Interfaces for Stream Processing Systems

Rajeev Alur, Konstantinos Mamouras, Caleb Stanford, and Val Tannen

University of Pennsylvania, Philadelphia, PA, USA
{alur,mamouras,castan,val}@cis.upenn.edu

Abstract. Efficient processing of input data streams is central to IoT systems, and the goal of this paper is to develop a logical foundation for specifying the computation of such stream processing. In the proposed model, both the input and output of a stream processing system consists of tagged data items with a dependency relation over tags that captures the logical ordering constraints over data items. While a system processes the input data one item at a time incrementally producing output data items, its semantics is a function from input data traces to output data traces, where a data trace is an equivalence class of sequences of data items induced by the dependency relation. This data-trace transduction model generalizes both acyclic Kahn process networks and relational query processors, and can specify computations over data streams with a rich variety of ordering and synchronization characteristics. To form complex systems from simpler ones, we define sequential composition and parallel composition operations over data-trace transductions, and show how to define commonly used idioms in stream processing such as sliding windows, key-based partitioning, and map-reduce.

1 Introduction

The last few years have witnessed an explosion of IoT systems in a diverse range of applications such as smart buildings, wearable devices, and healthcare. A key component of an effective IoT system is the ability to continuously process incoming data streams and make decisions in response in a timely manner. Systems such as Apache Storm (see `storm.apache.org`) and Twitter Heron [12] provide the necessary infrastructure to implement distributed stream processing systems with the focus mainly on high performance and fault tolerance. What's less developed though is the support for high-level programming abstractions for such systems so that *correctness with respect to formal requirements* and *predictable performance* can be assured at design time. The goal of this paper is to provide a logical foundation for modeling distributed stream processing systems.

An essential step towards developing the desired formal computational model for stream processing systems is to understand the *interface*, that is, the types of input, the types of output, and the logical computations such systems perform. (See [8] for the role of interfaces in system design.) As a starting point, we can view the input to be a sequence of data items that the system consumes one item at a time in a streaming fashion. Assuming a strict linear order over input

items is however not the ideal abstraction for two reasons. First, in an actual implementation, the input data items may arrive at multiple physical locations and there may be no meaningful logical way to impose an ordering among items arriving at different locations. Second, for the computation to be performed on the input data items, it may suffice to view the data as a *relation*, that is, a *bag* of items without any ordering. Such lack of ordering also has computational benefits since it can be exploited for parallelization of the implementation. *Partially ordered multisets* (pomsets), a structure studied extensively in concurrency theory [20], generalize both sequences and bags, and thus, assuming the input of a stream processing system to be a pomset seems general enough.

While the input logically consists of partially ordered data items, a stream processing system consumes it one item at a time, and we need a representation that is suitable for such a streaming model of computation. Inspired by the definition of *Mazurkiewicz traces* in concurrency theory [18], we model the input as a *data trace*. We assume that each data item consists of a *tag* and a value of a basic data type associated with this tag. The ordering of items is specified by a (symmetric) *dependency relation* over the set of tags. Two sequences of data items are considered equivalent if one can be obtained from the other by repeatedly commuting two adjacent items with independent tags, and a data trace is an equivalence class of such sequences. For instance, when all the tags are mutually dependent, a sequence of items represents only itself, and when all the tags are mutually independent, a sequence of items represents the bag of items it contains. A suitable choice of tags along with the associated dependency relation, allows us to model input streams with a rich variety of ordering and synchronization characteristics.

As the system processes each input item in a streaming manner, it responds by producing output data items. Even though the cumulative output items produced in a response to an input sequence is linearly ordered based on the order in which the input gets processed, we need the flexibility to view output items as only partially ordered. For instance, consider a system that implements *key-based partitioning* by mapping a linearly ordered input sequence to a *set* of linearly ordered sub-streams, one per key. To model such a system the output items corresponding to distinct keys should be unordered. For this purpose, we allow the output items to have their own tags along with a dependency relation over these tags, and a sequence of outputs produced by the system is interpreted as the corresponding data trace.

While a system processes the input in a specific order by consuming items one by one in a streaming manner, it is required to interpret the input sequence as a data trace, that is, outputs produced while processing two equivalent input sequences should be equivalent. Formally, this means that a stream processor defines a *function from input data traces to output data traces*. Such a *data-trace transduction* is the proposed interface model for distributed stream processing systems. We define this model in section 2, and illustrate it using a variety of examples and relating it to existing models in literature such as Kahn process networks [11, 14] and streaming extensions of database query languages [5, 15].

In section 3, we define two basic operations on data-trace transductions that can be used to construct complex systems from simpler ones. Given two data-trace transductions f and g, the *sequential composition* $f \gg g$ feeds the output data trace produced by f as input to g, and is defined when the output type of f coincides with the input type of g, while the *parallel composition* $f \parallel g$ executes the two in parallel, and is defined when there are no common tags in the outputs of f and g (that is, the output data items produced by the two components are independent). We illustrate how these two operations can be used to define common computing idioms in stream processing systems such as sliding windows and map reduce.

2 Data-Trace Transductions

In order to describe the behavior of a distributed stream processing system we must specify its *interface*, that is: the type of the input stream, the type of the output stream, and the input/output transformation that the system performs. This section contains formal descriptions of three key concepts: (1) *data traces* for describing finite collections of partially ordered data items, (2) *data-trace transductions* for modeling the input/output behavior of a stream processing system, and (3) *data-string transductions* for specifying the behavior of sequential implementations of such systems.

2.1 Data Traces

We use data traces to model streams in which the data items are partially ordered. Data traces generalize sequences (data items are linearly ordered), relations (data items are unordered), and independent stream channels (data items are organized as a collection of linearly ordered subsets). The concatenation operation, the prefix order, and the residuation operation on sequences can be generalized naturally to the setting of data traces.

Definition 1 (Data Type). A *data type* $A = (\Sigma, (T_\sigma)_{\sigma \in \Sigma})$ consists of a potentially infinite *tag alphabet* Σ and a value type T_σ for every tag $\sigma \in \Sigma$. The set of *elements* of type A is equal to $\{(\sigma, d) \mid \sigma \in \Sigma \text{ and } d \in T_\sigma\}$, which we will also denote by A. The set of *sequences* over A is denoted as A^*. □

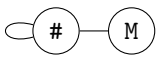
Example 2. Suppose we want to process a stream that consists of sensor measurements and special symbols that indicate the end of a one-second interval. The data type for this input stream involves the tags $\Sigma = \{\mathtt{M}, \mathtt{\#}\}$, where `M` is meant to indicate a sensor measurement and `#` is an end-of-second marker. The value sets for these tags are $T_{\mathtt{M}} = \mathbb{N}$ (natural numbers), and $T_{\mathtt{\#}} = \mathbb{U}$ is the unit type (singleton). So, the data type $A = (\Sigma, T_{\mathtt{M}}, T_{\mathtt{\#}})$ contains measurements $(\mathtt{M}, d)$, where d is a natural number, and the end-of-second symbol `#`. □

Definition 3 (Dependence Relation and Induced Congruence). A *dependence relation* on a tag alphabet Σ is a symmetric binary relation on Σ.

We say that the tags σ, τ are *independent* (w.r.t. a dependence relation D) if $(\sigma, \tau) \notin D$. For a data type $A = (\Sigma, (T_\sigma)_{\sigma \in \Sigma})$ and a dependence relation D on Σ, we define the dependence relation that is induced on A by D as $\{((\sigma, d), (\sigma', d')) \in A \times A \mid (\sigma, \sigma') \in D\}$, which we will also denote by D. Define $\equiv_D$ to be the smallest congruence (w.r.t. sequence concatenation) on A^* containing $\{(ab, ba) \in A^* \times A^* \mid (a, b) \notin D\}$. □

According to the definition of the relation $\equiv_D$ above, two sequences are equivalent if one can be obtained from the other by performing commutations of adjacent items with independent tags.

Example 4 (Dependence Relation). For the data type of Example 2, we consider the dependence relation $D = \{(\mathtt{M}, \mathtt{\#}), (\mathtt{\#}, \mathtt{M}), (\mathtt{\#}, \mathtt{\#})\}$. The dependence relation can be visualized as an undirected graph:



This means that the tag M is independent of itself, and therefore consecutive M-tagged items are considered unordered. For example, the sequences

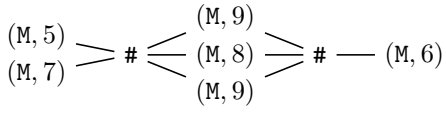
$$(\mathtt{M}, 5)\ (\mathtt{M}, 8)\ (\mathtt{M}, 5)\ \mathtt{\#}\ (\mathtt{M}, 9) \quad (\mathtt{M}, 5)\ (\mathtt{M}, 5)\ (\mathtt{M}, 8)\ \mathtt{\#}\ (\mathtt{M}, 9) \quad (\mathtt{M}, 8)\ (\mathtt{M}, 5)\ (\mathtt{M}, 5)\ \mathtt{\#}\ (\mathtt{M}, 9)$$

are all equivalent w.r.t. $\equiv_D$. □

Definition 5 (Data Traces, Concatenation, Prefix Relation). A *data-trace type* is a pair $X = (A, D)$, where $A = (\Sigma, (T_\sigma)_{\sigma \in \Sigma})$ is a data type and D is a dependence relation on the tag alphabet Σ. A *data trace* of type X is a congruence class of the relation $\equiv_D$. We also write X to denote the set of data traces of type X. Since the equivalence $\equiv_D$ is a congruence w.r.t. sequence concatenation, the operation of concatenation is also well-defined on data traces: $[u] \cdot [v] = [uv]$ for sequences u and v, where $[u]$ is the congruence class of u. We define the relation $\leq$ on the data traces of X as a generalization of the prefix partial order on sequences: for data traces $\mathbf{u}$ and $\mathbf{v}$ of type X, $\mathbf{u} \leq \mathbf{v}$ iff there are sequences $u \in \mathbf{u}$ and $v \in \mathbf{v}$ such that $u \leq v$ (i.e., u is a prefix of v). □

The relation $\leq$ on data traces of a fixed type is easily checked to be a partial order. Since it generalizes the prefix order on sequences (when the congruence classes of $\equiv_D$ are singleton sets), we will call $\leq$ the *prefix order* on data traces.

Example 6 (Data Traces). Consider the data-trace type $X = (A, D)$, where A is the data type of Example 2 and D is the dependence relation of Example 4. A data trace of X can be represented as a sequence of multisets (bags) of natural numbers and visualized as a pomset. For example, the data trace that corresponds to the sequence $(\mathtt{M}, 5)\ (\mathtt{M}, 7)\ \mathtt{\#}\ (\mathtt{M}, 9)\ (\mathtt{M}, 8)\ (\mathtt{M}, 9)\ \mathtt{\#}\ (\mathtt{M}, 6)$ can be visualized as the following labeled partial order (pomset)

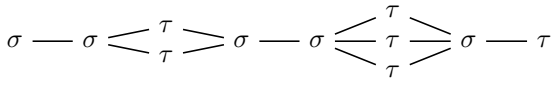


where a line from left to right indicates that the item on the right must occur after the item on the left. The end-of-second markers # separate multisets of natural numbers. So, the set of data traces of X has an isomorphic representation as the set $\mathsf{Bag}(\mathbb{N})^+$ of nonempty sequences of multisets of natural numbers. In particular, the empty sequence ε is represented as $\emptyset$ and the single-element sequence # is represented as $\emptyset\,\emptyset$. □

Let us observe that a singleton tag alphabet can be used to model sequences or multisets over a basic type of values. For the data type given by $\Sigma = \{\sigma\}$ and $T_\sigma = T$ there are two possible dependence relations for Σ, namely $\emptyset$ and $\{(\sigma, \sigma)\}$. The data traces of $(\Sigma, T, \emptyset)$ are multisets over T, which we denote as $\mathsf{Bag}(T)$, and the data traces of $(\Sigma, T, \{(\sigma, \sigma)\})$ are sequences over T.

Let us consider the slightly more complicated case of a tag alphabet $\Sigma = \{\sigma, \tau\}$ consisting of two elements (together with data values sets T_σ and T_τ). The two tags can be used to describe more complex sets of data traces:

1. Dependence relation $D = \{(\sigma, \sigma), (\sigma, \tau), (\tau, \sigma), (\tau, \tau)\}$: The set of data traces is (up to a bijection) $(T_\sigma \oplus T_\tau)^*$, where $\oplus$ is the disjoint union operation.
2. $D = \{(\sigma, \sigma), (\tau, \tau)\}$: The set of data traces is $T_\sigma^* \times T_\tau^*$.
3. $D = \{(\sigma, \sigma), (\sigma, \tau), (\tau, \sigma)\}$: In a data trace the items tagged with σ separate (possibly empty) multisets of items tagged with τ. For example:



   So, the set of data traces is $\mathsf{Bag}(T_\tau) \cdot (T_\sigma \cdot \mathsf{Bag}(T_\tau))^*$, where $\cdot$ is the concatenation operation for sequences.
4. $D = \{(\sigma, \sigma)\}$: The set of data traces is $T_\sigma^* \times \mathsf{Bag}(T_\tau)$.
5. $D = \{(\sigma, \tau), (\tau, \sigma)\}$: A data trace is an alternating sequence of σ-multisets (multisets with data items tagged with σ) and τ-multisets. More formally, the set of data traces is

$$\begin{aligned}\{\varepsilon\} \cup \mathsf{Bag}_1(T_\sigma) \cdot (\mathsf{Bag}_1(T_\tau) \cdot \mathsf{Bag}_1(T_\sigma))^* \cdot \mathsf{Bag}(T_\tau) \\ \cup \mathsf{Bag}_1(T_\tau) \cdot (\mathsf{Bag}_1(T_\sigma) \cdot \mathsf{Bag}_1(T_\tau))^* \cdot \mathsf{Bag}(T_\sigma),\end{aligned}$$

   where $\mathsf{Bag}_1(T)$ is the set of nonempty multisets over T.
6. $D = \emptyset$: The set of data traces is $\mathsf{Bag}(T_\sigma \oplus T_\tau)$, which is isomorphic to $\mathsf{Bag}(T_\sigma) \times \mathsf{Bag}(T_\tau)$.

The cases that have been omitted are symmetric to the ones presented above.

Example 7 (Multiple Input and Output Channels). Suppose we want to model a streaming system with multiple independent input and output channels, where the items within each channel are linearly ordered but the channels are completely independent. These assumptions are appropriate for distributed streaming systems, where the channels are implemented as network connections.

Fig. 1. Multiple channels $\mathtt{I}_1$ and $\mathtt{I}_2$, displayed on the left graphically, and on the right as the single trace type which actually defines them.

This is the setting of (acyclic) *Kahn Process Networks* [11] and the more restricted synchronous dataflow models [14]. We introduce tags $\Sigma_{\mathtt{I}} = \{\mathtt{I}_1, \ldots, \mathtt{I}_m\}$ for m input channels, and tags $\Sigma_{\mathtt{O}} = \{\mathtt{O}_1, \ldots, \mathtt{O}_n\}$ for n output channels. The dependence relation for the input consists of all pairs $(\mathtt{I}_i, \mathtt{I}_i)$ with $i = 1, \ldots, m$. This means that for all indexes $i \neq j$ the tags $\mathtt{I}_i$ and $\mathtt{I}_j$ are independent. Similarly, the dependence relation for the output consists of all pairs $(\mathtt{O}_i, \mathtt{O}_i)$ with $i = 1, \ldots, n$. Assume that the value types associated with the input tags are T_1, $\ldots$, T_m, and the value types associated with the output tags are U_1, $\ldots$, U_n. As we will show later in Proposition 10, the sets of input and output data traces are (up to a bijection) $T_1^* \times \cdots \times T_m^*$ and $U_1^* \times \cdots \times U_m^*$ respectively. □

Definition 8 (Residuals). Let u and v be sequences over a set A. If u is a prefix of v, then we define the *residual* of v by u, denoted $u^{-1}v$, to be the unique sequence w such that $v = uw$.

Let X be a data-trace type. Suppose $\mathbf{u}$ and $\mathbf{v}$ are of type X with $\mathbf{u} \leq \mathbf{v}$. Choose any representatives u and v of the traces $\mathbf{u}$ and $\mathbf{v}$ respectively such that $u \leq v$. Then, define the *residual* of $\mathbf{v}$ by $\mathbf{u}$ to be $\mathbf{u}^{-1}\mathbf{v} = [u^{-1}v]$. □

The left cancellation property of Lemma 9 below is needed for establishing that the residuation operation of Definition 8 is well-defined on traces, i.e. the trace $[u^{-1}v]$ does not depend on the choice of representatives u and v. It follows for traces $\mathbf{u}$, $\mathbf{v}$ with $\mathbf{u} \leq \mathbf{v}$ that $\mathbf{u}^{-1}\mathbf{v}$ is the unique trace $\mathbf{w}$ s.t. $\mathbf{v} = \mathbf{u} \cdot \mathbf{w}$.

Lemma 9 (Left and Right Cancellation). Let $X = (A, D)$ be a data-trace type. The following properties hold for all sequences $u, u', v, v' \in A^*$:

1. Left cancellation: If $u \equiv_D u'$ and $uv \equiv_D u'v'$ then $v \equiv_D v'$.
2. Right cancellation: If $v \equiv_D v'$ and $uv \equiv_D u'v'$ then $u \equiv_D u'$. □

Proposition 10 (Independent Ordered Channels). Let A be the data type with tag alphabet consisting of $\mathtt{C}_1$, $\ldots$, $\mathtt{C}_n$, and with respective value types $T_1, \ldots, T_n$. Define the data-trace type $X = (A, D)$, where $D = \{(\mathtt{C}_i, \mathtt{C}_i) \mid i = 1, \ldots, n\}$ is the dependence relation. The set of data traces X is isomorphic to $Y = T_1^* \times \cdots \times T_n^*$, where the concatenation operation on the elements of Y is defined componentwise.

Proposition 10 establishes a bijection between $Y = T_1^* \times \cdots \times T_n^*$ and a set of appropriately defined data traces X. The bijection involves the concatenation operation. This implies that the prefix order and the residuation operation can

be defined on Y so that they agree with the corresponding structure on the data traces. Since $\cdot$ on Y is componentwise concatenation, the order $\leq$ on Y is the componentwise prefix order. Finally, residuals are also defined componentwise on Y. So, the expanded structures $(Y, \cdot, \leq, ^{-1})$ and $(X, \cdot, \leq, ^{-1})$ are isomorphic.

Proposition 11 (Independent Unordered Channels). Let A be the data-trace type with tag alphabet consisting of $\mathtt{C}_1, \ldots, \mathtt{C}_n$, and with respective value types $T_1, \ldots, T_n$. Define the data-trace type $X = (A, D)$, where $D = \emptyset$ is the dependence relation. The set of data traces X is isomorphic to $Y = \mathsf{Bag}(T_1) \times \cdots \times \mathsf{Bag}(T_n)$, where the concatenation operation on the elements of Y is componentwise multiset union. □

Given the isomorphism between $Y = \mathsf{Bag}(T_1) \times \cdots \times \mathsf{Bag}(T_n)$ and the set of data traces described in Proposition 11, we define the prefix relation and residuation on Y that are induced by $\cdot$ on Y as follows: $\leq$ is defined as componentwise multiset containment, and $(P_1, \ldots, P_n)^{-1}(Q_1, \ldots, Q_n) = (Q_1 \setminus P_1, \ldots, Q_n \setminus P_n)$ where $\setminus$ is the multiset difference operation. It follows that this additional structure on Y agrees with the corresponding structure on the traces.

2.2 Data-Trace Transductions

Data-trace transductions formalize the notion of an *interface* for stream processing systems. Consider the analogy with a functional model of computation: the interface of a program consists of the input type, the output type, and a mapping that describes the input/output behavior of the program. Correspondingly, the interface for a stream processing systems consists of: (1) the type X of input data traces, (2) the type Y of output data traces, and (3) a monotone mapping $\beta : X \to Y$ that specifies the cumulative output after having consumed a prefix of the input stream. The monotonicity requirement captures the idea that output items cannot be retracted after they have been omitted. Since a transduction is a function from trace histories, it allows the modeling of systems that maintain state, where the output that is emitted at every step depends potentially on the entire input history.

Definition 12 (Data-Trace Transductions). Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types. A *data-trace transduction* with input type X and output type Y is a function $\beta : X \to Y$ that is monotone w.r.t. the prefix order on data traces: $\mathbf{u} \leq \mathbf{v}$ implies that $\beta(\mathbf{u}) \leq \beta(\mathbf{v})$ for all traces $\mathbf{u}, \mathbf{v} \in X$. We write $\mathcal{T}(X, Y)$ to denote the set of all data-trace transductions from X to Y. □

Fig. 2 visualizes a data-trace transduction $\beta : X \to Y$ as a block diagram element, where the input wire is annotated with the input type X and the output wire is annotated with the output type Y.

Example 13. Suppose the input is a sequence of natural numbers, and we want to define the data-trace transduction that outputs the current data item if it is

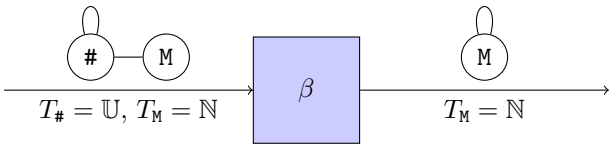


Fig. 2. A stream processing interface (data-trace transduction), consisting of (1) the input trace type, (2) the output trace type, and (3) the monotone map β.

strictly larger than all data items seen so far. This is described by the trace transduction $\beta : \mathbb{N}^* \to \mathbb{N}^*$, given by $\beta(\varepsilon) = \varepsilon$ and

$$\beta(a_1 \dots a_{n-1} a_n) = \begin{cases} \beta(a_1 \dots a_{n-1})\, a_n, & \text{if } a_n > a_i \text{ for all } i = 1, \dots, n-1; \\ \beta(a_1 \dots a_{n-1}), & \text{otherwise.} \end{cases}$$

In particular, the definition implies that $\beta(a_1) = a_1$. The table

current item	input history	β output
	ε	ε
3	3	3
1	3 1	3
5	3 1 5	3 5
2	3 1 5 2	3 5

gives the values of the transduction β for all prefixes of the stream 3 1 5 2. □

2.3 Data-String Transductions

In the previous section we defined the notion of a data-trace transduction, which describes abstractly the behavior of a distributed stream processing system using a monotone function from input data traces to output data traces. In a *sequential implementation* of a stream processor the input is consumed in a sequential fashion, i.e. one item at a time, and the output items are produced in a specific linear order. Such sequential implementations are formally represented as *data-string transductions*. We establish in this section a precise correspondence between string transductions and trace transductions. We identify a consistency property that characterizes when a string transduction implements a trace transduction of a given input/output type. Moreover, we show how to obtain from a given trace transduction the set of all its possible sequential implementations.

Definition 14 (Data-String Transductions). Let A and B be data types. A *data-string transduction* with input type A and output type B is a function $f : A^* \to B^*$. Let $\mathcal{S}(A, B)$ be the set of string transductions from A to B. □

A data-string transduction $f : A^* \to B^*$ describes a streaming computation where the input items arrive in a linear order. For an input sequence $u \in A^*$ the value $f(u)$ gives the output items that are emitted right after consuming the

sequence u. In other words, $f(u)$ is the output that is triggered by the arrival of the last data item of u. We say that f is a *one-step* description of the computation because it gives the *output increment* that is emitted at every step.

Let A be an arbitrary set, Y be a data-trace type, and $f : A^* \to Y$. We define the *lifting* of f to be the function $\overline{f} : A^* \to Y$ that maps a sequence $a_1 a_2 \dots a_n \in A^*$ to $\overline{f}(a_1 a_2 \dots a_n) = f(\varepsilon) \cdot f(a_1) \cdot f(a_1 a_2) \cdots f(a_1 a_2 \dots a_n)$. In particular, the definition implies that $\overline{f}(\varepsilon) = f(\varepsilon)$. That is, $\overline{f}$ accumulates the outputs of f for all prefixes of the input. Notice that $\overline{f}$ is *monotone* w.r.t. the prefix order: $u \leq v$ implies that $\overline{f}(u) \leq \overline{f}(v)$ for all $u, v \in A^*$. Suppose now that $\varphi : A^* \to Y$ is a monotone function. The *derivative* $\partial\varphi : A^* \to Y$ of φ is defined as follows: $(\partial\varphi)(\varepsilon) = \varphi(\varepsilon)$ and $(\partial\varphi)(ua) = \varphi(u)^{-1}\varphi(ua)$ for all $u \in A^*$ and $a \in A$. Notice that in the definition of ∂ we use the residuation operation of Definition 8. The lifting and derivative operators witness a bijection between the class of functions from A^* to Y and the monotone subset of this class. That is, $\partial\overline{f} = f$ for every $f : A^* \to Y$ and $\overline{\partial\varphi} = \varphi$ for every monotone $\varphi : A^* \to Y$.

Definition 15 (The Implementation Relation). Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types. We say that a string transduction $f : A^* \to B^*$ *implements* a trace transduction $\beta : X \to Y$ (or that f is a *sequential implementation* of β) if $\beta([u]) = [\overline{f}(u)]$ for all $u \in A^*$. □

An implementation f of a trace transduction β is meant to give the *output increment* that is emitted at every step of the streaming computation, assuming the input is presented as a totally ordered sequence. That is, for input u the value $f(u)$ gives some arbitrarily chosen linearization of the output items that are emitted after consuming u. The lifting $\overline{f}$ gives the *cumulative output* that has been emitted after consuming a prefix of the input stream.

Example 16. The trace transduction β of Example 13 can be implemented as a string transduction $f : \mathbb{N}^* \to \mathbb{N}^*$, given by $f(\varepsilon) = \varepsilon$ and

$$f(a_1 \dots a_{n-1} a_n) = \begin{cases} a_n, & \text{if } a_n > a_i \text{ for all } i = 1, \dots, n-1; \\ \varepsilon, & \text{otherwise.} \end{cases}$$

The following table gives the values of the implementation f on input prefixes:

current item	input history	f output	β output
	ε	ε	ε
3	3	3	3
1	3 1	ε	3
5	3 1 5	5	3 5
2	3 1 5 2	ε	3 5

Notice in the table that $\beta(3\,1\,5\,2) = f(\varepsilon) \cdot f(3) \cdot f(3\,1) \cdot f(3\,1\,5) \cdot f(3\,1\,5\,2)$. □

Definition 17 (Consistency). Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types. A data-string transduction $f \in \mathcal{S}(A, B)$ is (X, Y)-*consistent* if $u \equiv_D v$ implies $\overline{f}(u) \equiv_E \overline{f}(v)$ for all $u, v \in A^*$. □

Definition 17 essentially says that a string transduction f is consistent when it gives equivalent cumulative outputs for equivalent input sequences. The definition of the consistency property is given in terms of the lifting $\bar{f}$ of f. An equivalent formulation of this property, which is expressed directly in terms of f, is given in Theorem 18 below.

Theorem 18 (Characterization of Consistency). Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types, and $f \in \mathcal{S}(A, B)$. The function f is (X, Y)-consistent if and only if the following two conditions hold:
(1) For all $u \in A^*$ and $a, b \in A$, $(a, b) \notin D$ implies $f(ua)f(uab) \equiv_E f(ub)f(uba)$.
(2) For all $u, v \in A^*$ and $a \in A$, $u \equiv_D v$ implies that $f(ua) \equiv_E f(va)$. □

The following theorem establishes a correspondence between string and trace transductions. The consistent string transductions are exactly the ones that implement trace transductions. Moreover, the set of all implementations of a trace transduction can be given as a dependent function space by ranging over all possible linearizations of output increments. In other words, an implementation results from a trace transduction by choosing output increment linearizations.

Theorem 19 (Trace Transductions & Implementations). Let $X = (A, D)$ and $Y = (B, E)$ be data-trace types. The following hold:
(1) A data-string transduction f of $\mathcal{S}(A, B)$ implements some trace transduction of $\mathcal{T}(X, Y)$ iff f is (X, Y)-consistent.
(2) The set of all implementations of a data-trace transduction $\beta \in \mathcal{T}(X, Y)$ is the dependent function space $\prod_{u \in A^*} (\partial\gamma)(u)$, where the function $\gamma : A^* \to Y$ is given by $\gamma(u) = \beta([u])$ for all $u \in A^*$. □

Part (2) of Theorem 19 defines the monotone function $\gamma : A^* \to Y$ in terms of the trace transduction β. Intuitively, γ gives the cumulative output trace for every possible linearization of the input trace. It follows that $\partial\gamma : A^* \to Y$ gives the output increment, which is a trace, for every possible input sequence. So, the lifting of $\partial\gamma$ is equal to γ. Finally, an implementation of β can be obtained by choosing for every input sequence $u \in A^*$ some linearization of the output increment $(\partial\gamma)(u) \in Y$. In other words, any implementation of a data-trace transduction is specified uniquely by a linearization choice function for all possible output increments. For the special case where the function $[\cdot] : B^* \to Y$ is injective, i.e. every trace of Y has one linearization, there is exactly one implementation for a given trace transduction. We will describe later in Example 21 a trace transduction that has several different sequential implementations.

2.4 Examples of Data-Trace Transductions

In this section we present examples that illustrate the concept of a *data-trace transduction* and its *implementations* for several streaming computations on streams of partially ordered elements. We start by considering examples that fit into the model of process networks [11, 14], where the inputs and outputs are organized in collections of independent linearly ordered channels.

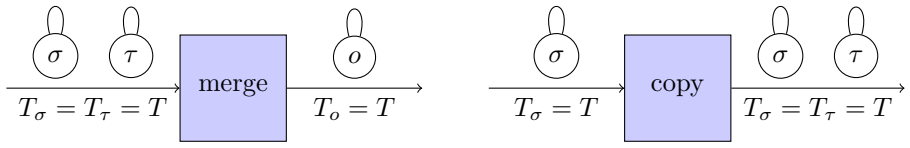


Fig. 3. Stream processing interfaces for merge (Example 20) and copy (Example 21).

Example 20 (Deterministic Merge). Consider the streaming computation where two linearly ordered input channels are merged into one. More specifically, this transformation reads cyclically items from the two input channels and passes them unchanged to the output channel. As described in Example 7, the input type is specified by the tag alphabet $\Sigma = \{\sigma, \tau\}$ with data values T for each tag, and the dependence relation $D = \{(\sigma, \sigma), (\tau, \tau)\}$. So, an input data trace is essentially an element of $T^* \times T^*$. The output type is specified by the tag alphabet $\{o\}$, the value type $T_o = T$, and the dependence relation $\{(o, o)\}$. So, the set of output data traces is essentially T^*. See the left diagram in Fig. 3. The trace transduction $\text{merge} : T^* \times T^* \to T^*$ is given as follows:

$$\text{merge}(x_1 \dots x_m, y_1 \dots y_n) = \begin{cases} x_1\ y_1\ \dots\ x_m\ y_m, & \text{if } m \le n; \\ x_1\ y_1\ \dots\ x_n\ y_n, & \text{if } m > n. \end{cases}$$

The sequential implementation of merge can be represented as a function $f : A^* \to T^*$ with $A = (\{\sigma\} \times T) \cup (\{\tau\} \times T)$, where $f(\varepsilon) = \varepsilon$ and

$$f(w\ (\sigma, x)) = \begin{cases} x\ y_{m+1}, & \text{if } \text{length}(w|_\sigma) = m,\ w|_\tau = y_1 \dots y_n \text{ and } m < n \\ \varepsilon, & \text{otherwise} \end{cases}$$

$$f(w\ (\tau, y)) = \begin{cases} x_{n+1}\ y, & \text{if } w|_\sigma = x_1 \dots x_m,\ \text{length}(w|_\tau) = n \text{ and } n < m \\ \varepsilon, & \text{otherwise} \end{cases}$$

for all $w \in A^*$. For an input tag, $\sigma\ w|_\sigma$ is the subsequence obtained from w by keeping only the values of the σ-tagged items. □

Example 21 (Copy). The copy transformation creates two copies of the input stream by reading each item from the input channel and copying it to two output channels. An input data trace is an element of T^*, and an output data trace is an element of $T^* \times T^*$. See the right diagram in Fig. 3. The trace transduction for this computation is given by $\text{copy}(u) = (u, u)$ for all $u \in T^*$. A possible implementation of copy can be given as $f : T^* \to B^*$, where $B = (\{\sigma\} \times T) \cup (\{\tau\} \times T)$. We put $f(\varepsilon) = \varepsilon$ and $f(ua) = (\sigma, a)\ (\tau, a)$ for all $u \in A^*$. Notice that the implementation makes the arbitrary choice to emit (σ, a) before (τ, a), but it is also possible to emit the items in reverse order. □

Example 22 (Key-based Partitioning). Consider the computation that maps a linearly ordered input sequence of data items of type T (each of which

contains a key), to a set of linearly ordered sub-streams, one per key. The function key : $T \to K$ extracts the key from each input value. The input type is specified by a singleton tag alphabet $\{\sigma\}$, the data value set T, and the dependence relation $\{(\sigma, \sigma)\}$. The output type is specified by the tag alphabet K, value types $T_k = T$ for every key $k \in K$, and the dependence relation $\{(k, k) \mid k \in K\}$. So, an input trace is represented as an element of T^*, and an output trace can is represented as a K-indexed tuple, that is, a function $K \to T^*$. The trace transduction $\text{partition}_{\text{key}} : T^* \to (K \to T^*)$ describes the partitioning of the input stream into sub-streams according to the key extraction map key: $\text{partition}_{\text{key}}(u)(k) = u|_k$ for all $u \in T^*$ and $k \in K$, where $u|_k$ denotes the subsequence of u that consists of all items whose key is equal to k. The unique implementation of this transduction can be represented as a function $f : T^* \to (K \times T)^*$ given by $f(\varepsilon) = \varepsilon$ and $f(wx) = (\text{key}(x), x)$ for all $w \in T^*$ and $x \in T$. □

Proposition 10 states that a set $T_1^* \times \cdots \times T_m^*$ can be isomorphically represented as a set of data traces, and also that the prefix relation on the traces corresponds via the isomorphism to the componentwise prefix order on $T_1^* \times \cdots \times T_m^*$. Theorem 23 then follows immediately.

Theorem 23. Every monotone function $F : T_1^* \times \cdots \times T_m^* \to U_1^* \times \cdots \times U_n^*$ can be represented as a data-trace transduction.

Another important case is when the input stream is considered to be unordered, therefore any finite prefix should be viewed as a multiset (relation) of data items. In this case, any reasonable definition of a streaming transduction should encompass the *monotone operations* on relations. Monotonicity implies that as the input relations get gradually extended with more tuples, the output relations can also be incrementally extended with more tuples. This computation is consistent with the streaming model, and it fits naturally in our framework.

Example 24 (Operations of Relational Algebra). First, we consider the relation-to-relation operations *map* and *filter* that are generalizations of the operations *project* and *select* from relational algebra. Suppose that the sets of input and output data traces are (up to a bijection) $\mathsf{Bag}(T)$. Given a function $op : T \to T$ and a predicate $\phi \subseteq T$, the trace transductions $map_{op} : \mathsf{Bag}(T) \to \mathsf{Bag}(T)$ and $filter_{\phi} : \mathsf{Bag}(T) \to \mathsf{Bag}(T)$ are defined by:

$$map_{op}(M) = \{op(a) \mid a \in M\} \qquad filter_{\phi}(M) = \{a \in M \mid a \in \phi\}$$

for every multiset M over T. The respective sequential implementations $\text{map}_{op} : T^* \to T^*$ and $\text{filter}_{\phi} : T^* \to T^*$ are defined as follows:

$$\begin{array}{cccc} \text{map}_{op}(\varepsilon) = \varepsilon & & \text{filter}_{\phi}(\varepsilon) = \varepsilon & \\ \text{map}_{op}(wa) = op(a) & & \text{filter}_{\phi}(wa) = a, \text{ if } a \in \phi & \text{filter}_{\phi}(wa) = \varepsilon, \text{ if } a \notin \phi \end{array}$$

for all $w \in T^*$ and $a \in T$.

Consider now the *relational join* operation for relations over T w.r.t. the binary predicate $\theta \subseteq T \times T$. An input data trace can be viewed as an element of $\mathsf{Bag}(T) \times \mathsf{Bag}(T)$, and an output trace as an element of $\mathsf{Bag}(T \times T)$. The trace transduction $\mathit{join}_\theta : \mathsf{Bag}(T) \times \mathsf{Bag}(T) \to \mathsf{Bag}(T \times T)$ is given by

$$\mathit{join}_\theta(M, N) = \{(a, b) \in M \times N \mid (a, b) \in \theta\} \text{ for multisets } M,\ N \text{ over } T.$$

Suppose the names of the input relations are `Q` and `R` respectively. An implementation of θ-join can be represented as a function $\mathrm{join}_\theta : \{(\mathtt{Q}, d), (\mathtt{R}, d) \mid d \in T\}^* \to (T \times T)^*$, given by $\mathrm{join}_\theta(\varepsilon) = \varepsilon$ and

$$\mathrm{join}_\theta(w\ (\mathtt{Q}, d)) = \mathrm{filter}_\theta((d, d_1)\ (d, d_2)\ \ldots\ (d, d_n)),$$

where $(\mathtt{R}, d_1)$, $(\mathtt{R}, d_2)$, $\ldots$, $(\mathtt{R}, d_n)$ are the `R`-tagged elements that appear in the sequence w (from left to right). The function join_θ is defined symmetrically when the input ends with a `R`-tagged element.

The relational operation that removes duplicates from a relation has a sequential implementation that can be represented as a function $\mathrm{distinct} : T^* \to T^*$, given by $\mathrm{distinct}(\varepsilon) = \varepsilon$ and

$$\mathrm{distinct}(wa) = \begin{cases} a, & \text{if } a \text{ does not appear in } w \\ \varepsilon, & \text{if } a \text{ appears in } w \end{cases}$$

for all $w \in T^*$ and $a \in T$. □

Example 24 lists several commonly used relational operations that can be represented as data-trace transductions. In fact, every monotone relational operation is representable, as stated in Theorem 25 below. The result follows immediately from Proposition 11.

Theorem 25. Every monotone operator $F : \mathsf{Bag}(T_1) \times \cdots \times \mathsf{Bag}(T_n) \to \mathsf{Bag}(T)$ can be represented as a data-trace transduction.

Consider now the important case of computing an *aggregate* (such as sum, max, and min) on an unordered input. This computation is meaningful for a static input relation, but becomes meaningless in the streaming setting. Any partial results depend on a particular linear order for the input tuples, which is inconsistent with the notion of an unordered input. So, for a computation of relational aggregates in the streaming setting we must assume that the input contains linearly ordered *punctuation markers* that trigger the emission of output (see [15] for a generalization of this idea). The input can then be viewed as an ordered sequence of relations (each relation is delineated by markers), and it is meaningful to compute at every marker occurrence an aggregate over all tuples seen so far. Our formal definition of data-trace transductions captures these subtle aspects of streaming computation with relational data.

Example 26 (Streaming Maximum). Suppose the input stream consists of unordered natural numbers and special symbols `#` that are linearly ordered. We

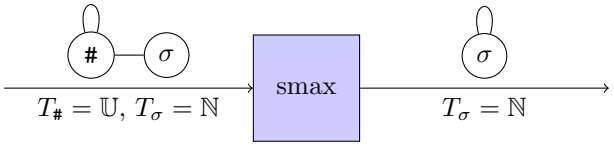


Fig. 4. Stream processing interface for streaming maximum (Example 26).

will specify the computation that emits at every # occurrence the maximum of all numbers seen so far. More specifically, the input type is given by $\Sigma = \{\sigma, \tau\}$, $T_\sigma = \mathbb{N}$, $T_\tau = \mathbb{U}$ (unit type), and $D = \{(\sigma, \tau), (\tau, \sigma), (\tau, \tau)\}$. So, an input data trace is essentially a nonempty sequence of multisets of numbers, i.e. an element of $\mathsf{Bag}(\mathbb{N})^+$. The correspondence between traces and elements of $\mathsf{Bag}(\mathbb{N})^+$ is illustrated by the following examples:

$$\begin{array}{ccc} \varepsilon \mapsto \emptyset & 1\ 2 \mapsto \{1,2\} & 1\ 2\ \#\ 3 \mapsto \{1,2\}\ \{3\} \\ 1 \mapsto \{1\} & 1\ 2\ \# \mapsto \{1,2\}\ \emptyset & 1\ 2\ \#\ 3\ \# \mapsto \{1,2\}\ \{3\}\ \emptyset \end{array}$$

The streaming maximum computation is described by the trace transduction $\mathrm{smax} : \mathsf{Bag}(\mathbb{N})^+ \to \mathbb{N}^*$, given as: $\mathrm{smax}(R) = \varepsilon$, $\mathrm{smax}(R_1 R_2) = \max(R_1)$, and

$$\mathrm{smax}(R_1 \dots R_n) = \max(R_1)\ \max(R_1 \cup R_2)\ \cdots\ \max(R_1 \cup R_2 \cup \cdots \cup R_{n-1}).$$

Notice that the last relation R_n of the input sequence is the collection of elements after the last occurrence of a # symbol, and therefore they are not included in any maximum calculation above. The sequential implementation of smax can be represented as a function $f : (\mathbb{N} \cup \{\#\})^* \to \mathbb{N}^*$, which outputs at every # occurrence the maximum number seen so far. That is, $f(\varepsilon) = \varepsilon$ and

$$f(a_1 a_2 \dots a_n) = \begin{cases} \varepsilon, & \text{if } a_n \in \mathbb{N}; \\ \max \text{ of } \{a_1, a_2, \dots, a_n\} \setminus \{\#\}, & \text{if } a_n = \#. \end{cases}$$

for all sequences $a_1 a_2 \dots a_n \in (\mathbb{N} \cup \{\#\})^*$. □

3 Operations on Data-Trace Transductions

In many distributed stream processing algorithms, the desired computation is passed through nodes which are composed *in parallel* and *in sequence*. Both *composition operations* can be implemented concurrently with potential time savings, and the decomposition makes this concurrency visible and exploitable. Crucial to the usefulness of an interface model, therefore, is that these composition operations correspond to semantic composition operations on the interfaces. In turn, given an interface for the overall computation, we may reason that only some distributed decompositions are possible.

In this section we define the sequential composition of two data-trace transductions, and the parallel composition of a (possibly infinite) family of data-trace

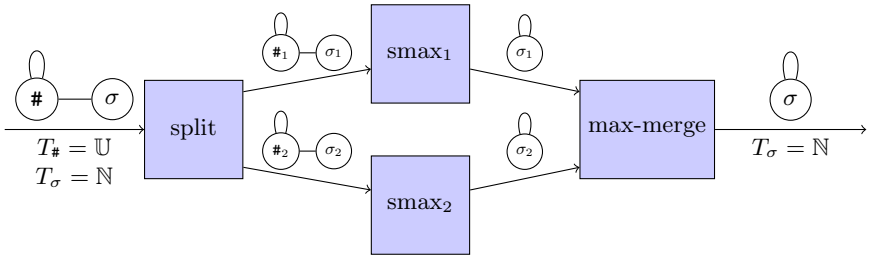


Fig. 5. Distributed evaluation of smax (Example 27).

transductions. We then define the implementation of both of these operations as operations on string transductions. A general block diagram, without feedback, can be obtained by the combination of (string or trace) transductions in sequence and in parallel.

Example 27 (Distributed evaluation of smax**).** For example, consider the problem of computing the maximum (smax), as in Example 26. We notice that the σ-tagged natural numbers are independent, forming a multiset, so the multiset could be split into multiple smaller sets and handled by separate components. To do so, we first have a component which copies the synchronization tags $\#$ into $\#_1$ and $\#_2$, and alternates sending σ tags to σ_1 and σ_2. This split component breaks the input stream into two independent streams. Next, components smax_1 and smax_2, which are instances of smax, handle each of the two input streams $\{\sigma_1, \#_1\}$ and $\{\sigma_2, \#_2\}$ separately, producing output tagged σ_1 and σ_2. Finally, a variant of Example 20, max-merge, can be constructed which takes one σ_1 and one σ_2 output and, rather than just producing both, maximizes the two arguments to produce a single σ tag. Altogether, the resulting block diagram of Figure 5 computes smax. We write:

$$\text{smax} = \text{split} \gg (\text{smax}_1 \parallel \text{smax}_2) \gg \text{max-merge},$$

which is an important sense in which computations like *Streaming Maximum* can be parallelized. □

Definition 28 (Sequential Composition). Let X, Y, and Z be data-trace types, and let $\alpha \in \mathcal{T}(X, Y)$ and $\beta \in \mathcal{T}(Y, Z)$ be data-trace transductions. The *sequential composition* of α and β, denoted $\gamma = \alpha \gg \beta$, is the data-trace transduction in $\mathcal{T}(X, Z)$ defined by $\gamma(\mathbf{u}) = \beta(\alpha(\mathbf{u}))$ for all $\mathbf{u} \in X$. □

Definition 29 (Parallel Composition). Let I be any index set. For each $i \in I$, let $X_i = (A_i, D_i)$ and $Y_i = (B_i, E_i)$ be trace types, and let $\alpha_i \in \mathcal{T}(X_i, Y_i)$. We require that for all $i \neq j$, A_i is disjoint from A_j and B_i is disjoint from B_j. Additionally, we require that $\alpha_i(\varepsilon) = \varepsilon$ for all but finitely many i.

Let $A = \bigcup_{i \in I} A_i$, $B = \bigcup_{i \in I} B_i$, $D = \bigcup_{i \in I} D_i$, and $E = \bigcup_{i \in I} E_i$. Let $X = (A, D)$ and $Y = (B, E)$. The *parallel composition* of all α_i, denoted $\alpha = \|_{i \in I} \alpha_i$,

is the data-trace transduction in $\mathcal{T}(X, Y)$ defined by $\alpha(\mathbf{u})|_{Y_i} = \alpha_i(\mathbf{u}|_{X_i})$, for all $u \in X$ and for all $i \in I$. Here $|_{Y_i}$ means the projection of a trace to Y_i. □

Definition 28 gives a well-defined trace transduction because the composition of monotone functions is monotone. In Definition 29, we have defined the value of $\alpha(\mathbf{u})$ by specifying the component of the output in each of the independent output streams Y_i. Specifically, a trace in Y is given uniquely by a trace in Y_i for each i, and the only restriction is that finitely many of the traces Y_i must be non-empty. Since each character in $\mathbf{u}$ only projects to one X_i and since $\alpha_i(\varepsilon) = \varepsilon$ for all but finitely many i, we satisfy this restriction, and parallel composition is well-defined. For only two (or a finite number) of channels, we can use the notation $f_1 \parallel f_2$ instead of $\|_{i \in I} f_i$.

Proposition 30 (Basic Properties). Whenever binary operations $\gg$ and $\parallel$ are defined, $\gg$ is associative and $\parallel$ is associative and commutative.

Definition 31 (Implementation of Sequential Composition). Let A, B, and C be data types. Let $f \in \mathcal{S}(A, B)$ and $g \in \mathcal{S}(B, C)$ be data-string transductions. The *sequential composition* of f and g, written $h = f \gg g$, is the unique data-string transduction in $\mathcal{S}(A, C)$ satisfying $\overline{h} = \overline{g} \circ \overline{f}$. I.e., $h(u) = \partial(\overline{g} \circ \overline{f})$. □

On input an item $a \in A$, we pass it to f, collect any result, and pass the result of that to g (if any). Because there may be multiple intermediate outputs from f, or none at all, this is most succinctly expressed by $\partial(\overline{g} \circ \overline{f})$.

Lemma 32. Let $X = (A, D)$, $Y = (B, E)$, $Z = (C, F)$ be data-trace types, $\alpha \in \mathcal{T}(X, Y)$, and $\beta \in \mathcal{T}(Y, Z)$. If $f \in \mathcal{S}(A, B)$ implements α and $g \in \mathcal{S}(B, C)$ implements β, then $f \gg g$ implements $\alpha \gg \beta$.

Definition 33 (Implementation of Parallel Composition). Let $(I, <)$ be an *ordered* index set. For each i, let A_i, B_i be data types and let $f_i \in \mathcal{S}(A_i, B_i)$ be a data-string transduction. As in Definition 29, we require that for all $i \neq j$, A_i is disjoint from A_j and B_i is disjoint from B_j; Also as in Definition 29, assume that $f_i(\varepsilon) = \varepsilon$ for all but finitely many i, say $i_1 < i_2 < \cdots < i_m$. Define $A = \bigcup_i A_i$ and $B = \bigcup_i B_i$. The *parallel composition* of all f_i, written $f = \|_{i \in (I,<)} f_i$, is the data-string transduction in $\mathcal{S}(A, B)$ defined as follows. First, $f(\varepsilon) = f_{i_1}(\varepsilon) f_{i_2}(\varepsilon) \cdots f_{i_m}(\varepsilon)$. Second, for all i, for all $a_i \in A_i$, and for all $u \in A^*$, $f(u a_i) = f_i(u|_{A_i} a_i)$, where $u|_{A_i}$ is the projection of u to A_i. □

We initially output $f_i(\varepsilon)$ for any i for which that is nonempty. Then, on input an item $a_i \in A_i$, we pass it to f_i, collect any result, and output that result (if any). Thus, while the definition allows an infinite family of string transductions, on a finite input stream only a finite number will need to be used.

The index set must be ordered for the reason that, on input ε, we need to produce the outputs $f_i(\varepsilon)$ in some order. By Theorem 19, any data-trace transduction can be implemented by some data-string transduction, and this construction picks just one possible implementation. Other than on input ε, the order does not matter. Regardless of the order, the following lemma states that we implement the desired data-trace transduction.

Lemma 34. Let I be an (unordered) index set. Let $X_i = (A_i, D_i)$, $Y_i = (B_i, E_i)$, and $\alpha_i \in \mathcal{T}(X_i, Y_i)$, such that the parallel composition $\alpha = \|_{i \in I} \alpha_i$ is defined. If $f_i \in \mathcal{S}(A_i, B_i)$ implements α_i for all i, then for *any* ordering $(I, <)$ of I, $f = \|_{i \in (I,<)} f_i$ implements α.

We now illustrate various examples of how these composition operations on trace transductions, which can be implemented as string transductions, can be used.

Example 35 (Partition by key, Reduce & Collect). Consider an input data stream of credit card transactions. For simplicity, we assume that each data item is simply a key-value pair (k, v), where k is a *key* that identifies uniquely a credit card account and v is the monetary value of a purchase. We write K to denote the set of keys, and V for the set of values. Suppose that the input stream contains additionally end-of-minute markers `#`, which indicate the end of each one-minute interval and are used to trigger output. We want to perform the following computation: "find at the end of each minute the the maximum total purchases associated with a credit card account". This computation can be structured into a pipeline of three stages:

1. *Stage* partition: Split the input stream into a set of sub-streams, one for each key. The marker items `#` are propagated to every sub-stream.

$$\begin{aligned} \text{input type} : & \text{ tags } K \cup \{\#\}, \\ & \text{values } T_k = V \text{ for every } k \in K \text{ and } T_\# = \mathbb{U}, \\ & \text{full dependence relation } (K \cup \{\#\}) \times (K \cup \{\#\}) \\ \text{output type} : & \text{ tags } K \cup \{\#_k \mid k \in K\}, \\ & \text{values } T'_k = V \text{ and } T'_{\#_k} = \mathbb{U} \text{ for every } k \in K, \\ & \text{dependencies } \textstyle\bigcup_{k \in K} (\{k, \#_k\} \times \{k, \#_k\}) \end{aligned}$$

   The definition of the data-trace transduction is similar to the one in Example 22, with the difference that `#` markers have to be propagated to every output channel.
2. For each key $k \in K$ perform a *reduce* operation, denoted reduce_k, that outputs at every occurrence of a `#` symbol the total of the values over the entire history of the k-th sub-stream. For reduce_k we have:

$$\begin{aligned} \text{input} &: \text{tags } \{k, \#_k\}, \text{ values } V \text{ and } \mathbb{U}, \text{ dependencies } \{k, \#_k\} \times \{k, \#_k\} \\ \text{output} &: \text{tags } \{k\}, \text{ values } V, \text{ dependencies } \{k\} \times \{k\} \end{aligned}$$

   The overall reduce stage is the parallel composition $\|_{k \in K} \text{reduce}_k$.
3. The outputs of all the reduce operations (one for each key, triggered by the same occurrence of `#`) are aggregated using a collect operation, which outputs the maximum of the intermediate results.

$$\begin{aligned} \text{input} &: \text{tags } K, \text{ values } V \text{ for each } k \in K, \text{ dependencies } \{(k, k) \mid k \in K\} \\ \text{output} &: \text{tags } \{o\}, \text{ values } V, \text{ dependencies } \{(o, o)\} \end{aligned}$$

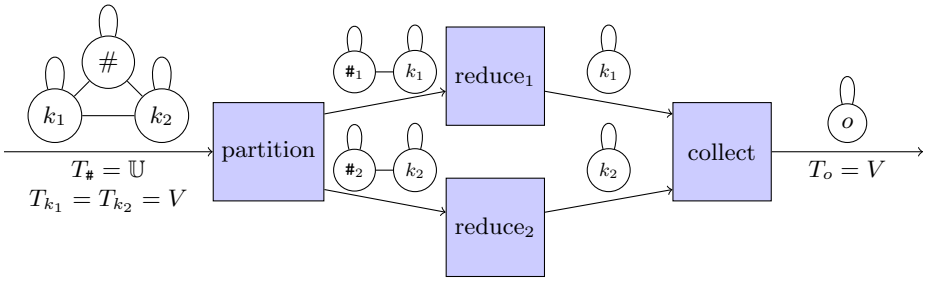


Fig. 6. Partition by key, reduce, and collect with two keys (Example 35).

The overall streaming map-reduce computation is given by a sequential composition with three stages: partition $\gg (\|_{k \in K}\ \text{reduce}_k) \gg$ collect.

Example 36 (Streaming Variant of Map-Reduce [9]). Suppose the input data stream contains key-value pairs, where the *input keys* K are partition identifiers and the values V are fragments of text files. The *intermediate keys* K' are words, and the corresponding values V' are natural numbers. The input stream contains additionally special markers `#` that are used to trigger output. The overall computation is the following: "output at every `#` occurrence the word frequency count for every word that appeared since the previous `#` occurrence". So, this is a *tumbling window* version of a map-reduce operation on a static data set [9]. The computation can be expressed as a pipeline of five stages:

1. *Stage* partition: Split the stream into a set of sub-streams, one for each input key. The marker items `#` are propagated to every sub-stream. This stage is similar to the one described in Example 35.
2. *Stage* map: Apply a function $\text{map} : K \times V \to (K' \times V')^*$ function to each key-value pair of the input stream. This function scans the text fragment and outputs a pair $(w, 1)$ for every word w that it encounters. The marker items `#` are propagated to every sub-stream. This stage is expressed as the parallel composition of transductions $\{\text{map}_k \mid k \in K\}$ with input/output types:

$$\begin{aligned} \text{input type} : &\text{ tags } \{k, \#_k\}, \\ &\text{values } V \text{ for the tag } k, \text{ and } \mathbb{U} \text{ for the tag } \#_k, \\ &\text{dependence relation } \{k, \#_k\} \times \{k, \#_k\} \\ \text{output type} : &\text{ tags } \{k, \#_k\}, \\ &\text{values } K' \times V' \text{ for the tag } k, \text{ and } \mathbb{U} \text{ for the tag } \#_k, \\ &\text{dependence relation } \{k, \#_k\} \times \{k, \#_k\} \end{aligned}$$

3. *Stage* reshuffle: The outputs from all map_k, $k \in K$, transductions between consecutive `#` occurrences are collected and reorganized on the basis of the

intermediate keys.

$$\begin{aligned}\text{input type} : &\text{ tags } K \cup \{\#_k \mid k \in K\},\\ &\text{values } K' \times V' \text{ for the every tag } k \in K, \text{ and } \mathbb{U} \text{ for every tag } \#_k,\\ &\text{dependence relation } \textstyle\bigcup_{k\in K}(\{k, \#_k\} \times \{k, \#_k\})\\ \text{output type} : &\text{ tags } K' \cup \{\#_{k'} \mid k' \in K'\},\\ &\text{values } V' \text{ for the every tag } k' \in K', \text{ and } \mathbb{U} \text{ for every tag } \#_{k'},\\ &\text{dependencies } \textstyle\bigcup_{k'\in K'}\{(k', \#_{k'}), (\#_{k'}, k'), (\#_{k'}, \#_{k'})\}\end{aligned}$$

4. *Stage* reduce: For each intermediate key $k' \in K'$ perform a *reduce* operation, denoted $\text{reduce}_{k'}$, that outputs at every occurrence of a # the total frequency count for the word k' since the previous occurrence of a # symbol. The data-trace types for reduce_k we have:

$$\begin{aligned}\text{input type} : &\text{ tags } \{k', \#_{k'}\},\\ &\text{values } V' \text{ for the tag } k', \text{ and } \mathbb{U} \text{ for the tag } \#_{k'},\\ &\text{dependencies } \{(k', \#_{k'}), (\#_{k'}, k'), (\#_{k'}, \#_{k'})\}\\ \text{output type} : &\text{ tags } \{k', \#_{k'}\},\\ &\text{values } V' \text{ for the tag } k', \text{ and } \mathbb{U} \text{ for the tag } \#_{k'},\\ &\text{dependencies } \{(k', \#_{k'}), (\#_{k'}, k'), (\#_{k'}, \#_{k'})\}\end{aligned}$$

   The overall reduce stage is the parallel composition $\text{reduce}_{k'_1} \parallel \cdots \parallel \text{reduce}_{k'_n}$, where $k'_1, \ldots, k'_n$ is an enumeration of the intermediate keys.
5. *Stage* collect: The outputs of all the reduce operations (one for each key, triggered by the same occurrence of #) are collected into a single multiset.

$$\begin{aligned}\text{input type} : &\text{ tags } K' \cup \{\#_{k'} \mid k' \in K'\},\\ &\text{values } V' \text{ for the every tag } k' \in K', \text{ and } \mathbb{U} \text{ for every tag } \#_{k'},\\ &\text{dependencies } \textstyle\bigcup_{k'\in K'}\{(k', \#_{k'}), (\#_{k'}, k'), (\#_{k'}, \#_{k'})\}\\ \text{output type} : &\text{ tags } \{o, \#\},\\ &\text{values } K' \times V' \text{ for the tag } o, \text{ and } \mathbb{U} \text{ for every tag } \#,\\ &\text{dependencies } \{(o, \#), (\#, o), (\#, \#)\}\end{aligned}$$

The overall streaming map-reduce computation is given by a sequential composition with five stages:

$$\text{partition} \gg (\|_{k\in K}\ \text{map}_k) \gg \text{reshuffle} \gg (\|_{k'\in K'}\ \text{reduce}_{k'}) \gg \text{collect}.$$

Example 37 (Time-Based Sliding Window). Suppose that the input is a sequence of items of the form (v, t), where v is a value and t is a timestamp. We assume additionally that the items arrive in increasing order of timestamps, that is, in an input sequence $(v_1, t_1) \cdots (v_n, t_n)$ it holds that $t_1 \leq \cdots \leq t_n$. We want to compute a so-called *moving aggregate*: "compute every second the sum of the values over the last 10 seconds". This sliding-window computation can be set up as a pipeline of three stages:

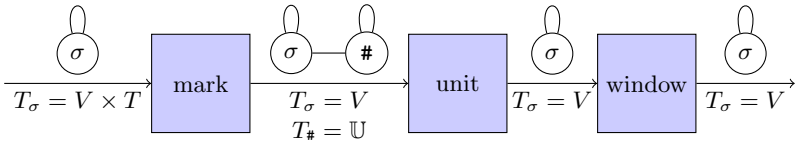


Fig. 7. Sliding window computation (Example 37).

1. *Stage* mark: Insert at the end of every second an end-of-second marker `#`.
2. *Stage* unit: Compute at every occurrence of a `#` marker the sum of the values over the last second. The output of this state is a sequence of partial sums, i.e. one value for each one-second interval.
3. *Stage* window: Compute with every new item the sum of the last 10 items.

The overall sliding window computation is expressed as: mark $\gg$ unit $\gg$ window.

4 Related Work

Synchronous Computation Models: The data-trace transduction model is a synchronous model of computation as it implicitly relies on the *synchrony hypothesis*: the time needed to process a single input item by the system is sufficiently small so that it can respond by producing outputs before the next input item arrives [6]. Data-trace transductions are a generalization of what is considered by acyclic *Kahn process networks* [11], where the interface consists of a a finite number of input and output channels. A process network consists of a collection of processes, where each process is a sequential program that can read from the input channels and write to the output channels. The input/output channels are modeled as first-in first-out queues. A specialization of process networks is the model of *Synchronous Dataflow* [14], where each process reads a fixed finite number of items from the input queues and also emits a fixed finite number of items as output. We accommodate a finite number of independent input or output streams, but also allow more complicated independence relations on the input and output, and in particular, viewing the input or output stream as a bag of events is not possible in Kahn process networks. We do not consider any particular implementation for data-trace transductions in this paper, but dataflow networks could be considered as a particular implementation for a subclass of our interfaces.

Merging of multiple input and output channels: In our model, even stream processing components with multiple input channels receive the items *merged* into a linear order. Traditionally, this merging of two streams into one linearly ordered stream has been considered a nondeterministic operation, and there is a body of work investigating the semantics and properties of dataflow systems built from such nondeterministic nodes. Brock and Ackerman [7] show that the relation

from inputs to possible outputs is not compositional, i.e. it is not an adequate semantics for these systems. Panangaden et al [19] consider variants of nondeterministic merge and their expressive power. Because we disallow these inherently nondeterministic merge operations, our semantics is simple and compositional. In particular the function from input histories to output histories is deterministic, and the nondeterminism of merge is hidden by expressing it only in the types, in the independence relation. We also have not considered feedback in a network defined by operations.

Partial order semantics for concurrency: The traditional model for asynchronous systems is based on *interleaving* the steps of concurrent processes, and the observational semantics of an asynchronous system consists of a set of behaviors, where a behavior is a (linear) sequence of interspersed input and output events (see, for instance, the model of I/O automata [16]). Such a semantics does not capture the distinction between coincidental ordering of observed events versus *causality* between them (see [13] for a discussion of causality in concurrent systems). This motivated the development of a variety of models with partial order semantics such as pomsets [20] and Mazurkiewicz traces [18]. We build upon the ideas in this line of research though our context, namely, synchronous, deterministic, streaming processors, is quite different.

Streaming extensions of database query languages: There is a large body of work on streaming query languages and database systems such as Aurora [2], Borealis [1], STREAM [5], and StreamInsight [3]. The query language supported by these systems (for example, CQL [5]) is typically a version of SQL with additional constructs for sliding windows over data streams. This allows for rich relational queries, including set-aggregations (e.g. sum, maximum, minimum, average, count) and joins over multiple data streams. A precise semantics for how to merge events from different streams has been defined using the notion of *synchronization markers* [15]. The pomset view central to the formalism of data-trace transductions is strictly more general, and gives the ability to view the stream in many different ways, for example: as a linearly ordered sequence, as a relation, or even as a sequence of relations. This is useful for describing streaming computations that combine relational operations with sequence-aware operations. Extending relational query languages to pomsets has been studied in [10], though not in the context of streaming.

5 Conclusion

We have proposed data-trace transductions as a mathematical model for specifying the observable behavior of a stream processing system. This allows consumption of inputs and production of outputs in an incremental manner that is suitable for streaming computation, while retaining the ability to view input and output streams as partially ordered multisets. The basic operations of sequential composition and parallel composition can be defined naturally on data-trace

transductions. The examples illustrate that the flexibility of our model is useful to specify the desired behavior of a wide variety of commonly used components in stream processing systems.

Defining the interface model is only the first step towards a programming system and supporting analysis tools that can help designers build stream processing systems with formal guarantees of correctness and performance. An immediate next step is to formalize a *transducer* model to define the computations of data-trace transductions with a type system that enforces the consistency requirement of Definition 17. Future directions include defining a declarative query language to specify data-trace transductions (see [10] for operations over pomsets and [4, 17] for specifying quantitative properties of linearly ordered streams), efficient implementation of data-trace transductions on existing high-performance architectures for stream processing (such as Apache Storm), and techniques for verifying correctness and performance properties of data-trace transductions.

References

1. D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.
2. D.J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.
3. M. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer. The extensibility framework in Microsoft StreamInsight. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, pages 1242–1253, 2011.
4. R. Alur, D. Fisman, and M. Raghothaman. Regular programming for quantitative properties of data streams. In *Programming Languages and Systems - 25th European Symposium on Programming*, LNCS 9632, pages 15–40, 2016.
5. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
6. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
7. J. D. Brock and W. B Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of programming concepts, International Colloquium*, LNCS 107, pages 252–259, 1981.
8. L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual ACM Symposium on Foundations of Software Engineering (FSE)*, pages 109–120, 2001.
9. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI '04, pages 137–149. USENIX Association, 2004.
10. S. Grumbach and T. Milo. An algebra of pomsets. *Information and Computation*, 150:268–306, 1999.

11. G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.
12. S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.
13. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.
14. E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
15. J. Li, D. Maier, K. Tufte, V. Papamidos, and P.A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 311–322, 2015.
16. N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
17. K. Mamouras, M. Raghothaman, R. Alur, Z.G. Ives, and S. Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proc. 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 693–708, 2017.
18. A. Mazurkiewicz. Trace theory. In *Advances in Petri nets: Proceedings of an advanced course*, LNCS 255, pages 279–324. Springer-Verlag, 1987.
19. P. Panangaden and V. Shanbhogue. The expressive power of indeterminate dataflow primitives. *Information and Computation*, 98(1):99–131, 1992.
20. V.R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1), 1986.