Observe that the weight of an unsuccessful path is equal to 0. A weighted automaton $\mathcal{A}$ is said to be *unambiguous* if for every word $u \in \Sigma^*$ there is at most one successful u-path in $\mathcal{A}$.

An unambiguous weighted automaton can be over a monoid, instead of a semiring: modify I and F to be partial functions instead of total, where their domains are the initial and final states, respectively. Then removing $+$ and 0, we are left a single associative operation $\cdot$ on D with identity 1. The unambiguous automaton then defines, in general, a *partial* function from input strings to elements of the monoid $(D, \cdot, 1)$. The following theorem shows a correspondence between unambiguous weighted automata over the monoid $(D, \cdot, 1)$ and *copyless* UCRAs that can use the multiplication operation $\cdot$ in a certain restricted way: only multiplying by constants on the right. For a constant $d \in D$, we write $(- \cdot d)$ for the function that multiplies by d on the right.

Recall from Section 2.4 that, for a monoid M, the set of operations $\mathcal{O}_M^{\mathsf{unary}}$ contains the identity 1 and the unary operation $(- \cdot d)$ for every $d \in D$.

Theorem 31. *Suppose that $M = (D, \cdot, 1)$ is a monoid, i.e. $\cdot$ is an associative binary operation on D and 1 is a left and right identity for $\cdot$. Then the following are equal: (1) the class of total functions $\Sigma^* \to D$ computed by unambiguous weighted automata over M, (2) the class of total transductions in $\mathsf{SLR}(\mathcal{O}_M^{\mathsf{unary}})$, (3) the class of total transductions in $\mathsf{SR}(\mathcal{O}_M^{\mathsf{unary}})$.*

Proof. Recall from Lemma 8 that $\mathsf{SR}(\mathcal{O}_M^{\mathsf{unary}}) = \mathsf{SLR}(\mathcal{O}_M^{\mathsf{unary}})$. So we just have to show that unambiguous weighted automata over $(D, \cdot, 1)$ are expressively equivalent to copyless UCRAs over $\mathcal{O}_M^{\mathsf{unary}}$.

It is easy to see that every unambiguous weighted automaton over $(D, \cdot, 1)$ can be simulated by a copyless single-register UCRA over $\mathcal{O}_M^{\mathsf{unary}}$. So it remains to show that the computation of a copyless UCRA $\mathcal{A}$ over $\mathcal{O}_M^{\mathsf{unary}}$ can be simulated by an

unambiguous weighted automaton $\mathcal{B}$ over $(D, \cdot, 1)$. We apply the same construction as in Lemma 8, based on the idea that at every step of the computation of $\mathcal{A}$, the content of at most one register can contribute to the final output. So, $\mathcal{B}$ is defined to have state space $Q \times (X \cup \{\perp\})$, where Q is the state space of $\mathcal{A}$ and X is the set of registers of $\mathcal{A}$. We also may need additional states to compute the finalization function. □

Similarly to Theorem 31 we now establish a correspondence between (potentially ambiguous) weighted automata over a semiring $(D, +, \cdot, 0, 1)$ and copyful CRAs that can use the multiplication operation $\cdot$ only to right-multiply by constants.

Theorem 32. *Suppose that D is the type of data values and $\mathcal{D} = (D, +, \cdot, 0, 1)$ is a semiring. Moreover, assume that $\mathcal{O}$ consists of the constants $0, 1$, the binary operation $+$, and the family of unary operations $(- \cdot d)$ for every $d \in D$. The class of total transductions of $\mathsf{SR}(\mathcal{O})$ is equal to the class of transductions computed by weighted automata over $\mathcal{D}$.*

Proof. Every weighted automaton $(Q, \Sigma, \Delta, I, F)$ can be simulated by a copyful single-state CRA by considering one register x_q for each state q of the weighted automaton. The registers are used to encode the configuration of the weighted automaton, which is a mapping from the state space to D. For every tag a, we include an a-labeled transition in the CRA that updates the variable x_q with the assignment $x_q := \sum_{p \in Q} x_p \cdot \Delta(p, a, q)$.

Consider now a copyful CRA $\mathcal{A}$ that computes a total transduction. It remains to show that the computation of $\mathcal{A}$ can be simulated by a weighted automaton. We will describe a weighted automaton $\mathcal{B}$ that computes the same transduction as $\mathcal{A}$. We may algebraically simplify every update of $\mathcal{A}$ to be of the form

$$x = x_1 \cdot d_1 + \cdots + x_n \cdot d_n + d,$$

where $x_1, \ldots, x_n$ is an enumeration of the registers of $\mathcal{A}$ and $d, d_i \in D$ are constant weights. The state space of $\mathcal{B}$ consists of states q and pairs (q, x), where q is a state of $\mathcal{A}$ and x is a register of $\mathcal{A}$. Using the above algebraic simplification for each register x_1 to x_n, the register update function of a transition $q \to^a q'$ of $\mathcal{A}$ can be presented in the following form:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^\top = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^\top \cdot \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1n} \\ d_{21} & d_{22} & \cdots & d_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \cdots & d_{nn} \end{bmatrix} + \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}^\top,$$

where $d_{i,j} \in D$ and $d_i \in D$ are constant weights. The above matrix equation can be reformulated as follows:

$$x_j = (\textstyle\sum_{i=1}^{n} x_i \cdot d_{ij}) + d_j \text{ for all } j = 1, \ldots, n.$$

The update function is simulated in the weighted automaton $\mathcal{B}$ by defining the following transitions:

$$\Delta((q, x_i), a, (q', x_j)) = d_{ij} \qquad \Delta(q, a, (q', x_j)) = d_j \qquad \Delta(q, a, q') = 1$$

The function Δ assigns the weight 0 to every other possible transition. If q is a final state of $\mathcal{A}$ with finalization function $x_1 \cdot d_1 + \cdots + x_n \cdot d_n + d$, we define the final weight function F of $\mathcal{B}$ by $F(q) = d$ and $F((q, x_i)) = d_i$. If q is not a final state, then we put $F(q) = 0$ and $F(q, x) = 0$. Suppose that I is the initialization function of $\mathcal{A}$. We will describe the initial weight function J of $\mathcal{B}$. If q is an initial state of $\mathcal{A}$, we define $J(q) = 1$ and $J((q, x)) = [\![I(q)(x)]\!]$. If q is not an initial state of $\mathcal{A}$, we define $J(q) = 0$ and $J((q, x)) = 0$. □

It is important that we have restricted the allowed operations of the CRA to include only right-multiplication by a constant, a unary operation. If a (copyful) CRA can use the binary $\cdot$ operation of the semiring $(D, +, \cdot, 0, 1)$, then it can compute the function $a^i \mapsto d^{2^i}$ (where $a \in \Sigma$ and $d \in D$), which is not implementable by a weighted automaton.

5. Related work

Weighted automata and extensions Weighted Automata, which were introduced in 1961 by Schützenberger [15] (see also the more recent monograph [9]), 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 [16]. Weighted automata have found applications in speech and language processing [17], and they have been extended for modeling systems and verifying quantitative properties of these systems [18]. The studied models for the latter purpose include *Nested Weighted Automata* [19], a two-level variant of weighted automata for infinite strings. 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 [20,21], so the studied models are usually equipped with a very limited set of primitive operations that are insufficient for expressing realistic computations over streaming data. We have shown in Section 4 that CRAs generalize weighted automata. In particular, when the family $\mathcal{O}$ of available operations is chosen appropriately, CRAs correspond precisely to unambiguous and nondeterministic weighted automata respectively.

Register automata Another approach to augment classical automata with quantitative features has been with the addition of *registers* that can store values from a potentially infinite set. These models are typically varied in two aspects: by the choice of data types and operations that are allowed for register manipulation, and by the ability to perform tests on the registers for control flow.

The literature on data words, data/register automata and their associated logics [22–26] studies models that 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 and $\mathbb{N}$ is the set of the natural numbers. They allow comparing data values for equality, and these equality tests can affect the control flow. Due to this feature many interesting decision problems become undecidable [23].

Cost register automata Cost Register Automata (CRAs) [5,27] and Streaming String Transducers (SSTs) [11,28,29] have recently been proposed as models to permit more complex operations over the cost domain. Unlike the case of register automata, which traditionally only allow checking for equality, CRAs are parameterized by the set of permissible operations $\mathcal{O}$ over the values of the registers. On the other hand, CRAs and SSTs also enforce a strict separation between data and control, so that data values from the input sequence can only affect the contents of the registers but cannot determine the state transitions. SSTs and CRAs (we are referring to the original deterministic CRA definition of [5] with demands copylessness and allows terms with parameters in the registers) have robust expressive power [5,28], including equivalent characterizations in MSO and closure under various transformations, and have been used in applications such as verifying list processing programs [29]. Work has been devoted to study the circuit complexity of the functions being computed [30, 31], and on analysis problems such as determining the register complexity [27,32]. Previous work on CRAs has either focused on some concrete families of allowed operations (e.g. addition and multiplication over a semiring [33]), or it has allowed registers whose contents may include *syntactic terms* rather than concrete values from the data domain. In this work, we study CRAs over a general set of operations, either copyful or copyless, and we do not allow syntactic terms, so as to identify the *streamable* regular functions.

More recently, [10] proposes a model that is expressively equivalent to CRAs, but exponentially more succinct, and still permits streaming evaluation. Rather than having a finite set of states and a separate finite set of registers, this model (Data Transducers) employs a finite set of combined "state variables" which are either inactive or active (like a CRA state), but also hold a value if they are active (like a CRA register). A more succinct model than CRAs is appropriate for the design of practical query languages based on streamable regular functions (see the paragraph on quantitative regular expressions, below).

MSO-definable graph transformations An important goal for language theorists has been to characterize operational models such as automata and transducers using properties and functions expressible in various logical theories. The earliest of these results includes the seminal work of Büchi [34] and Elgot [35], establishing the equivalence between regular languages and properties expressible using monadic second-order logic (MSO). Courcelle considered the notion of MSO-definable graph transductions [12]. Engelfriet and Hoogeboom [14] showed the equivalence of string-to-string transformations expressible in Courcelle's framework with those computable using two-way finite state transducers, and Alur and Černy equated their expressive power with the deterministic one-way model of streaming string transducers (SSTs) [28]. These have also been extended to transformations of infinite strings [36]. Streaming tree transducers [11] and CRAs [5] are similar characterizations of tree-to-tree and string-to-tree transformations respectively. With this background, the present paper shows that copyful CRAs over $\mathcal{O}$ can compute exactly those functions which can be expressed as word-to-DAG transformations in MSO, i.e. the class of streamable regular transductions, $\mathsf{SR}(\mathcal{O})$.

Quantitative regular expressions The connection between finite automata and regular expressions has motivated research into identifying similar linguistic characterizations of the transductions computable by CRAs and SSTs [1,37]. The resulting formalisms, *Quantitative Regular Expressions* (QREs) and DReX respectively, have subsequently been applied in expressing string transformations [38], monitoring network traffic (NetQRE) [4], and processing streaming data (StreamQRE) [2,3]. StreamQRE has been used to specify streaming algorithms for medical monitoring [39,40], and the efficiency of its evaluation has been investigated in [10,41,42]. StreamQRE has also inspired work on the design of novel type-based language abstractions for distributed stream processing [43,44]. The closure of the transduction classes SR and SLR under the regular combinators naturally facilitates the design of these declarative languages. These papers are then largely concerned with twin problems of expressiveness—providing translation algorithms to and from CRAs and SSTs—and fast query evaluation. As with previous research on CRAs, they focus on copyless models, and exploit these for time- and memory-efficient one-pass query evaluation. An important technical challenge is the design of a type system which constrains the rate of the transduction and the composition rules to permit efficient modular evaluation. The combinators of Table 1, for example, have been refined in [2,3] with a type system that eliminates sources of complexity. When the registers of CRAs maintain terms, an important consideration is term *simplification*, so that the contents of each register may be maintained in constant space, regardless of the length of the input stream seen so far [1]. In this context, developing a regular expression-like characterization of the class of streamable regular transductions, $\mathsf{SR}(\mathcal{O})$, which we consider in this paper, is a problem for future work.

6. Conclusion

We have studied the class of *streamable regular transductions* (SR), which are partial functions from input streams of tagged values to output values that can be computed by unambiguous Cost Register Automata (UCRAs). The subclass of *streamable linear regular transductions* (SLR) consists of those transductions that are computed by *copyless* UCRAs, where the register updates are restricted so that each register appears at most once in the right-hand side of a parallel update. We have shown that the classes SR and SLR have appealing logical characterizations: SR (resp., SLR) corresponds to MSO-definable transformations from strings to DAGs (trees) without backward edges. A precise relationship with the classical model of weighted automata has also been established.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported by NSF award CCF 1763514.

References

[1] R. Alur, D. Fisman, M. Raghothaman, Regular programming for quantitative properties of data streams, in: Proceedings of the 25th European Symposium on Programming, ESOP '16, 2016, pp. 15–40.
[2] K. Mamouras, M. Raghothaman, R. Alur, Z.G. Ives, S. Khanna, StreamQRE: modular specification and efficient evaluation of quantitative queries over streaming data, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '17, ACM, New York, NY, USA, 2017, pp. 693–708.
[3] R. Alur, K. Mamouras, An introduction to the StreamQRE language, in: Dependable Software Systems Engineering, vol. 50, 2017, p. 1.
[4] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, B.T. Loo, Quantitative network monitoring with NetQRE, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, ACM, 2017, pp. 99–112.
[5] R. Alur, L. D'Antoni, J. Deshmukh, M. Raghothaman, Y. Yuan, Regular functions and cost register automata, in: Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '13, 2013, pp. 13–22.
[6] R. Bloem, J. Engelfriet, A comparison of tree transductions defined by monadic second order logic and by attribute grammars, J. Comput. Syst. Sci. 61 (1) (2000) 1–50, https://doi.org/10.1006/jcss.1999.1684.
[7] J. Engelfriet, S. Maneth, Macro tree transducers, attribute grammars, and MSO definable tree translations, Inf. Comput. 154 (1) (1999) 34–91, https://doi.org/10.1006/inco.1999.2807.
[8] R. Alur, L. D'Antoni, Streaming tree transducers, J. ACM 64 (5) (2017) 1–55, https://doi.org/10.1145/3092842.
[9] M. Droste, W. Kuich, H. Vogler (Eds.), Handbook of Weighted Automata, Springer, 2009.
[10] R. Alur, K. Mamouras, C. Stanford, Modular quantitative monitoring, in: Proceedings of the ACM on Programming Languages 3, POPL, 2019, pp. 1–31.
[11] R. Alur, L. D'Antoni, Streaming tree transducers, in: Proceedings of the 39th International Colloquium on Automata, Languages, and Programming, ICALP '12, 2012, pp. 42–53.
[12] B. Courcelle, Monadic second-order definable graph transductions: a survey, Theor. Comput. Sci. 126 (1) (1994) 53–75, https://doi.org/10.1016/0304-3975(94)90268-2.
[13] W. Thomas, Languages, automata, and logic, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages: Volume 3 Beyond Words, Springer, 1997, pp. 389–455.
[14] J. Engelfriet, H.J. Hoogeboom, MSO definable string transductions and two-way finite-state transducers, ACM Trans. Comput. Log. 2 (2) (2001) 216–254, https://doi.org/10.1145/371316.371512.
[15] M.P. Schützenberger, On the definition of a family of automata, Inf. Control 4 (2) (1961) 245–270, https://doi.org/10.1016/S0019-9958(61)80020-X.
[16] K. Chatterjee, L. Doyen, T.A. Henzinger, Quantitative languages, ACM Trans. Comput. Log. 11 (4) (2010) 23, https://doi.org/10.1145/1805950.1805953.
[17] M. Mohri, Finite-state transducers in language and speech processing, Comput. Linguist. 23 (2) (1997) 269–311.
[18] K. Chatterjee, T.A. Henzinger, J. Otop, Quantitative monitor automata, in: X. Rival (Ed.), Proceedings of the 23rd International Symposium on Static Analysis, SAS '16, Springer, Berlin, Heidelberg, 2016, pp. 23–38.
[19] K. Chatterjee, T.A. Henzinger, J. Otop, Nested weighted automata, in: Proceedings of the 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '15, IEEE Computer Society, 2015, pp. 725–737.
[20] D. Krob, The equality problem for rational series with multiplicities in the tropical semiring is undecidable, Int. J. Algebra Comput. 4 (3) (1994) 405–425, https://doi.org/10.1142/S0218196794000063.
[21] S. Almagor, U. Boker, O. Kupferman, What's decidable about weighted automata?, in: T. Bultan, P.-A. Hsiung (Eds.), Proceedings of the 9th International Symposium on Automated Technology for Verification an Analysis, ATVA '11, Springer, Berlin, Heidelberg, 2011, pp. 482–491.
[22] M. Kaminski, N. Francez, Finite-memory automata, Theor. Comput. Sci. 134 (2) (1994) 329–363, https://doi.org/10.1016/0304-3975(94)90242-9.
[23] F. Neven, T. Schwentick, V. Vianu, Finite state machines for strings over infinite alphabets, ACM Trans. Comput. Log. 5 (3) (2004) 403–435, https://doi.org/10.1145/1013560.1013562.
[24] S. Demri, R. Lazić, LTL with the freeze quantifier and register automata, ACM Trans. Comput. Log. 10 (3) (2009) 1–30, https://doi.org/10.1145/1507244.1507246.
[25] H. Björklund, T. Schwentick, On notions of regularity for data languages, Theor. Comput. Sci. 411 (4) (2010) 702–715, https://doi.org/10.1016/j.tcs.2009.10.009, Fundamentals of Computation Theory.
[26] M. Bojańczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin, Two-variable logic on data words, ACM Trans. Comput. Log. 12 (4) (2011) 1–26, https://doi.org/10.1145/1970398.1970403.
[27] R. Alur, M. Raghothaman, Decision problems for additive regular functions, in: F.V. Fomin, R. Freivalds, M. Kwiatkowska, D. Peleg (Eds.), Proceedings of the 40th International Colloquium on Automata, Languages, and Programming, Part II, ICALP 2013, Springer, Berlin, Heidelberg, 2013, pp. 37–48.
[28] R. Alur, P. Cerný, Expressiveness of streaming string transducers, in: K. Lodaya, M. Mahajan (Eds.), IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 8, Schloss Dagstuhl–Leibniz–Zentrum fuer Informatik, Dagstuhl, Germany, 2010, pp. 1–12.

[29] R. Alur, P. Černý, Streaming transducers for algorithmic verification of single-pass list-processing programs, in: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, ACM, New York, NY, USA, 2011, pp. 599–610.
[30] E. Allender, I. Mertz, Complexity of regular functions, in: A.-H. Dediu, E. Formenti, C. Martín-Vide, B. Truthe (Eds.), Language and Automata Theory and Applications, Springer International Publishing, Cham, 2015, pp. 449–460.
[31] E. Allender, A. Krebs, P. McKenzie, Better Complexity Bounds for Cost Register Automata, in: K.G. Larsen, H.L. Bodlaender, J.-F. Raskin (Eds.), 42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 83, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017, pp. 1–14.
[32] L. Daviaud, P.-A. Reynier, J.-M. Talbot, A generalised twinning property for minimisation of cost register automata, in: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, ACM, New York, NY, USA, 2016, pp. 857–866.
[33] F. Mazowiecki, C. Riveros, On the expressibility of copyless cost register automata, CoRR, arXiv:1504.01709.
[34] J.R. Büchi, Weak second-order arithmetic and finite automata, Math. Log. Q. 6 (1–6) (1960) 66–92, https://doi.org/10.1002/malq.19600060105.
[35] C.C. Elgot, Decision problems of finite automata design and related arithmetics, Trans. Am. Math. Soc. 98 (1) (1961) 21–51.
[36] R. Alur, E. Filiot, A. Trivedi, Regular transformations of infinite strings, in: Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science, LICS '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 65–74.
[37] R. Alur, A. Freilich, M. Raghothaman, Regular combinators for string transformations, in: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, ACM, New York, NY, USA, 2014, pp. 1–10.
[38] R. Alur, L. D'Antoni, M. Raghothaman, DReX: a declarative language for efficiently evaluating regular string transformations, in: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, ACM, New York, NY, USA, 2015, pp. 125–137.
[39] H. Abbas, A. Rodionova, K. Mamouras, E. Bartocci, S.A. Smolka, R. Grosu, Quantitative regular expressions for arrhythmia detection, in: IEEE/ACM Transactions on Computational Biology and Bioinformatics, 2018.
[40] H. Abbas, R. Alur, K. Mamouras, R. Mangharam, A. Rodionova, Real-time decision policies with predictable performance, in: Special Issue on Design Automation for Cyber-Physical Systems, Proc. IEEE 106 (9) (2018) 1593–1615, https://doi.org/10.1109/JPROC.2018.2853608.
[41] R. Alur, K. Mamouras, C. Stanford, Automata-based stream processing, in: I. Chatzigiannakis, P. Indyk, F. Kuhn, A. Muscholl (Eds.), Proceedings of the 44th International Colloquium on Automata, Languages, and Programming, ICALP '17, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 80, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017, pp. 1–15.
[42] R. Alur, K. Mamouras, D. Ulus, Derivatives of quantitative regular expressions, in: L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, R. Mardare (Eds.), Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday, in: Lecture Notes in Computer Science, vol. 10460, Springer International Publishing, Cham, 2017, pp. 75–95.
[43] K. Mamouras, C. Stanford, R. Alur, Z.G. Ives, V. Tannen, Data-trace types for distributed stream processing systems, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, ACM, New York, NY, USA, 2019, pp. 670–685.
[44] R. Alur, K. Mamouras, C. Stanford, V. Tannen, Interfaces for stream processing systems, in: M. Lohstroh, P. Derler, M. Sirjani (Eds.), Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday, in: Lecture Notes in Computer Science, vol. 10760, Springer, Cham, 2018, pp. 38–60.