Modular Quantitative Monitoring

RAJEV ALUR, University of Pennsylvania
KONSTANTINOS MAMOURAS, University of Pennsylvania
CALEB STANFORD, University of Pennsylvania

In real-time decision making and runtime monitoring applications, declarative languages are commonly used as they facilitate modular high-level specifications with the compiler guaranteeing evaluation over data streams in an efficient and incremental manner. We introduce the model of Data Transducers to allow modular compilation of queries over streaming data. A data transducer maintains a finite set of data variables and processes a sequence of tagged data values by updating its variables using an allowed set of operations. The model allows unambiguous nondeterminism, exponentially succinct control, and combining values from parallel threads of computation. The semantics of the model immediately suggests an efficient streaming algorithm for evaluation. The expressiveness of data transducers coincides with streamable regular transductions, a robust and streamable class of functions characterized by MSO-definable string-to-DAG transformations with no backward edges. We show that the novel features of data transducers, unlike previously studied transducers, make them as succinct as traditional imperative code for processing data streams, but the structuring of the transition function permits modular compilation. In particular, we show that operations such as parallel composition, choice, prefix-sum, and quantitative analogs of combinators for unambiguous parsing, can be implemented by natural and succinct constructions on data transducers. To illustrate the benefits of such modularity in compilation, we define a new language for quantitative monitoring, QRE-Past, that integrates features of past-time temporal logic and quantitative regular expressions. While this combination allows a natural specification of a cardiac arrhythmia detection algorithm in QRE-Past, compilation of QRE-Past specifications into efficient monitors comes for free thanks to succinct constructions on data transducers.

1 INTRODUCTION

Applications ranging from network traffic engineering to runtime monitoring of autonomous control systems require computation over data streams in an efficient and incremental manner. Declarative programming is a particularly appealing approach to specify the desired logic in such applications as it can provide natural and high-level constructs for processing streaming data with guaranteed bounds on computational resources used by the compiled implementation. This has motivated the development of a number of declarative query languages. For example, in runtime verification, a monitor observes a sequence of events produced by a system, and issues an alert when a violation of a safety property is detected, where the safety property is described in a temporal logic with past-time operators such as always-in-the-past and since [Havelund and Roșu 2004; Manna and Pnueli 2012]. In quantitative monitoring, a monitor associates a numerical value with an input stream of data values, where the desired computation is described using quantitative regular expressions (QRE) that combine regular patterns with numerical aggregation operations such as min, max, sum, and average [Alur et al. 2016; Mamouras et al. 2017; Yuan et al. 2017]. In each such case, the declarative specification is automatically compiled into a monitor that adheres to the streaming model of computation [Muthukrishnan 2005]: memory and per-item processing time is polynomial in the size of the specification of the query and, roughly speaking, does not grow with the length of the input stream.

In existing query languages over streaming data, while a programmer can specify the desired computation in a modular fashion using constructs of the query language, the compiler generates...
monolithic code for a given query. What is lacking though is an intermediate representation for streaming computations that supports composition operations with succinct constructions so that high-level queries can be compiled modularly. The motivation for such a model is two-fold. From a practical viewpoint, it can facilitate the design of new query languages. For instance, suppose a user wants to specify a monitoring property using past-time temporal logic, where the atomic predicates involve comparing quantitative summaries defined naturally using QREs. Such a specification would be possible in a new query language that combines the features of QREs and past-time temporal logic, and while one would expect such queries to be amenable to streaming evaluation, one would have to design a compilation algorithm for the integrated query language. Such a compiler will be free though if we had a modular compilation algorithm for the combinators of the two component languages. From a theoretical viewpoint, designing such a representation is a technical challenge since it needs to support both parallel threads of computation and unambiguous regular parsing. In particular, QREs can potentially be compiled into a quantitative version of automata known as cost register automata [Alur et al. 2013], but since this compilation has provably exponential lower bound, it is not employed by the evaluation algorithm, and in fact, no existing formalism supports modular compilation of QREs. The main contribution of this paper is the model of Data Transducers (DT) as this desired modular intermediate representation for streaming computations.

A Data Transducer processes a data stream—a sequence of tagged data values, and produces a numerical (or Boolean) value using a fixed set of data variables that are updated using a constant number of operations as it processes each data item. A DT can be viewed as a natural quantitative generalization of (unambiguous) NFAs. Whereas an NFA configuration consists of a finite set of “tokens”, i.e. a set of active states, a DT configuration consists of a finite set of data variables, each of which can be inactive, active with a value, or in a special “conflict” mode. A DT configuration thus consists of succinctly represented finite control integrated with data values. As a DT computes by consuming data items, it updates its variables using a specified allowed set of operations. The values of active variables can be combined using operations to form new values, but there is also the possibility of a “collision”. This is analogous to how two tokens of active NFA states can be merged into one token during evaluation when they are placed on the same state. Since the merging of data values is not in general a meaningful operation, a collision of values results in a variable being set to “conflict” mode. Since multiple transitions can write to the same data variable while processing a single item, and the updated value of a variable can depend on the updated values of the others, the semantics is defined using fixed points. We show that how this semantics can be implemented by an efficient streaming algorithm for evaluation that executes linear (in the size of DT) number of data operations while processing each data item.

The domain of a DT, i.e. the set of stream histories for which its output is defined, is a regular language over the tags of the input stream. In fact, DTs capture a robust class of functions over data streams with an elegant logical characterization: MSO-definable transformations from strings to directed acyclic graphs with no backward edges. This class called streamable regular transductions has been recently studied in [Alur et al. 2018; Courcelle 1994; Engelfriet and Maneth 1999], and the closure properties of this class, as opposed to some specific constructs supported by query languages in the existing literature, guide the choice of operations over data transducers for which we seek (hopefully succinct) constructions.

In particular, we show that DTs are closed under the combinators quantitative concatenation, quantitative iteration, choice and parallel composition, and that the corresponding constructions are succinct. The design choices in the precise formal definition of the model turn out to be critical in these constructions. A key syntactic restriction on DTs, which we call restartability, that is required for constructions related to unambiguous parsing is identified. This restriction says that it is possible to “restart” the automaton during a computation by placing additional tokens on its initial states.
Then, the automaton output is the same as if multiple copies of the automaton were computing independently on multiple stream suffixes as long as only one of these copies ultimately contributes to the final output. This ability is necessary for efficient unambiguous parsing: several parsing possibilities are explored simultaneously, but the required space is bounded. We also consider the prefix-sum operation that combines the outputs on all prefixes using a specified aggregator, and this operation can be easily implemented on DTs. Temporal operators such as “always in the past”, “sometime in the past”, and “since” can be defined using prefix-sum and its variants.

To illustrate the benefits of modular compilation, we define a new query language, called QRE-Past, that combines the features of a past-time temporal logic and QREs. We specify a cardiac arrhythmia detection algorithm [Abbas et al. 2018] in QRE-Past to illustrate how the combination of features leads to a natural high-level specification. The theory of DTs immediately leads to a streaming evaluation algorithm for QRE-Past as every construct in QRE-Past maps to a corresponding construction on component DTs without causing a blow-up. In fact, there is nothing sacred about QRE-Past: a designer of a high-level query language over streaming data for a specific domain can introduce new combinators, in addition to the ones in this paper, as long as one can find corresponding succinct constructions on the low-level automaton-like model of DTs.

Data transducers define streamable regular transductions, and while there are existing models with identical expressiveness, DTs are exponentially more succinct (for instance, compared to unambiguous cost register automata). To gain a better understanding of the expressiveness and succinctness of DTs, consider a (generic) streaming algorithm that maintains a fixed number of Boolean and data variables, and processes each data item by updating these variables by executing a loop-free code. While such algorithms capture all streaming computations, the class of all streaming computations is not suitable for modular specifications. For instance, consider quantitative concatenation operation: given transductions $f$ and $g$, and a binary data operation $op$, $h = \text{split}(f, g, op)$ splits the inputs stream $w$ uniquely into two parts $w = w_1 w_2$ and returns $h(w) = op(f(w_1), g(w_2))$. While the class of streamable regular transductions is closed under this operation, the class of all streaming algorithms is not. We can enforce regularity of a generic streaming algorithm by requiring, for instance, that the updates to the Boolean variables are not influenced by the values of the data variables. We show that streaming algorithms with these restrictions can be translated to DTs without any blow-up, thus establishing that DTs are the most succinct (upto a constant factor) representation of streamable regular transductions. The structure of a DT—as variables ranging over data/undefined/conflict values and update code as a set of transitions of a particular form, as opposed to traditional loop-free update code, not only enforces regularity, but is also what allows us to define succinct constructions on the representation.

Outline of paper. Section 2 introduces the model of data transducers with illustrative examples. In Section 3 we consider a number of semantic operations with corresponding succinct constructions on DTs, and in particular, introduce and study the key concept of restartability necessary for some of them. In Section 4, we define the query language QRE-Past, and show how constructions on DTs immediately yield modular compilation into a streaming evaluation algorithm. We also show how the features of QRE-Past are useful in specifying a cardiac arrhythmia detection algorithm in a succinct and high-level manner. Section 5 is focused on expressiveness and succinctness of DTs. Section 6 contains a comparison with related works, and Section 7 concludes the paper.

2 DATA TRANSDUCERS

2.1 Preliminaries

Let $D$ be a (possibly infinite) set of data values, such as the set of integers or real numbers. Let $\Sigma$ be a finite set of tags. A data word [Bouyer et al. 2003] is a sequence of tagged data values, $w \in (\Sigma \times D)^*$. 

We write \( w \downarrow \Sigma \) to denote the projection of \( w \) to a string in \( \Sigma^* \). We use bold \( u, v \) and \( w \) denote data words. We reserve non-bold \( u, v, w \) for simple strings of tags in \( \Sigma^* \). We use \( x, y \) for un-tagged data vectors: elements of \( D^X \) for a finite set \( X \). We write \( d, d_i \) for elements of \( D \). We use \( \sigma \) to denote an arbitrary tag in \( \Sigma \), and in the examples we write particular tags in typewriter font, e.g. \( a, b \).

A signature is a tuple \((D, Op)\), where \( D \) is a set of data values and \( Op \) is a set of allowed operations. Each operation has an arity \( k \geq 0 \) and is a map from \( D^k \rightarrow D \). We use \( Op_k \) to denote the \( k \)-ary operations. For instance, if \( D \) is all 64-bit integers, we might support 64-bit arithmetic, as well as integer division and equality tests. Alternatively we might have \( D = \mathbb{N} \) with the operations + (arity 2), \( \min \) (arity 2), and 0 (arity 0). In general, we may have arbitrary user-defined operations on \( D \).

We define two special values in addition to the values in \( D \): \( \bot \) denotes undefined and \( T \) denotes conflict. We let \( \hat{D} := D \cup \{ \bot, T \} \) be the set of extended data values, and refer to elements of \( D \) in this context as defined. We lift the operations \( Op \) to operations on \( \hat{D} \) by thinking of \( \bot \) as the empty multiset, elements of \( D \) as singleton multisets, and \( T \) as any multiset of two or more items. The specific behavior of \( op \in Op \) on values in \( \hat{D} \) is given in Table 1, illustrated for the case \( op \in Op_2 \). It is also useful to define a union operation \( \sqcup : \hat{D} \times \hat{D} \rightarrow \hat{D} \): if one of its arguments is undefined it returns the other one, whereas if both are defined or conflicted it returns conflict. This represents multiset union: note that \( d_1 \sqcup d_2 = T \) even if \( d_1 = d_2 \). This is essential: it guarantees that for all operations on extended data values, whether the result is undefined, defined, or conflict can be determined from knowing only whether the inputs are undefined, defined, or conflict.

![Fig. 1. Behavior of operations on undefined (\( \bot \)) and conflict (\( T \)) values, for all \( d_1, d_2 \in D \). The set of extended data values \( \hat{D} = D \cup \{ \bot, T \} \) forms a complete lattice, in which these operations are monotone increasing.

\[
\begin{array}{c|c|c|c|c}
\text{\( \sqcup \)} & \bot & d_2 & T \\
\hline
\bot & \bot & d_2 & T \\
\hline
d_1 & d_1 & T & T \\
\hline
T & T & T & T
\end{array}
\quad
\begin{array}{c|c|c|c|c}
\text{\( op \)} & \bot & d_2 & T \\
\hline
\bot & \bot & \bot & \bot \\
\hline
d_1 & \bot & \text{\( op(d_1, d_2) \)} & T \\
\hline
T & T & T & T
\end{array}
\]

\( \hat{D} \) is a complete lattice, partially ordered under the relation \( \leq \) which is defined by \( \bot \leq d \leq T \) for all \( d \in D \), and distinct elements \( d, d' \in D \) are incomparable. This order extends coordinate-wise to an ordering on data vectors: thus we may write \( x \leq y \) for \( x, y \in D^X \). All operations in \( Op \) are monotone increasing with respect to this partial order. \( \sqcup \) is commutative and associative, with identity \( \bot \), and all \( k \)-ary operations distribute over it.

Given a signature \((D, Op)\), and a collection of variables \( Z \), the set of terms \( Tm[Z] \) consists of all syntactically correct expressions with free variables in \( Z \), using the set of operations \( Op \). For example: \( \min(x, 0) + \min(y, 0) \) and \( x + x \) are terms over the signature \((\mathbb{N}, \{+, \min, 0\})\) with \( Z = \{x, y\} \).

We have restricted to a single data type \( D \) for clarity of presentation. It is straightforward to define a generalized signature that includes multiple types and operations between them (or to allow multiple types which denote subsets of \( D \)), and to enforce typing restrictions in the model.

### 2.2 Syntax and semantics

Let \((D, Op)\) be a fixed signature. A data transducer (DT) is a 5-tuple \( \mathcal{A} = (Q, \Sigma, \Delta, I, F) \), where:

- \( Q \) is a finite set of state variables and \( \Sigma \) is a finite set of tags. We write \( Q' \) for a copy of the variables in \( Q \): for \( q \in Q, q' \in Q' \) denotes the copy.
- \( \Delta \) is a finite set of transitions, where each transition is a tuple \((\sigma, X, q', t)\). Here, \( \sigma \in \Sigma \cup \{1\} \), where \( 1 \notin \Sigma \), and if \( \sigma = 1 \) this is a special initial transition. \( X \subseteq Q \cup Q' \) is a set of source variables and \( q' \in Q' \) is the target variable. \( t \in Tm[X \cup \{\text{cur}\}] \) gives a new value of the target variable given values of the source variables and given the value of cur, which represents the current data value in the input data word. Assume that cur \( \notin X \). Additionally, for initial transitions, we require that \( X \subseteq Q' \) and that cur does not appear in \( t \).
• \( I \subseteq Q \) is a set of initial states and \( F \subseteq Q \) is a set of final states.

The number of states of \( \mathcal{A} \) is \( |Q| \). The size of \( \mathcal{A} \) is the number of states plus the total length of all transitions \((\sigma, X, q', t)\), which includes the length of description of all the terms \( t \).

**Semantics.** The input to a DT has two components. First, an initial vector \( x \in D^I \), which assigns an extended data value to each initial state. Second, an input data word \( w \in (\Sigma \times D)^* \), which is a sequence of tagged data values to be processed by the transducer. On input \((x, w)\), the DT’s final output vector is an extended data value at each of its final states. Thus, the semantics of \( \mathcal{A} \) will be

\[
[\mathcal{A}] : D^I \times (\Sigma \times D)^* \rightarrow D^F.
\]

A configuration is a vector \( c : Q \rightarrow D \). For any \( \sigma \in \Sigma \), the set of transitions \((\sigma, X, q', t)\) collectively define a function \( \Delta_\sigma : (Q \rightarrow D) \times D \rightarrow (Q \rightarrow D) \): given the current configuration and the current data value from the input data word, \( \Delta_\sigma \) produces the next configuration. We define \( \Delta_\sigma(c, d)(q) := c'(q') \), where \( c' : (Q \cup Q' \cup \text{cur}) \rightarrow D \) is the least vector satisfying \( c'(\text{cur}) = d \); for all \( q \in Q \), \( c'(q) = c(q) \); and for all \( q' \in Q' \),

\[
c'(q') = \bigwedge_{(\sigma, X, q', t) \in \Delta} t(c'(X)),
\]

where we define \( t(c'(X)) \) to be \( \perp \) if there is any variable \( x \in X \) such that \( c'(x) = \perp \), and otherwise, \( \top \) if there is any variable \( x \in X \) such that \( c'(x) = \top \). Thus, we produce \( \perp \) or \( \top \) if a variable in \( X \) is \( \perp \) or \( \top \), even if that variable is not present in \( t \). If all variables in \( X \) are defined, then \( t(c'(X)) \) is the value of the expression \( t \) with variables assigned the values in \( c' \). Since \( D \) is a complete lattice, the least fixed point exists by the Knaster-Tarski theorem.

The case of initial transitions (\( \Delta_1 \)) is slightly different. The purpose of initial transitions to compute an initial configuration \( c_0 : Q' \rightarrow D \), given the initial input vector \( x \in D^I \). There is no previous configuration \( Q \rightarrow D \), and no current data item, which is why we required \( X \subseteq Q' \) for initial transitions and \( \text{cur} \) was not allowed. We define the function \( \Delta_1 : D^I \rightarrow (Q \rightarrow D) \) with the same fixed point computation from Equation (1), except that the initial states are additionally assigned values given by the vector \( x \). Define that \( x(q) = \perp \) if \( q \notin I \). Then define \( \Delta_1(x) = c' \), where \( c' \) is the least vector satisfying, for all \( q \in Q \), \( c'(q') = x(q) \cup \bigwedge_{(T, X, q', t) \in \Delta} t(c'(X)) \).

Now \( \mathcal{A} \) is evaluated on input \((x, w) \in D^I \times (\Sigma \times D)^* \) by starting from the initial configuration and applying the update functions in sequence as illustrated in Figure 2. Finally, the output \( y \in D^F \) is given by \( y = c|_F \), the projection of \( c \) to the final states.

![Fig. 2. Example evaluation of a data transducer \( \mathcal{A} \) with two initial states and one final state on initial vector \((x_1, x_2)\) and an input data word \( w \) consisting of four characters: \((a, d_1), (b, d_2), (a, d_3), (a, d_4)\), to produce output \( y \). Here \( c_0, c_1, c_2, c_3 \), and \( c_4 \) are configurations; \( d_1 \in D \); and \( x_1, x_2, y \in D \). Each \( \Delta_\sigma \) is a set of transitions, collectively describing the next configuration in terms of the previous one.](image-url)
Why both initial states and initial transitions? Initial states give the data transducer the ability to take an initial vector \( x \) as input. Initial transitions give the data transducer the ability to compute initial values for its other states, including potentially producing output, before any characters of the data stream have been processed. If the desired computation does not require any initial input, then it is appropriate to take \( I = \emptyset \) and instead initialize only with initial transitions.

### 2.3 Streaming evaluation algorithm

**Complexity assumptions.** We are interested in evaluation in the streaming setting, where the input data word \( w \) arrives character-by-character and must be processed in real-time without being stored. After processing each character, we should produce the output on the current prefix of \( w \). The complexity bounds of interest in this setting are (1) the maximum time to process each element of \( w \) and (2) the maximum space usage while reading the entire word. Both should both be very small compared to the large input \( w \) (ideally constant or poly-logarithmic).

For our data transducers, we do not provide such unconditional bounds, as evaluation complexity depends on the particular data type \( \mathbb{D} \) and the operations that are supported. Instead, we provide a constant bound on (1) the number of data operations in \( \mathsf{Op} \) to process each element and (2) the number of data registers of type \( \mathbb{D} \) that need to be stored. In many cases, these can translate to efficient streaming complexity bounds when instantiated with a particular signature \((\mathbb{D}, \mathsf{Op})\). These bounds assume a sequential implementation.

The evaluation problem. We are given a data transducer \( \mathcal{A} \), an initial input \( x \), and a data word \( w \) presented as a data stream of elements. The problem at hand is to evaluate \( \mathcal{A}(x, w) \) in a streaming fashion. We know that we can process each element \((\sigma, d)\) of the stream by applying \( \Delta_\sigma \) with \(\text{cur} \) and the previous configuration as input, as in Figure 2. At each step we can produce the output \( y \) given by the extended data values of the final states. Therefore, the nontrivial remaining task is to compute \( \Delta_\sigma(c, d) \) from \( c \) and \( d \), for any \( \sigma \in \Sigma \cup \{\top, \bot\} \), which in particular means computing the least vector \( c' : Q \cup Q' \cup \{\text{cur}\} \to \mathbb{D} \) defined above.

We first discuss a naive approach. To compute \( c' \), first set \( c'(\text{cur}) = d \), \( c'(q) = c(q) \), and \( c'(q') = \bot \) for all \( q \in Q \). Then, repeatedly set \( c'(q') = \bigcup_{(\sigma, X, q', t) \in \Delta} t(c'(X)) \) for all \( q' \in Q' \) in parallel, until we reach a fixed point. Because the parallel assignment is a monotone increasing map from \( \mathbb{D}^{Q'} \) to itself, we reach the least fixed point after finitely many iterations (Kleene fixed-point theorem). In particular, this requires at most \( 2|Q'| \) iterations, since the partial order on \( \mathbb{D} \) has height 3. The number of operations in \( \mathsf{Op} \) that need to be applied for one iteration is bounded by the size of \( \mathcal{A} \). Thus if \( m \) is the size and \( n \) the number of states of \( \mathcal{A} \), we require \( 2mn = O(mn) \) data operations total. The number of data registers we need in memory at any time is \( |Q \cup Q' \cup \{\text{cur}\}| = O(n) \).

However, the fact that we repeatedly update all variables in parallel just to see whether one of them changes suggests that this algorithm is inefficient. Consider, instead, the following algorithm. We maintain, for each state \( q' \in Q' \) and each transition in \( \Delta_\sigma \), a current value for that state or transition (initially, \( \bot \)), as well as a triple of nonnegative integers \((a_\bot, a_\bot, a_\top)\). Similarly, for a transition \((\sigma, X, q', t)\), the triple stores the number of variables in \( X \) that are currently set to undefined, defined, and conflict, respectively. Similarly, for a state \( q' \), the triple stores the number of transitions \((\sigma, X, q', t)\) (transitions targetting \( q' \)) which are currently set to undefined, defined, and conflict, respectively. The idea is that the triples \((a_\bot, a_\bot, a_\top)\) can tell us whether the value of a state or transition has changed without having to re-calculate it, so that we only actually calculate the value of each state or transition at most 3 times.

Maintain a partition of \( Q' \cup \Delta_\sigma \) into an unvisited set \( U \) and a visited set \( V \), where initially everything is unvisited. Pick an unvisited state or transition, and calculate its value: \( t(c'(X)) \) for a transition, or \( \bigcup_{(\sigma, X, q', t) \in \Delta} t(c'(X)) \) for a vertex. Then set this state or transition to visited. If it’s a...
transition and its value changed, update \((a_\perp, a_\ast, a_\uparrow)\) for the state it targets, and if this causes the state’s value to change, set it to unvisited. (For example, if a triple goes from \((2, 0, 0)\) to \((1, 1, 0)\), the value of the state changes from undefined to defined.) Similarly, if it’s a state and its value changed, update \((a_\perp, a_\ast, a_\uparrow)\) for each transition which contains it as a source variable, and if this causes that transition’s value to change, set it to unvisited. Continue picking vertices until there are no unvisited states or transition, and then return the configuration \(e'\). Observe that the number of times we calculate the value of each state or transition is bounded above by the number of times it is set to unvisited, which is at most 3. The amount of time we spend updating all states and transitions, plus the time we spend updating triples \((a_\perp, a_\ast, a_\uparrow)\), is therefore \(O(m)\). Finally, if all states are visited then we have reached the fixed point. This proves the following theorem.

**Theorem 2.1.** Evaluation of a data transducer \(A\), with number of states \(n\) and size \(m\) on input \((x, w)\), requires \(O(n)\) data registers to store the state, and \(O(m)\) operations and additional data registers to process each element in \(\Sigma \times D\), independent of \(w\).

An instructive special case is when the transitions \(\Delta_\sigma\), for all \(\sigma \in \Sigma \cup \{1\}\), are acyclic. By this we mean that the following directed graph is acyclic: take vertices \(Q \cup Q'\), with an edge from \(x\) to \(y\) if there is a transition \((\sigma, X, q', t)\) with \(x \in X\). If the transitions \(\Delta_\sigma\) are acyclic for all \(\sigma\) then we say this is an **acyclic data transducer**. Then the least fixed point is also the only fixed point and is obtained by iterating over the states \(q' \in Q'\) in a single pass, in a topologically sorted order, and assigning the value of \(e'(q')\). This is an \(O(m)\) algorithm, so we get the result of Theorem 2.1 for this special case much more easily than before.

### 2.4 Examples

We do not envision that the machine model of data transducers would be directly programmed by users, for a number of reasons, including the conceptual challenge of programming with undefined, defined, and conflicted state variables. Rather, data transducers would be a low-level, back-end model for general-purpose stream processing. The purpose of this section is mainly to illustrate, informally and through examples, the basic features and execution semantics of the model.

We present only acyclic data transducers in this section, and we take \(I = \emptyset\): all initialization is done with initial transitions \(\Delta_1\). Additionally, we use the abbreviation \(q' := t\) to denote a transition \((\sigma, X, q', t)\), where \(X\) is exactly the set of variables present in the term \(t\) (and in contexts where \(\sigma\) is clear). In general, \(X\) may include other variables unused in \(t\), and the semantics of the transition does depend on the unused variables as well (see formal definition in §2.2).

**Pattern matching.** DTs are based on the idea of merging data registers and finite control into the single set of “state variables” \(Q\). Suppose we wish to monitor a stream of \(a\)-events, \(b\)-events, and \#-events, where each \(a\)- or \(b\)-event is the price at which an item was bought, and \# indicates the end of a day. We thus have \(D = Q\) and \(\Sigma = \{a, b, \#\}\). For the operations \(O_p\), we allow +, −, ·, max, min, division / (this must return a default value on division by 0), and integer constants. Suppose we want to output the average price of a sliding window containing the last three \(a\) prices, which resets at the end of the day. This is essentially a pattern match over the input tags to locate the last three, which are then averaged. \(A_1\) in Figure 3 is based on this idea. The transitions listed under transitions(\(\sigma\)) are those labeled with \(\sigma\); we use \(\parallel\) to emphasize that the transitions are not ordered.

The machine \(A_1\) uses state variables \(\text{sum}1\), \(\text{sum}2\), and \(\text{sum}3\) to keep track of the sum of the last 1, 2, and 3 \(a\)’s so far. In addition to pattern-matching, the variables are updated to keep track of the sum. For example, the transition \(\text{sum}2' := \text{sum}1 + \text{cur}\) indicates that if \(\text{sum}1\) was defined before then \(\text{sum}2\) should now be defined and equal to the sum...
plus the current data item. The transition \( \text{avg}' := \text{sum3}' / 3 \) indicates that if \( \text{sum3} \) is now defined (note the the \( \text{sum3}' \)), then \( \text{avg} \) should be set to the average of the last three prices.

\[
Q = \{\text{sum1}, \text{sum2}, \text{sum3}, \text{avg}\}, \ I = \emptyset, \ F = \{\text{avg}\}
\]

| transitions(i) | \( \text{transitions(a)} = || \text{sum1}' := \text{cur} \) |
|---------------|--------------------------------------------------|
|               | \( || \text{sum2}' := \text{sum1} + \text{cur} \) |
|               | \( || \text{sum3}' := \text{sum2} + \text{cur} \) |
|               | \( || \text{avg}' := \text{sum3}' / 3 \)          |
| transitions(b) | \( || \text{sum1}' := \text{sum1} \) |
|               | \( || \text{sum2}' := \text{sum2} \) |
|               | \( || \text{sum3}' := \text{sum3} \) |
| transitions(#) | \( \emptyset \) |

Example evaluation on input

\[
\begin{array}{cccc}
\text{w} = (a, 6)(a, 5)(a, 7)(b, 2)(a, 8)(\#)(0)(b, 2)(a, 7).
\end{array}
\]

\[
\begin{array}{cccc|c}
\text{w} (\text{input}) & \text{sum1} & \text{sum2} & \text{sum3} & \text{avg} (\text{output}) \\
\hline
(a, 6) & 6 & \bot & \bot & \bot \\
(a, 5) & 5 & 11 & \bot & \bot \\
(a, 7) & 7 & 12 & 18 & 6.000 \\
(b, 2) & 7 & 12 & 18 & \bot \\
(a, 8) & 8 & 15 & 20 & 6.667 \\
(\#, 0) & \bot & \bot & \bot & \bot \\
(b, 2) & \bot & \bot & \bot & \bot \\
(a, 7) & 7 & \bot & \bot & \bot \\
\end{array}
\]

Fig. 3. Data transducer \( \mathcal{A}_1 \) monitoring a stream of purchase events for two types of items, tagged a and b, and # to represent the end of each day. Throughout the day we output the average price in a sliding window of the last three a-items. The language of strings on which \( \mathcal{A}_1 \) produces output is \((a \cup b \cup \#)^*ab^*ab^*a\).

**Multiple transitions with a single target.** The machine \( \mathcal{A}_1 \) has a simplifying syntactic property that for every \( \sigma \in \Sigma \) and for every state \( q' \), there is only one transition \( q' := t \). In other words, there is only one rule stating how to assign \( q' \) a value. In general, there may be multiple rules, and the resulting value of \( q' \) will be the union (\( \cup \)) over all transitions. For instance, suppose we have the same input stream over \( \Sigma = \{a, b, \#\} \), and we want to output the average price of an a-item at the end of each day. However, if there are no a-items on a given day, we instead want to output the average from the previous day. A machine implementation of this is provided by \( \mathcal{A}_2 \) in Figure 4.

\[
Q = \{\text{sum}, \text{count}, \text{avg}, \text{prev_avg}\}, \ I = \emptyset, \ F = \{\text{avg}\}
\]

| transitions(i) | \( \text{transitions(a)} = || \text{prev_avg}' := 0 \) |
|---------------|--------------------------------------------------|
|               | \( || \text{sum}' := \text{prev_avg} \cdot 0 + \text{cur} \) |
|               | \( || \text{sum}' := \text{sum} + \text{cur} \) |
|               | \( || \text{count}' := \text{prev_avg} \cdot 0 + 1 \) |
|               | \( || \text{count}' := \text{count} + 1 \) |
| transitions(b) | \( \text{transitions(b)} = || \text{sum}' := \text{sum} \) |
|               | \( || \text{count}' := \text{count} \) |
|               | \( || \text{prev_avg}' := \text{prev_avg} \) |
| transitions(#) | \( \text{transitions(#) = || avg' := \text{sum}/\text{count} \) |
|               | \( || \text{avg}' := \text{prev_avg} \) |
|               | \( || \text{prev_avg}' := \text{avg}' \) |

Fig. 4. Data transducer \( \mathcal{A}_2 \) monitoring the stream to produce, at the end of each day, either the average price of an a-item (if there was at least one a) or the previous average (if there was no a). When there are multiple transitions \( q' := t_1 \) and \( q' := t_2 \), the semantics is such that we assign \( q' := t_1 \cup t_2 \).

In \( \mathcal{A}_2 \), \text{sum} and \text{count} store the sum of a-items and number of a-items on each day, respectively, and are defined only if there has been at least one a. On the other hand, \text{prev_avg} stores the previous average, but it is defined only if there has not been any a yet. (We also initialize this to 0 arbitrarily on the very first day.) The state \text{avg} stores the output, and is only defined after a \# character. The logic of this computation involves two places where we need to have multiple
transitions targeting a state. First, on receiving an a, we set sum to be equal to the previous sum plus the current value, but we also set it to be equal to 0 • prev_avg + cur. This works because exactly one of these two values will be defined, and the other will be ⊥: either we have seen an a already, in which case we can update the sum, or we haven’t seen one yet, in which case prev_avg is still defined. Second, the overall output avg has two possible values, either sum/count or prev_avg, and again, exactly one of these two values will be defined, and the other will be ⊥. Thus, we have designed A2 so that each ⊔ never produces a conflict ⊤.

Combining output from parallel threads of computation. Our final example attempts to illustrate the feature which gives data transducers their succinctness (see §5): the ability to update multiple threads independently and then combine their results. Suppose we want to compute, at the end of each day, the difference between the maximum price of a and the maximum price of b, if there was at least one a and at least one b. The data transducer A3 in Figure 5 implements this computation. The state a_init of A3 stores 0 and is only defined if we haven’t seen an a yet; similarly for b_init.

![Fig. 5. Data transducer A3 monitoring the stream to produce, at the end of each day, the difference between the maximum price of a item and the maximum price of a b-item.](image)

**2.5 Regularity**

Data transducers define regular functions on data words. Whether the output is defined (or undefined, or conflict) is a regular property in the sense that it depends only on whether the input values are undefined, defined, or conflict, together with some regular property of the sequence of tags. For data vectors x1, x2 ∈ D^X, we say that x1 and x2 are equivalent, and write x1 ∼= x2, if for all x ∈ X, x1(x) and x2(x) are both undefined, both defined, or both conflict. We get the following theorem:

**Theorem 2.2.** Let A = (Q, Σ, Δ, I, F) be a data transducer over (D, Op). Then: (i) For all initial vectors x1, x2 ∈ D^I, and for all input words w1, w2, if x1 ∼= x2 and w1 ⊥ Σ = w2 ⊥ Σ, then [A](x1, w1) ∼= [A](x2, w2). (ii) For any equivalence class of input vectors x and equivalence class of output vectors y, the set of strings w ⊥ Σ such that [A](x, w) ∼= y is regular.

**Proof.** In evaluating a data transducer we may collapse all values in D to a single value •, so each state takes values in {⊥, •, ⊤}. This gives a projection from A to a data transducer P over the unit signature (⊔, UOp), where U = {•} is a set with just one element, and UOp consists of, for each k, the unique map opk : U^k → U. The projection homomorphically preserves the semantics. Then, (i) follows because the computation of P is exactly the same on x1, w1 and x2, w2, and (ii) follows because P has finitely many possible configurations. □

We can thus define the language of A to be L(A) = {w ⊥ Σ | [A](x, w) ∈ D^F for some x ∈ D^I}, so L(A) ⊆ Σ^* . This is the set of tag strings w ⊥ Σ such that, if the initial vector of values is all defined, after reading in w all final states are defined. We similarly define the set of strings on
which a data transducer is defined or conflict, on input of the same form: the extended language $\overline{L}(A)$ is $\{w \downarrow \Sigma \mid [\mathcal{A}(x, w) \in (\mathcal{D} \cup \{\top\})^I \text{ for some } x \in (\mathcal{D} \cup \{\top\})^I\}$. An immediate corollary of Theorem 2.2 is that (i) $L(A)$ is regular, (ii) $\overline{L}(A)$ is regular, and (iii) $L(A) \subseteq \overline{L}(A)$. Finally, say that DTs $A_1$ and $A_2$ are equivalent if for all $x_1 \equiv x_2$ and for all $w$, $[A_1(x_1, w)] \equiv [A_2(x_2, w)]$.

**Theorem 2.3.** On input $A_1$, $A_2$, deciding whether $A_1$ and $A_2$ are equivalent is PSPACE-complete.

**Proof.** We first decide if the two are not equal in NPSPACE. It suffices to project $A_1$ and $A_2$ to data transducers over the unit signature, $P_1$ and $P_2$, as in the previous proof, and decide if $P_1 \neq P_2$. Let $n$ be the number of states between $P_1$ and $P_2$, and let $m$ be their combined size. The number of configurations for $P_1$ and $P_2$ together is $3^n$. Therefore, if there is a counterexample, it occurs in a string of length at most $3^n$. Guessing the counterexample one character at a time requires linear in $n$ space to record the count and $O(m)$ space to update $P_1$ and $P_2$ (by Theorem 2.1).

To show it is PSPACE-hard, it suffices to exhibit a translation from NFAs to data transducers which reduces language equality of NFAs to equivalence of data transducers. Specifically, we conflicted on strings for which the NFA is defined. The translation works by directly copying the

**3 CONSTRUCTIONS ON DATA TRANSDUCERS**

As discussed in the introduction, our primary interest in the DT model is to support a variety of succinct composition operations which are not simultaneously supported by any existing model. In particular, such composition operations can provide support for a quantitative monitoring language like QRE-PAST in §4: language constructs can be implemented by the compiler as constructions on DTs, rather like how (true) regular expressions are compiled to nondeterministic finite automata.

For example, suppose we have DTs implementing two functions $f, g : (\Sigma \times D)^* \rightarrow \overline{D}$, and we would like to implement the function $f + g$, which applies $f$ and $g$ to the input stream and adds the results. To do so, we copy the states of the transducers for $f$ and $g$, and we initialize and update the states in parallel (they do not interfere). Then, we provide a new output state, and a single new transition which says that the new final state should be assigned the value of the final state of $f$ plus the value of the final state of $g$. This works for any operation, and not just $+$: the combination of $k$ computations by applying a $k$-ary operation $op \in Op_k$ can be implemented by a corresponding $k$-ary construct on the $k$ underlying DTs. Moreover, the size of the DT will only be the sum of the sizes of the $k$ DTs, plus a constant. In contrast, even an operation as simple as $f + g$ is not succinctly implementable using the most natural existing alternative to DTs, Cost-Register Automata (see §5).

This construction for $f + g$ requires no assumptions about the DTs implementing $f$ and $g$. However, not all operations are this straightforward. Consider the following quantitative generalization of concatenation. Given $f : (\Sigma \times D)^* \rightarrow \overline{D}$, $g : (\Sigma \times D)^* \rightarrow \overline{D}$, and $op \in Op_2$, we wish to implement $split(f, g, op)$: on input $w$, split the input stream into two parts, $w = u \cdot v$, such that $f(u) \neq \perp$ and $g(v) \neq \perp$ (respectively, $f$ matches $u$ and $g$ matches $v$), and return $op(f(u), g(v))$. Assume that the decomposition of $w$ into $u$ and $v$ such that $f(u) \neq \perp$ and $g(v) \neq \perp$ is unique. In order to implement this operation, on an input string $w$, we must not only keep track of the current configuration of $f$ on $w$, but for every split $w = uv$ where $f$ matches $u$, we must keep track of the current configuration of $g$ on $v$. If there are many possible prefixes $u$ of $w$ such that $f(u) \neq \perp$, we may have to keep arbitrarily many configurations of $g$. This naive approach is therefore impossible using only the finite space that a DT allows, if we treat $f$ and $g$ only as black boxes.

What we need is an additional structural condition on $g$. Rather than keeping multiple copies of $g$, we would like to perform every computation using the same copy: whenever the current
prefix matches \( f \), restart \( g \) with a new thread of computation (keeping the old threads as well). To motivate this idea, consider the analogous concatenation construction for two NFAs: every time the first NFA accepts, we are able to “restart” the second NFA by adding a token to its start state (we don’t need an entirely new NFA every time). This property for DTs is called restartability. Restartable DTs are an equally expressive subclass consisting of those DTs for which restarting computation on the same transducer does not cause interference in the output.

The set of strings that a DT “matches” is captured by its extended language, defined in §2.5. Correspondingly, we assume that whenever a DT is restarted, the new initial input is either all \( \bot \), or all not \( \bot \) (in \( \mathbb{D} \cup \{ \top \} \)). If the output of a DT also satisfies this property (on any input it is either all \( \bot \), or all not \( \bot \)), then we say that the DT is output-synchronized. This property is required in the concatenation and iteration constructions, but it is not as crucial to the discussion as restartability.

We begin in §3.1 by giving general constructions that do not rely on restartability. We highlight the implemented semantics, the extended language, and the size of the constructed DT in terms of its constituent DTs. Then in §3.2, we define restartability and use it to give succinct constructions for unambiguous parsing operations, namely concatenation and iteration. Moreover, we show that (under certain conditions) our operations preserve restartability, thus enabling modular composition using the restartable DTs. We also show that checking restartability is hard (PSPACE-complete), and we mention converting a non-restartable DT to a restartable one, but with exponential blowup.

### 3.1 General constructions

**Notation.** It is convenient to introduce shorthand for a transition which is evaluated on initialization and on every input character. We write \( (\epsilon, X, q', t) \) as shorthand for the union of \(|\Sigma| + 1 \) transitions: \( (\sigma, X, q', t) \) for every \( \sigma \in \Sigma \cup \{ \bot \} \). Because this includes an initial transition, this requires that \( X \subseteq Q' \) and that \( \text{cur} \) does not appear in \( t \). We call such a collection of transitions an epsilon transition because, like epsilon transitions from classical automata, the transition may require that \( X \) and its output-synchronized successors are output-synchronized. We wish to define a DT which feeds the input \( I \) into both DTs in parallel. To do so, we define \( \Delta \) as shorthand for the union of \(|\Sigma| + 1 \) transitions: \( (\epsilon, i', \pi_1(i'), i') : i \in I \) \( \cup \) \( \{(\epsilon, i', \pi_2(i'), i') : i \in I \} \).

Here, the transitions we have added (those in \( \Delta \) but not in \( \Delta_1 \) or \( \Delta_2 \)) copy values from \( I \) into both \( I_1 \) and \( I_2 \). This is only relevant on initialization \( \Delta_1 \), since after that states \( I \) will not be defined. The reason we have used an epsilon transition instead of just a \( i \) transition is to preserve restartability, which will be discussed in §3.2. Since we have added no other transitions between states in \( Q_1 \) and
and states in $Q_2$, the least fixed point (Equation 1) defining the next (or initial) configuration will always decompose into the least fixed point on states $Q_1$, and the least fixed point on states $Q_2$. It follows that the semantics satisfies $A(x, u) = (A_1(x, u), A_2(x, u))$. Here, $(y_1, y_2)$ denotes the vector $y : F \rightarrow \overline{D}$ that is $y_1$ on $F_1$ and $y_2$ on $F_2$. Parallel composition is commutative and associative.

**Parallel composition.** If $A_1 : I \rightarrow F_1$ and $A_2 : I \rightarrow F_2$, then $A_1 \parallel A_2 : I \rightarrow F_1 \cup F_2$ implements the semantics

$$[A_1 \parallel A_2](x, w) = ([A_1](x, w), [A_2](x, w)),$$

such that size($A_1 \parallel A_2$) = size($A_1$) + size($A_2$) + $O(|I|)$. It therefore matches the set of tag strings $\overline{L}(A_1 \parallel A_2) = \overline{L}(A_1) \cap \overline{L}(A_2)$.

The utility of parallel composition is that it allows us to combine the outputs $y_1$ and $y_2$ later on. This is accomplished by concatenation with another DT which combines the outputs (§3.2).

**Union.** Suppose we are given DTs $A_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $A_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, and assume that the sets of initial and final states are the same up to some bijections: $\pi_1 : I \rightarrow I_1, \pi_2 : I \rightarrow I_2, \rho_1 : F \rightarrow F_1, \rho_2 : F \rightarrow F_2$, for sets $I$ and $F$ with $|I| = |I_1| = |I_2|$ and $|F| = |F_1| = |F_2|$. We wish to define a DT which feeds the input $(x, w)$ into both DTs in parallel and returns the union ($\cup$) of the two results. We define $A = A_1 \cup A_2 = (Q, \Sigma, \Delta, I, F)$ by $Q = Q_1 \cup Q_2 \cup I \cup F$ and

$$\Delta = \Delta_1 \cup \Delta_2 \cup \{((i, \epsilon), \pi_1(i'), i') : i \in I\} \cup \{((i', \epsilon), \pi_2(i'), i') : i \in I\}$$

$$\cup \{((i', \rho_1(f'), f'), \rho_2(f')) : f \in F\} \cup \{((i, \rho_2(f'), f', \rho_2(f')) : f \in F\}.$$

Similar to the parallel composition construction, the additional transitions here ensure that we copy values from $I$ into $I_1$ and $I_2$, and copy values from $F_1$ and $F_2$ into $F$, whenever these values are defined. In particular, on initialization the initial vector $x$ will be copied into $I_1$ and $I_2$, and on any data word the output values $y_1$ and $y_2$ of $A_1$ and $A_2$ will be copied into the same set of final states, so that they have to be joined by $\cup$. In particular, if both $y_1$ and $y_2$ are defined, the output will be $\top$. We see therefore that the semantics is such that $A(x, u) = A_1(x, u) \cup A_2(x, u)$. Like parallel composition, union is commutative and associative.

**Union.** If $A_1 : I \rightarrow F$ and $A_2 : I \rightarrow F$, then $A_1 \cup A_2 : I \rightarrow F$ implements the semantics

$$[A_1 \cup A_2](x, w) = [A_1](x, w) \cup [A_2](x, w),$$

s.t. size($A_1 \cup A_2$) = size($A_1$) + size($A_2$) + $O(|I| + |F|)$. It matches $\overline{L}(A_1 \cup A_2) = \overline{L}(A_1) \cup \overline{L}(A_2)$.

**Prefix summation.** Now we consider a more complex operation. Suppose we are given $A_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$, and a data word $w$, such that the output on the empty data word is $y^{(0)}$, the output after receiving one character of the data word is $y^{(1)}$, and in general the output after $k$ characters is $y^{(k)}$. The problem is to return the sum of these outputs: we want a DT that returns $y^{(0)} + \cdots + y^{(i)}$ after receiving $i$ characters. This is called the prefix sum because $y^{(k)}$ is the value of $A$ on the $k$th prefix on the data word.

In general, instead of $+$, we can take an arbitrary operation which folds the outputs of $A$ on each prefix. We suppose that this operation given by a data function $G$ which, for some set $F$, is a function $G : \overline{D} \rightarrow \overline{D}$. It takes the previous “sum” $y^{(i-1)} \in \overline{D}$, combines it with the new output of $A_1$, $y_1^{(i)} \in \overline{D}$, and produces the next “sum” $y^{(i)} \in \overline{D}$. So, we’ll have $[A_2](y^{(i-1)}, y_1^{(i)}) = y^{(i)}$. We then want to compute the DT that, on input initial values for $I_1$ and also initial values $y^{(-1)}$ for $F$, will return $y^{(i)}$. 

Formally, we convert $G$ to a DT $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, with bijections $\pi : (F \cup F_1) \to I_2$, $\rho : F \to F_2$, which only contains epsilon-transitions: for each term $t$ giving a value of $f_2 \in F_2$ in terms of some variables $P \subseteq I_2$, we create a single epsilon transition $(\epsilon, P', f'_2, t)$. Then we define the prefix sum $\oplus_G \mathcal{A}_1 = (Q, \Sigma, \Delta, (I_1 \cup F), F_2)$, where $Q = Q_1 \cup Q_2 \cup F$ and

$$\Delta = \Delta_1 \cup \Delta_2 \cup \{(\epsilon, f'_1, \pi(f'_1), f'_1) : f_1 \in F_1\}$$

$$\cup \{\epsilon, f', \pi(f'), f' : f \in F\} \cup \{(\sigma, \rho(f'), \pi(f'), \rho(f') : f \in F, \sigma \in \Sigma\}.$$

First on the empty data word, we have that variables $F'$ are set to the initial values; then the outputs $F'_2$ of $\mathcal{A}_1$ and the variables $F'$ are copied into $I_2$, and $\mathcal{A}_2$ produces the correct output $y_1^{(0)} = G(y^{(-1)}_1, y_1^{(0)})$. Now, when we read in a character in $\Sigma \times \mathbb{D}$, the output states $F'_2$ flow back into inputs to $\mathcal{A}_2$, and the new output of $\mathcal{A}_1$ also flows in. Because the machine $\mathcal{A}_2$ was constructed to be just a set of epsilon-transitions from $I_2$ to $F_2$, it does not save any internal state, but just computes the output in terms of the input again. So the next output will be $G(y_1^{(1)}, y_1^{(2)})$, and then $G(y_1^{(1)}, y_1^{(2)})$, and so forth. The extended language of matched strings depends on $G$; we do not write it out explicitly.

**Prefix sum.** If $\mathcal{A}_1 : I \to Z$ and $G : F \cup Z \to F$, then $\oplus_G \mathcal{A}_1 : I \cup F \to F$ implements the semantics

$$\left[\oplus_G \mathcal{A}_1\right](x, y, \epsilon) = [G](y, \left[\mathcal{A}_1\right](x, \epsilon))$$

$$\left[\oplus_G \mathcal{A}_1\right](x, y, w(\sigma, \delta)) = [G]\left(\left[\oplus_G \mathcal{A}_1\right]\((x, y), w\right), \left[\mathcal{A}_1\right](x, w(\sigma, \delta))\right)$$

such that $\text{size}(\oplus_G \mathcal{A}_1) = \text{size}(\mathcal{A}_1) + \text{size}(G) + O(|Z| + |F|)$.

Processing undefined and conflict values explicitly. The semantics of DTs is constrained to behave on $\bot$ and $\top$ in a rigidly defined way. For example, in a transition $(\sigma, X, q', t)$, if any of the variables in $X$ are undefined, the result is undefined. But we may want to process $\bot$ and $\top$ differently: for instance, we want to return $\bot$ instead of $\top$, or return a certain default value instead of $\bot$. To simplify the problem, suppose that we are given $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$, and we want to construct a DT $\mathcal{A}_\bot$, with no input states, the same set of final states, and the following behavior: for any $x \in \mathbb{D}^I$ (not $\mathbb{D}^0$), any $u \in (\Sigma \times \mathbb{D})^*$, and any $f \in F$, if $\mathcal{A}_1(x, u)(f) = \bot$ then $\mathcal{A}_\bot(u)(f) = \bot$, and otherwise, $\mathcal{A}_\bot(u)(f) = \bot$. Here since $I = \emptyset$, the first argument is omitted. We similarly want to define $\mathcal{A}_\top$ which is in $\mathbb{D}$ if $\mathcal{A}_1$ is in $\mathbb{D}$, and $\bot$ otherwise, and $\mathcal{A}_\bot$ which is $\bot$ if $\mathcal{A}_1$ is $\bot$, and $\top$ otherwise. So that $\mathbb{D}$ is not empty, we assume that there is some constant operation in $\mathbb{O}_{\mathbb{P}_0}$, say $d_\star$ (so $d_\star \in \mathbb{D}$). Unlike the previous constructions, we do not copy the states and transitions of $\mathcal{A}_1$ and add more; here, we have to replace every state and transition with a different set of states and transitions. The drawback we will see later is that these operations do not preserve restartability.

The idea of the construction is to replace $Q_1$ with $Q_1 \times \{\bot, \star, \top\}$. For each state $q$ of $Q_1$, we have three states $(q, \bot)$, $(q, \star)$, $(q, \top)$, of which exactly one will be defined and the other two will be $\bot$. Which state is defined should correspond to whether $q$ was undefined, defined, or conflict, respectively. (This is adapted from the classic trick of dealing with negation by replacing all values with pairs of either (true, false) or (false, true).) However, in order for this to work without blowup our DT needs to be acyclic, and all transitions need to have only two source variables, and all states in $Q'$ need to appear as a target variable in at most two transitions. Therefore we have three phases: first, convert second, decompose transitions; third, encode states with triples.

For the first stage, we recall the naive evaluation approach in §2.3. The idea was that we could iterate the fixed-point computation at most $2n$ times, where $n$ is the number of states of the DT, to reach the fixed point. To make the transducer acyclic, then, we just need $2n$ copies of the states. We
We now want to capture the idea of restartability — that many threads may be kept simultaneously active at once, and we will do so by keeping a separate configuration (thread) of $g$ from that point forward. For example, suppose that $\Phi$ is irrational, and consider the following function:

$$f(x) = \begin{cases} 
1 & \text{if } x \leq \Phi \\
0 & \text{otherwise}
\end{cases}$$

Given an input $w$, we can implement the desired semantics of $\text{split}(f, g, op)$ by keeping a separate configuration (thread) of $g$ from that point forward. For example, suppose that $\Phi$ is irrational, and consider the following function:

$$f(x) = \begin{cases} 
1 & \text{if } x \leq \Phi \\
0 & \text{otherwise}
\end{cases}$$

For the second stage, we do not directly preserve the semantics of the DT, but only whether each state is undefined, defined, or conflict on every input. First, for every transition $(σ, X, q', t)$, suppose that $X = \{x_1, x_2, \ldots, x_k\}$. Then we create $k$ intermediate states $q_1, q_2, q_3, \ldots, q_k$, where $q_k = q$ and $q_1$ represents the value of a transition with source variables $\{x_1, \ldots, x_k\}$. Specifically, we replace the transition itself with $k$ intermediate transitions $e_1, e_2, e_3, \ldots, e_k$, where $e_1 = (σ, x_1, q'_1, x_1)$, and for $i ≥ 2$, $e_i = (σ, x_i, q'_{i-1}, q'_i, x_i')$. Then, observe that whether $q_k = q$ is undefined, defined, or conflict is preserved by this transformation. If $X = \emptyset$, we can just replace $(σ, q', t)$ with $(σ, q', d_*)$, where $d_*$ is a constant. Now each transition has at most two source variables. Next, consider a state $q'$ which is the target variable for $k$ transitions. We similarly replace $q'$ with a sequence of states $q_1, q_2, \ldots, q_k$, each of which is the target of only two transitions ($q_{k-1}$ and one of the $k$ original transitions). Overall, the total size of the transducer for this stage multiples by a constant.

Now we are finally ready to look at the third stage. Assume we have $A_1 = (Q_1, Σ, Δ_1, I_1, F_1)$ which is acyclic, and in which every state is the target of at most two transitions, each with at most two source variables. We define three DTS, $[A_1] = \bot, [A_1] ∈ D$, and $[A_1] = \top$. All of them have $Q = Q_1 \times \{\bot, *, \top\}, I = \emptyset$, and the same set of transitions $Δ$, but the set of final states for the three constructions is $F_1 \times \{\bot\}, F_1 \times \{*, \top\}$, and $F_1 \times \{\top\}$, respectively. Now we want to encode the transitions $Δ$. For every state $q'$, we know that there are at most two transitions targetting it with at most two source variables each; so we can build a table of the value of $q'$ given the values of the (at most) four source variables, where “value” means one of $\bot, *, \top$. This is a table with $3^4$ entries. For each entry, we make an appropriate transition: for instance, if $q'$ is $*$ on source variables $x_1 = \bot, x_2 = \top$, and $x_3 = \top$, then we make a transition $(σ, \{(x_1, \bot), (x_2, \top), (x_3, \top)\}, (q', *, d_*))$. This results in a constant number of transitions for each original transition (81, but a more careful analysis gives 9). We also need to initialize the states correctly. In addition to converting the initial transitions, we add initial transitions $(1, \emptyset, (q', *), d_*)$ for each $q ∈ I$, so that each initial state is initially designated $*$ (a value in $D$).

Support. Let $d_* ∈ D$. If $A_1 : I → F$, then $[A_1] = \bot : \emptyset → F, [A_1] ∈ D : \emptyset → F$, and $[A_1] = \top : \emptyset → F$. These constructions implement the following semantics. For all $f ∈ F$:

$[[A_1] = \bot](w)(f) = d_*$ if $[[A_1](x, w)(f)] = \bot ∀x ∈ D$; $\bot$ otherwise

$[[A_1] ∈ D](w)(f) = d_*$ if $[[A_1](x, w)(f)] ∈ D ∀x ∈ D$; $\bot$ otherwise

$[[A_1] = \top](w)(f) = d_*$ if $[[A_1](x, w)(f)] = \top ∀x ∈ D$; $\bot$ otherwise

such that $\text{size}([A_1] = \bot) = O(\text{size}(A_1)^2)$ and likewise for the other two. Alternatively, if $A_1$ is acyclic, the size will only be $O(\text{size}(A_1))$.

3.2 Unambiguous parsing and restartability

We now want to capture the idea of restartability — that many threads may be kept simultaneously active at once — with a formal definition. Recall the example in the introduction of $\text{split}(f, g, op)$. During the execution of $f$ on input $w$, whenever the current prefix $u$ of $w$ where its output is not $\bot$, we can implement the desired semantics of $\text{split}(f, g, op)$ by keeping a separate configuration (thread) of $g$ from that point forward. For example, suppose that

\( w = (a, d_1) (b, d_2) (a, d_3) (a, d_4) \), and that the output of \( f \) is defined after receiving each \( a \)-item, and undefined otherwise. Then \( f \) is defined on input \((a, d_1)\), on \((a, d_1)(b, d_2)\), and on input \((a, d_1)b, d_2) (a, d_3) (a, d_4)\). Corresponding to these three inputs, we have three configurations of \( g \): \( c_1 \) on input \((b, d_2) (a, d_3) (a, d_4)\), \( c_2 \) on input \((a, d_4)\), and \( c_3 \) on input \( d \). Suppose that each configuration \( c_i \) includes an output state with the value of \( y_i = op(f(u), g(w)) \). The value of split \((f, g, op)\) can then be computed as the union of the outputs from all these threads: \( \text{split}(f, g, op)(w) = y_1 \cup y_2 \cup y_3 \).

We apply the union here because we expect the split \( w = u \cdot v \), where \( u \in \overline{I}(f) \) and \( v \in \overline{I}(g) \), to be unique. Thus all but at most one of \( y_i \) will be \( \perp \), and the union gives us the unique answer (if any).

A DT will be called restartable if a single configuration \( c \) can simulate the behavior of these several configurations \( c_1, c_2, \) and \( c_3 \). This is a relation between configurations of \( g \) and an arbitrarily large sequence of configurations of \( g \) (technically, a multiset would be enough instead of a sequence).

The relation \( c \sim [c_1, c_2, c_3] \) is intended to capture that \( c \) is observationally indistinguishable from the sequence of configurations \( c_1, c_2, c_3 \). For starters, we require that the output is the same: if \( y \) is the output of \( c \), then \( y \sim y_1 \cup y_2 \cup y_3 \). But we also require that the simulation is preserved when we update the sequence of configurations of \( g \), by reading in a new input character and/or starting a new thread. The definition allows the simulation to be undefined on configurations or configurations that are never reachable in an actual execution — it need not be true that any sequence of configurations is simulated, only that the starting configurations are simulated, and that the simulation is preserved under any sequence of updates and/or new threads.

With this intuition, the simulation relation on configurations of \( g \) should satisfy the following properties (see the definition below). Property (i) addresses the base case before any input characters are received (i.e. initialization \( i \)). Suppose that on initialization, the machine for \( g \) is started with \( k \geq 0 \) threads, given by input vectors \( x_1, \ldots, x_k \). (In our example, these threads would arise as the output of \( f \) on initialization.) Then the configuration in a single copy of \( g \) on input \( x_1 \cup \cdots \cup x_k \) should simulate the behavior of \( k \) separate copies of \( g \). Property (ii) requires that the simulation then be preserved as input characters are read in. Suppose that \( c \sim [c_1, \ldots, c_k] \), and we now read in a character \( (\sigma, d) \) to \( g \). Simultaneously, we start zero or more new threads represented by the vector \( x \) (e.g., \( x \) is the new output produced by \( f \) on input \( (\sigma, d) \)). Then if we update and re-initialize the initial states of \( c \) with \( x \), that configuration should simulate updating each \( c_i \) separately, and adding one or more new threads represented by \( x \). Finally, property (iii) says that our simulation is sound: for any configuration which simulates a sequence of configurations, the output of the one configuration is equal to the union of the sequence of outputs.

For property (ii) in particular, we need to define what it means to update a configuration \( c \) and simultaneously restart new threads by placing values \( x \) on the initial states \( I' \). (Such an update function is only needed for the simulating configuration, not the sequence of simulated configurations.) For each \( \sigma \in \Sigma \) and for every \( x \in \overline{D} \) we define a generalized evaluation \( \Delta_{\sigma, x} : (Q \rightarrow \overline{D}) \times \overline{D} \rightarrow (Q \rightarrow \overline{D}) \). This represents executing \( \Delta_{\sigma} \) and then starting zero or more new threads, by initializing the new initial states with \( x \). We modify the least fixed point definition of \( c' \) in Equation (1) to include the new initialization on states \( I' \): \( c' \) is the least vector satisfying

\[
c'(q') = x(q') \cup \bigcup_{(\sigma, x, q', t) \in \Delta} t(c'(X)),
\]

where \( x(q) = \perp \) if \( q \notin I \). This resembles the way we already incorporated \( x \) into the definition of \( \Delta_1 \). We restrict the vector \( x \) in each restart to be in the space \( X = \{ \perp \}^I \cup (\perp \cup \{ \top \})^I \), which is closed under \( \cup \). Let \( \overline{I} \) be the vector with every entry equal to \( \perp \).

**Definition 3.1 (Restartability).** Let \( \mathcal{A} = (Q, \Sigma, \Delta, I, F) \) be a DT over signature \((\overline{D}, \text{Op})\); let \( C = \overline{D}^Q \) be the set of configurations of \( \mathcal{A} \), and \([C]\) the set of finite lists of configurations of \( \mathcal{A} \). Let \( X = \)
where

\[ A \]

A

The idea is very simple; every output of

\[ \{\bot\}^I \cup (\mathbb{D} \cup \{\top\})^I \]

be the set of possible initializations for a restarted thread. \( A \) is restartable if there exists a binary relation \( \sim \subseteq C \times [C] \) (called a "simulation") with the following properties:

i. (Base case) For all \( x_1, \ldots, x_k \in X \), \( A_1 \left( \prod_{i=1}^{k} (x_i) \right) \sim [\Delta_1(x_1), \ldots, \Delta_1(x_k)] \). (If \( k = 0 \), we get \( \Delta_1(\bot) \sim [\bot] \), where \( [\bot] \in [C] \) denotes the empty list.)

ii. (Update with restarts) For all \( (\sigma, d) \in (X \times \mathbb{D}) \), for all \( x \in X \), and for all \( c, c_1, c_2, \ldots, c_k, \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_l \), if \( c \sim [c_1, c_2, \ldots, c_k] \) and \( \Delta_1(x) \sim [\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_l] \) then

\[ \Delta_{\sigma, x}(c, d) \sim [\Delta_{\sigma}(c_1, d), \ldots, \Delta_{\sigma}(c_k, d), \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_l]. \]

iii. (Implies same output) If \( c \sim [c_1, c_2, \ldots, c_k] \), and the output vectors for these configurations (extended data values at the final states) are \( y, y_1, y_2, \ldots, y_k \), respectively, then we have

\[ y = y_1 \sqcup y_2 \sqcup \cdots \sqcup y_k. \]

Here is a counterexample to the definition. Consider a DT \( A \) which sums the value of a single initial state and the last \( \alpha \): take \( Q = \{i, f\} \), \( I = \{i\} \), \( F = \{f\} \), and the following transitions on input \( \alpha \): \( i' := i, f' := i + \text{cur} \). We may represent configurations as \( (x, y) \), for the values at \( i, f \), respectively. To see this is not restartable, consider starting \( A \) with a single input \( x_1 \in \mathbb{D} \), then reading in \( (a, d) \) and starting a second input \( x_2 \in \mathbb{D} \). Starting with \( x_1 \) results in the configuration \( (x_1, \bot) \); then reading in \( (a, d) \) and starting with \( x_2 \) results in \( (\top, \top) \). However, if \( A \) were restartable, then by property (ii), we could instead read in \( (a, d) \) and add the second input \( x_2 \) separately: we thus would have \( (\top, \top) \sim [(x_1, x_1 + d), (x_2, \bot)] \). The problem is that this violates (iii): the output of \( A \) is \( \top \), which is not the same as \( (x_1 + d) \sqcup \bot = x_1 + d \).

What is relevant for properties (i), (ii), and (iii) is actually only the configurations, input, and output up to equivalence, i.e., where we replace \( \mathbb{D} \) with a 3-element set \( \{\bot, \star, \top\} \). There are only finitely many configurations up to equivalence. This is why restartability is decidable (see Theorem 3.3).

Concatenation. Suppose we are given two DTs \( A_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1) \) and \( A_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2) \), where \( F_1 \) and \( I_2 \) are the same up to bijection (say, \( \pi : F_1 \rightarrow I_2 \)). Now we want to compute the following parsing operation: on input \( (x, w) \), consider all splits of \( w \) into two strings, \( w = w_1, w_2 \). Apply \( A_1 \) to \( (x, w_1) \) to get a result \( y_1 \), and apply \( A_2 \) to \( (y_1, w_2) \) to get \( y_2 \). Return the union (\( \sqcup \)) over all such splits of \( y_2 \). In particular, assuming there is only one way to split \( w = w_1, w_2 \) such that \( y_2 \) does not end up being undefined, this operation splits the input string uniquely into two parts such that \( A_1 \) matches \( w_1 \) and \( A_2 \) matches \( w_2 \), and then applies \( A_1 \) and \( A_2 \) in sequence.

We implement this by taking \( A = A_1 \cdot A_2 = (Q, \Sigma, \Delta, I, F) \) with \( Q = Q_1 \cup Q_2 \), \( I = I_1 \cup I_2 \), and \( F = F_1 \cap \left\{ f \in F_2 : f_1 \in F_1 \right\} \).

The idea is very simple; every output of \( A_1 \) (i.e., a value produced at a state in \( F_1 \)) should be copied into the corresponding input state of \( A_2 \). This happens on initialization, and on every update. However, the semantics is not so simple, because every time we read in a character, \( A_2 \)'s initial states \( I_2 \) are being re-initialized with new values (the values from \( F_1 \)).

This "re-initialization" is exactly captured by our generalized update function \( \Delta_{\sigma, x} \). Let us represent configurations of \( A \) by \( (c_1, c_2) \), where \( c_1 \) is the component restricted to \( Q_1 \), i.e. the induced configuration of \( A_1 \). Now consider an input \( (x, w) \) to \( A \). We see that for the \( i \)th configuration of \( A \), \( (c_1^{(i)}, c_2^{(i)}) \), \( c_1^{(i)} \) is the same as the \( i \)th configuration of \( A_1 \) on input \( (x, w) \). Moreover, if \( y_i^{(i)} \) is the \( i \)th output of \( A_1 \), this is used to reinitialize \( A_2 \); so we see that \( c_2^{(i)} = \Delta_{\sigma, y_i^{(i)}(c_1^{(i-1)}), d} \) (where this is the update function of \( A_2 \)). The output \( y_i^{(i)} = c_2^{(i)}|_F \) of \( A_2 \) is the output of \( A \).

Assume that \( A_1 \) is output-synchronized: this means that each \( y_i^{(i)} \in X \), i.e., all values are \( \bot \) or all values are in \( \mathbb{D} \cup \{\top\} \). And assume that \( A_2 \) is restartable. Then the simulation relation allows us to,
at every step, replace \( c_2 \) by a list of configurations where each configuration is \( A_2 \) on a different suffix of \( w \). In particular, we recursively replace \( \Delta_{c_2^{(i-1)}}(c_2, d) \) with the list of configurations for \( \Delta_{c_2^{(i-1)}}(d) \) and a single new thread \( \Delta_1(y_1^{(i)}) \). Because \( y_1^{(i)} \in X \), this is guaranteed by property (ii) of restartability. Property (iii) then implies the semantics given in the following summary.

\[
\text{Concatenation. } \text{Let } A_1 : I \to Z \text{ and } A_2 : Z \to F, \text{ such that } A_1 \text{ is output-synchronized and } A_2 \text{ is restartable. Then } A_1 \cdot A_2 : I \to F \text{ implements the semantics}
\]

\[
\begin{array}{c}
\mathbb{A}_2(x, w) = \\
\bigcup_{w = w_1 w_2} [\mathbb{A}_1 \cdot \mathbb{A}_2](x, w_1), w_2).
\end{array}
\]

such that \( \text{size}(A_1 \cdot A_2) = \text{size}(A_1) + \text{size}(A_2) + O(|Z|) \). It matches \( \mathcal{L}(A_1 \cdot A_2) = \mathcal{L}(A_1) \cdot \mathcal{L}(A_2) \).

**Concatenation with data functions.** A special case of concatenation can be described which does not require restartability, and which we use in §4. Suppose we are given \( A_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1) \) and we want to concatenate with a data function \( G_2 : F_1 \Rightarrow F_2 \) on input \( (x, w) \), return \( G_2(A_1(x, w)) \). This can be implemented by converting \( G_2 \) into a DT \( A_2 \) on states \( F_1 \cup F_2 \) (as in the prefix sum construction), and then simply constructing \( A_1 \cdot A_2 \). Even if \( A_2 \) is not restartable, we can see directly that on any input, the output states \( F_2 \) are equal to \( G_2 \) applied to the output of \( A_1 \). Similarly, if \( G_1 : I_1 \Rightarrow I_2 \) and \( A_2 : (Q_2, \Sigma, \Delta_2, I_2, F_2) \), then we may convert \( G_1 \) into a DT \( A_1 \) on states \( I_1 \cup I_2 \). Then the construction \( A_1 \cdot A_2 \), on any input \( (x, w) \), returns \( A_2(G_1(x), w) \). We overload the concatenation notation and write these constructions as \( A_1 \cdot G_2 \) and \( G_1 \cdot A_2 \). For these constructions, as with prefix sum, we do not write out the extended language of matched strings explicitly.

**Concatenation with data functions.** If \( A_1 : I \to Z \) and \( G_2 : Z \Rightarrow F \), then \( A_1 \cdot G_2 : I \to F \) implements the semantics

\[
\begin{array}{c}
\mathbb{A}_2(x, w) = \\
G_2([\mathbb{A}_1](x, w)),
\end{array}
\]

such that \( \text{size}(A_1 \cdot G_2) = \text{size}(A_1) + \text{size}(G_2) + O(|Z|) \). Likewise, if \( G_1 : I \Rightarrow Z \) and \( A_2 : Z \Rightarrow F \), then \( G_1 \cdot A_2 : I \to F \) implements the semantics

\[
\begin{array}{c}
G_1 \cdot \mathbb{A}_2(x, w) = \\
G_2([\mathbb{A}_2](x, w)),
\end{array}
\]

such that \( \text{size}(G_1 \cdot A_2) = \text{size}(G_1) + \text{size}(A_2) + O(|Z|) \).

**Iteration.** Now suppose we are given \( A_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1) \), where \( I_1 \) and \( F_1 \) are the same up to some bijection. On input \( (x, w) \), we want to split \( w \) into \( w_1 w_2 w_3 \ldots \), then apply \( A_1(x, w_1) \) to get \( y_1, A_1(y_1, w_2) \) to get \( y_2 \), and so on. Then, the answer is the union over all possible ways to write \( w = w_1 w_2 \ldots w_k \) of \( y_k \). Let \( I \) be a set the same size as \( I_1, F_1 \) with bijections \( \pi : I \to I_1, \rho : F \to F_1 \). Then we implement this by taking \( A_1 = (A_1)^\ast = (Q, \Sigma, \Delta, I, F) \) with \( Q = Q_1 \cup I \) and

\[
\Delta = \Delta_1 \cup \{ (\epsilon, \{i' \}, \pi(i'), i') : i \in I \} \cup \{ (\epsilon, \{\rho(i')\}, i', \rho(i') : i \in I \}.
\]

The idea is again very simple; we have a set of states \( I \) that is both initial and final; we always copy the values of these states into the input of \( A_1 \) and copy the output states of \( A_1 \) back into \( I \). But the semantics is again more complicated. Here (unlike all other constructions), we do not necessarily preserve acyclicity. When we copy \( F_2 \) into \( I \) and back into \( I_2 \), this may then propagate back into \( F_2 \) again. Essentially, if \( A_1 \) produces output on the empty data word, then \( (A_1)^\ast \) will always be \( T \), as this will create a cycle with least fixed point \( T \).
We assume that $\mathcal{A}_1$ is both output-synchronized and restartable. We can write configurations of $\mathcal{A}$ as $(c, y)$, where $c$ is a configuration of $\mathcal{A}_1$. On an input word $w = (\sigma_1, d_1), \ldots, (\sigma_k, d_k)$, let the sequence of configurations be $(c_0, y_0), (c_1, y_1), \ldots, (c_k, y_k)$, so the output of $\mathcal{A}$ is $y_k$. Then the least-fixed-point semantics (Equation 1) implies that, for $i = 1, \ldots, k$, $y_i$ is the least vector satisfying $y_i = (\Delta_{\sigma_i, y_i}(c_{i-1}, d_i))|_{F_i}$. Similarly, for $i = 0$, $y_0$ is the least vector satisfying $y_0 = (\Delta_1(y_0))|_{F_1}$. Now we want to show by induction that $c_i$ simulates the list, over all possible splits of $w = w_1w_2 \ldots w_k$, of the configuration of $\mathcal{A}_1$ obtained by sequentially applying $\mathcal{A}_1$ $k$ times. The proof of the inductive step is to take the property $y_i = (\Delta_{\sigma_i, y_i}(c_{i-1}, d_i))|_{F_i}$ and decompose the configuration $\Delta_{\sigma_i, y_i}(c_{i-1}, d_i)$ using the simulation relation, and see that it simulates the list of all splits $w = w_1 \ldots w_k$ where $w_k$ has size at least 1, plus the additional initialized thread $\Delta_1(y_i)$. Overall, we obtain the following summary.

**Iteration.** Let $\mathcal{A} : I \rightarrow I$ be output-synchronized and restartable. Then $\mathcal{A}^* : I \rightarrow I$ satisfies

$$[\mathcal{A}^*](x,w) = \bigcup_{w=w_1w_2 \ldots w_k} [\mathcal{A}](\ldots [\mathcal{A}][\mathcal{A}(x,w_1),w_2] \ldots ,w_k),
$$

s.t. $\text{size}(\mathcal{A}^*) = \text{size}(\mathcal{A}) + O(|I|)$. It matches $\overline{I}(\mathcal{A}^*) = \overline{I}(\mathcal{A})^*$.

**Properties of restartability.**

**Theorem 3.2.** If $\mathcal{A}_1$ and $\mathcal{A}_2$ are restartable, then so are $\mathcal{A}_1 \parallel \mathcal{A}_2$ and $\mathcal{A}_1 \uplus \mathcal{A}_2$. If $\mathcal{A}_1$ is additionally output-synchronized, then $\mathcal{A}_1 \parallel \mathcal{A}_2$ and $\mathcal{A}_1^*$ are restartable. If $\mathcal{A}_1$ is restartable and output-synchronized and additionally $\overline{I}(\mathcal{A}_1) = \Sigma^*$, and if $G$ is a data function where each output value is given by a single term value given by the input values, then $\oplus_G \mathcal{A}_1$ is restartable.

**Proof.** For $\mathcal{A}_1 \parallel \mathcal{A}_2$ and $\mathcal{A}_1 \uplus \mathcal{A}_2$, we represent configurations of the machine has pairs $(c_1, c_2)$, and we define $(c_1, c_2) \sim [(c_{1,1}, c_{2,1}), \ldots, (c_{1,k}, c_{2,k})]$ if both $c_1 \sim [c_{1,1}, \ldots , c_{1,k}]$ and $c_2 \sim [c_{2,1}, \ldots , c_{2,k}]$.

For prefix sum, the restartability holds for somewhat trivial reasons: if we restart with only $\bot$, the output is $\bot$: if we restart with only one non-$\bot$ thread, the output is the prefix-sum, and if we restart with two or more non-$\bot$ threads, the output is $\top$ everywhere.

For concatenation, we have configurations that are pairs $(c_1, c_2)$ of a configuration in $\mathcal{A}_1$ and one in $\mathcal{A}_2$. We define $(c_1, c_2) \sim [(c_{1,1}, c_{2,1}), \ldots, (c_{1,k}, c_{2,k})]$ if $c_1 \sim [c_{1,1}, \ldots , c_{1,k}]$ and there exists sequences $l_{2,1}, l_{2,2}, \ldots , l_{2,k}$, such that $c_{2,i}$ simulates $l_{2,i}$ and $c_2$ simulates the entire sequence of sequences, $l_{2,1} \circ l_{2,2} \circ \cdots \circ l_{2,k}$. The idea is that a configuration in $\mathcal{A}$ simulates a list of configurations where each configuration consists of only a single thread in $\mathcal{A}_1$, but may have many threads in $\mathcal{A}_2$ (since one thread in $\mathcal{A}_1$ may cause $\mathcal{A}_2$ to be restarted several times). However, we still need that there exists some further simulation of the configuration in $\mathcal{A}_2$ into a set of individual threads, such that the overall configuration of $\mathcal{A}_2$ in $\mathcal{A}$ simulates all of these individual threads.

For iteration, we have to do this more recursively. We say that $c \sim [c_1, \ldots , c_k]$ if either $c \sim [c_{i,1}, \ldots , c_{i,k}]$ in $\mathcal{A}_1$, or recursively, if $c \sim [c_{i,1}, \ldots , c_{i,k}]$ in $\mathcal{A}$ and $c_i \sim [c_{i,1}, \ldots , c_{i,j}]$ in $\mathcal{A}$ such that in $\mathcal{A}_1$, $c$ simulates $[c_{i,j}]_{i,j}$.

**Theorem 3.3.** On input a DT $\mathcal{A}$, checking if $\mathcal{A}$ is restartable is PSPACE-complete.

**Proof.** Recall that in the proof of Theorem 2.2, we constructed a DT $\mathcal{P}$ over the unit signature $(U, UOp)$, where $U = \{\star\}$ is a set with just one element, such that the states of $\mathcal{P}$ and $\mathcal{A}$ always agree on whether they are undefined, defined, or conflict. Our main lemma towards obtaining the
proof of this theorem is that restartability of \( \mathcal{A} \) is equivalent to restartability of the machine \( \mathcal{P} \) (which may seem much weaker).

**Lemma:** Let \( \mathcal{A} \) be a DT over signature \((\mathcal{D}, \text{Op})\), and let \( \mathcal{P} \) be the projection over signature \((\mathcal{U}, \text{UOp})\). Then \( \mathcal{A} \) is restartable if \( \mathcal{P} \) is restartable. **Proof:** We use \( \pi \) to denote configurations of \( \mathcal{A} \) and \( \pi_t \) to denote configurations of \( \mathcal{P} \). The forward direction is immediate: define the relation \( p \sim [p_1, p_2, \ldots, p_k] \) if there exists \( c \sim [c_1, c_2, \ldots, c_k] \) such that \( p = \Pi(c) \) and \( p_t = \Pi(c_t) \). Facts (i), (ii), and (iii) are preserved by the projection to the unit signature \((\mathcal{U}, \text{UOp})\).

The backward direction is nontrivial. We need to define the simulation relation between configurations and lists of configurations. We define the reachable relation \( R \subseteq C \times [C] \) to be the minimal relation that is implied by properties (i) and (ii). In other words, it is the set of pairs \((c, [c_1, c_2, \ldots, c_k])\) reachable from initialization followed by some sequence of elements of \( \Sigma \times \mathcal{D} \) with restarts. We will show that \( R \) is a simulation by showing that (iii) holds of all reachable pairs.

The key observation — which holds even if \( \mathcal{A} \) is not restartable — is that for any reachable pair \((c, [c_1, c_2, \ldots, c_k])\), \( c \geq c_i \) for all \( i \), where \( \geq \) is the coordinate-wise partial ordering on sequences of \( \mathcal{D} \) elements (the partial ordering on \( \mathcal{D} \) is described in §2.1). Intuitively, since the operations on \( \mathcal{D} \) are monotone, starting multiple threads only increases the resulting configuration over any individual one of those threads.

Now we claim that \( R \) satisfies (iii). Let \((c, [c_1, c_2, \ldots, c_k])\) be reachable. Fix \( f \in F \). Since \( \mathcal{P} \) is restartable, we know that \( c(f) \) and \( c_1(f) \sqcup \cdots \sqcup c_k(f) \) are either both undefined, both defined, or both conflict. Thus the only way they can be unequal (violating (iii)) is if they are both in \( \mathcal{D} \), and distinct. If they are both in \( \mathcal{D} \), then \( c_i(f) = \perp \) for all \( i \) except one, say \( c_i(f) = d' \). But from the key observation above, \( c(f) \succeq c_i(f) \), and since \( c, c_i \in \mathcal{D} \), they are equal. This completes the proof of the Lemma.

Now we give a coNPSPACE algorithm for the restartability problem. Given a DT \( \mathcal{A} \), it is enough to construct \( \mathcal{P} \) and check its restartability, by the above lemma. So we need to check if there exists a reachable pair \((p, [p_1, \ldots, p_k])\), where \( p \) and \( p_t \) are configurations of \( \mathcal{P} \), such that \( F(p) = F(p_1) \sqcup F(p_2) \sqcup \cdots \sqcup F(p_k) \).

But choose \( k \) to be minimal; then we do not need to keep track of \( p_1, \ldots, p_{k-1} \), but can instead collapse these into a single configuration \( p' \). Specifically, before the \( k \)-th restart, suppose we are at \((p', [p'_1, p'_2, \ldots, p'_{k-1}])\); then rather than keeping \( p'_1 \) through \( p'_{k-1} \), we know the output will always be the same as taking \( p' \), so we keep track only of \( p' \). Using this trick, the space required to store \((p, [p_1, \ldots, p_k])\) constant: three configurations of \( \mathcal{P} \). Overall, we guess a sequence of moves to get to \((p', [p'_1, \ldots, p'_{k-1}])\), then guess a sequence of moves to get to \( p \) from there, and guess a place to stop and try checking if \( p(f) = p_1(f) \sqcup p_2(f) \sqcup \cdots \sqcup p_k(f) \) for all \( f \in F \). The total space is bounded and some thread accepts if and only if there is a counterexample, meaning the machine is not restartable.

PSPACE-hardness can be shown by a reduction from the problem of universality for NFAs. We exploit that if NFAs \( N_1 \) and \( N_2 \) are translated to DTs which always output \( \perp \) or \( \top \), and \( G \) is a DT with two inputs that applies a binary operation to get one output, the DT construction \( N_1 \parallel N_2 \cdot G \) is restartable iff there do not exist strings \( u, v \) such that \( u \in L(N_1), u \notin L(N_2), uv \notin L(N_1), uv \in L(N_2) \), or vice versa.

**Converting to restartable.** We finally mention the complexity of converting a DT to a restartable one. It is shown in Theorem 5.1 that a DT of size \( m \) can be converted to a deterministic CRA of size \( \exp(m) \); and that a deterministic CRA of size \( m \) can be converted into a restartable DT of size \( O(m) \). This gives a procedure to convert DT to restartable DT, with exponential blowup.
4 PROPOSED MONITORING LANGUAGE: QRE-PAST

In this section we present the QRE-Past query language for quantitative runtime monitoring (Quantitative Regular Expressions with Past-time temporal operators). Each query compiles to a streaming algorithm, given as a DT, whose evaluation has precise complexity guarantees in the size of the query. Specifically, the complexity is a quadratic number of registers and quadratic number of operations to process each element, in the size of the query, independent of the input stream. Our language employs several constructs from the StreamQRE language [Mamouras et al. 2017]. To this core set of combinators we add the prefix-sum operation, fill and fill-with operations, and also past-time temporal logic operators which allow querying temporal safety properties: for example, “is the average of the last five measurements always more than two standard deviations above the average over the last two days?” We have picked constructs which we believe to be intuitive to program and useful in the application domains we have studied, but we do not intend them to be exhaustive; there are many other combinators which could be easily be defined, added to the language, and implemented using the back-end support provided by the constructions of §3.

By compiling to the DT machine model, we show that the compiled code has the same precise complexity guarantee of the code produced by the StreamQRE engine of [Mamouras et al. 2017], including the additional temporal operators. Because compiled StreamQRE code was shown to perform significantly better than existing competitors when deployed on a single machine, this is good evidence that QRE-Past would see similar success with more flexible language constructs.

4.1 Syntax of QRE-Past

Expressions in the language are divided into three types: quantitative queries of two types, either base-level (α) or top-level (β), and temporal queries (φ). Base-level quantitative queries specify functions from data words to quantities (extended data values D), and are compiled to restartable DTs with a single initial state and single final state, of quadratic size. These queries are based on StreamQRE and the original Quantitative Regular Expressions of [Alur et al. 2016]. Top-level quantitative queries also specify functions from data words to quantities, but the compiled DT may not be restartable. Temporal queries specify functions from data words to Booleans, may be constructed from quantitative queries, and are compiled to DTs which output Booleans. Temporal queries are based on the operators of past-time temporal logic [Manna and Pnueli 2012] and informed by successful existing work on monitoring of safety properties [Havelund and Roşu 2004], which adapts to our setting via constructions on DTs. Unlike in model checking, past-time temporal operators are preferred over future-time temporal operators in monitoring, because we must produce an answer on each finite trace.

We model Booleans as elements in D. Thus, we assume that 0, 1 ∈ D, and that ≤, ≥, = ∈ Op₂: these are comparison operations on data values returning 0 or 1. We also assume that we have Boolean operators ¬ ∈ Op₁ and ∧, ∨, →, ↔ ∈ Op₂. Their behavior on non-Boolean inputs is unimportant.

Each query has an associated regular rate L(α), defined by a regular expression on Σ defined recursively with the query. The rate expresses the set of strings on which the compiled DT is defined or conflict. For temporal queries φ, the rate is Σ*'. We also may refer to the language L(α) ⊆ L(α), which is the set of strings on which the compiled DT is defined. There are a few typing restrictions, mainly constraints on the extended languages of the queries. Because each extended language is given by a regular expression, the typing restrictions are type-checkable in polynomial time. The typing restrictions arise in order to guarantee restartability so that the constructions of §3 apply.
We describe each construction’s semantics, and how it is directly implemented as a DT. For technical reasons, for each quantitative query (not for temporal queries) α or β we produce two DTs. The first is $A_\alpha : X \rightarrow Y$, where $|X| = |Y| = 1$. The semantics will be such that $A_\alpha(x, w)$ is the value of query $\alpha$ on input $w$, if $x$ is defined. So $x$ is not really used, except that it allows the machine to be restartable (at least one initial state is needed for restarts). The second is $I_\alpha : X \rightarrow Y$, which has the following identity semantics: $I_\alpha(x, w) = x$ if $A_\alpha(x, w) \in D$, $\top$ if $A_\alpha(x, w) = \top$, and $\bot$ if $A_\alpha(x, w) = \bot$. The reason we need this second machine $I_\alpha$ is to save values for using later. For example, to implement $\text{split}(f, g, op)$ we concatenate the machine for $f$ with a machine which both saves the output of $f$ and starts $g$; then when $g$ is finished we combine the saved output of $f$ with the output of $g$ via $op$. We will guarantee in the translation that $I_\alpha$ has size only linear in the query, but $A_\alpha$ has worst-case quadratic size.

Atomic expressions: atom, eps. The atomic expressions are the building blocks of all queries. The query $\text{atom}(\sigma, t)$ matches a data word containing a single character ($\sigma, d$), and returns $t$ evaluated with $\text{cur} = d$. Similarly, the query $\text{eps}(t)$ matches the empty data word and returns the evaluation of $t$.

We define $A_{\text{atom}(\sigma, t)}$ to have two states: $Q = \{q_i, q_f\}$, with $I = \{q_i\}$ and $F = \{q_f\}$. We define only one transition: $(\sigma, \{q_i\}, q_f, t)$. The result is that whenever we read a $\sigma$, if $q_i$ is defined, then the output $q_f$ is set to $t$. If we read in any other character, or more than one character, the output will be undefined again. Moreover, this machine is restartable: define $c \sim [c_1, \ldots, c_k]$ whenever $c = \bigcup_{i=1}^k c_i$. Then we see that this is a bisimulation relation between configurations of the machine and a list of copies of configurations of the machine, such that initializing $q_i$ with additional values in $c$ is equivalent ($\sim$) to adding new threads with those additional values to the list, and the output of $c$ is the union of the outputs of $c_i$.

Fig. 6. Summary of the QRE-Past language: syntax for quantitative queries $\alpha$, $\beta$ and temporal queries $\varphi$. The second column gives the rate of the query as a regular expression.

### 4.2 Semantics and compilation algorithm

We describe each construction’s semantics, and how it is directly implemented as a DT. For technical reasons, for each quantitative query *not* for temporal queries $\alpha$ or $\beta$ we produce two DTs. The first is $A_\alpha : X \rightarrow Y$, where $|X| = |Y| = 1$. The semantics will be such that $A_\alpha(x, w)$ is the value of query $\alpha$ on input $w$, if $x$ is defined. So $x$ is not really used, except that it allows the machine to be restartable (at least one initial state is needed for restarts). The second is $I_\alpha : X \rightarrow Y$, which has the following identity semantics: $I_\alpha(x, w) = x$ if $A_\alpha(x, w) \in D$, $\top$ if $A_\alpha(x, w) = \top$, and $\bot$ if $A_\alpha(x, w) = \bot$. The reason we need this second machine $I_\alpha$ is to save values for using later. For example, to implement $\text{split}(f, g, op)$ we concatenate the machine for $f$ with a machine which both saves the output of $f$ and starts $g$; then when $g$ is finished we combine the saved output of $f$ with the output of $g$ via $op$. We will guarantee in the translation that $I_\alpha$ has size only linear in the query, but $A_\alpha$ has worst-case quadratic size.

Atomic expressions: atom, eps. The atomic expressions are the building blocks of all queries. The query $\text{atom}(\sigma, t)$ matches a data word containing a single character ($\sigma, d$), and returns $t$ evaluated with $\text{cur} = d$. Similarly, the query $\text{eps}(t)$ matches the empty data word and returns the evaluation of $t$.

We define $A_{\text{atom}(\sigma, t)}$ to have two states: $Q = \{q_i, q_f\}$, with $I = \{q_i\}$ and $F = \{q_f\}$. We define only one transition: $(\sigma, \{q_i\}, q_f, t)$. The result is that whenever we read a $\sigma$, if $q_i$ is defined, then the output $q_f$ is set to $t$. If we read in any other character, or more than one character, the output will be undefined again. Moreover, this machine is restartable: define $c \sim [c_1, \ldots, c_k]$ whenever $c = \bigcup_{i=1}^k c_i$. Then we see that this is a bisimulation relation between configurations of the machine and a list of copies of configurations of the machine, such that initializing $q_i$ with additional values in $c$ is equivalent ($\sim$) to adding new threads with those additional values to the list, and the output of $c$ is the union of the outputs of $c_i$. 

We define \( \mathcal{A}_{\epsilon\text{psl}(t)} \) with the same \( Q,I \), and \( F \), but a single epsilon-transition \((\epsilon,\{q_i\},q'_f,t)\) (note the \( q'_f \) instead of \( q_i \), and recall that \( \epsilon \) is an abbreviation for a copy of the transition for every \( \sigma \in \Sigma \cup \{\epsilon\} \). In any reachable configuration \( c \) of the machine, the value of \( q_f \) is \( t(c(q_i)) \), even with restarts. So we define \( c \sim [c_1,\ldots,c_k] \) if \( c \) and all \( c_i \) satisfy this, and \( c = \bigcup_{i=1}^k c_i \).

The definition of \( I_{\text{atom}(\sigma,t)} \) is the same as \( \mathcal{A}_{\text{atom}(\sigma,t)} \) except that the term \( t \) in the transition is replaced by \( q_i \); and likewise the definition of \( I_{\epsilon\text{psl}(t)} \) is the same as \( \mathcal{A}_{\epsilon\text{psl}(t)} \) except the term \( t \) in the transition is replaced by \( q_i \). The identical argument shows they are restartable.

Regular operators: or, split, iter. These regular operators are like union, concatenation, and iteration, except that if the parsing of the string (data word) is not unique, the result will be \( \top \).

The union operation \( \oplus(\alpha_1,\alpha_2) \) should match any data word that matches either \( \alpha_1 \) or \( \alpha_2 \); if it matches only one, its value is that query, but if it matches both, its value is \( \top \). In particular, conflict values “propagate upwards” because even if only one of \( \alpha_1,\alpha_2 \) matches, if the value is \( \top \) then the result is \( \top \). This is exactly the semantics of the DT construction \( \mathcal{A}_{\alpha_1} \cup \mathcal{A}_{\alpha_2} \). It is restartable because \( \mathcal{A}_{\alpha_1} \) and \( \mathcal{A}_{\alpha_2} \) are restartable, by Theorem 3.2. Similarly, we can take \( I_{\oplus(\alpha_1,\alpha_2)} = I_{\alpha_1} \cup I_{\alpha_2} \). Both of these constructions add only a constant to the size.

The operation split \((\alpha_1,\alpha_2,\text{op})\) splits a data word \( w \) into two parts, \( w_1 \cdot w_2 \), such that \( w_1 \) matches \( \alpha_1 \) and \( w_2 \) matches \( \alpha_2 \). If there are multiple splits, the result is \( \top \); otherwise, the result is \( \text{op}(\alpha_1(w_1),\alpha_2(w_2)) \). Here, we have to do some work to save the value of \( \alpha_1(w_1) \) in the DT construction. We implement split as \( \mathcal{A}_{\text{split}(\alpha_1,\alpha_2,\text{op})} = (\mathcal{A}_{\alpha_1} \cdot (I_{\alpha_2} \parallel \mathcal{A}_{\alpha_2})) \cdot \text{G}_{\text{op}} \), where \( \text{G}_{\text{op}} \) is a data function with two inputs \( y_1, y_2 \) which returns one output \( \text{op}(y_1,y_2) \), where \( y_1 \) is the final state of \( I_{\alpha_2} \), and \( y_2 \) is the final state of \( \mathcal{A}_{\alpha_2} \). Let’s parse what this is saying. We split the string \( w \) into two parts \( w_1 \cdot w_2 \) such that \( w_i \in I(\alpha_i) \), and apply \( \alpha_i \) to the first part; for the second part, we construct a transducer which takes the output of \( \alpha_i \) as input and produces both that value as \( y_1 \), as well as the new output of \( \alpha_2 \) as \( y_2 \). Then both of these are passed to \( \text{G}_{\text{op}} \) which returns \( \text{op}(y_1,y_2) \). To define \( I_{\text{split}(\alpha_1,\alpha_2,\text{op})} \) is much easier: we take \( I_{\alpha_1} \cdot I_{\alpha_2} \).

The operation iter \((\alpha,\text{init},\text{op})\) splits \( w \) into \( w_1 \cdots w_k \) such that \( w_i \in I(\alpha_i) \) and then folds \( \text{op} \) over the list of outputs of \( \alpha_i \), starting from \( I \), to get a result: for instance if \( k = 3 \), the result is \( \text{op}(\text{op}(\text{init},\alpha_1(w_1)),\alpha_1(w_2)),\alpha_1(w_3)) \). If the parsing is not unique, the result is \( \top \). We have \( \mathcal{A}_{\text{iter}(\alpha,\text{init},\text{op})} = \mathcal{A}_{\epsilon\text{psl(init)}} \cdot ((I_{\alpha_1} \parallel \mathcal{A}_{\alpha_1}) \cdot \text{G}_{\text{op}})^* \).

We need to argue that these constructions preserve restartability. For concatenation, we need that the \( \parallel \) is output-consistent: we need that \( \mathcal{A}_{\alpha_1} \) and \( I_{\alpha_2} \) have the same rate. This is true by construction: \( I \) has the same language has \( \mathcal{A} \) and only differs in that it is the identity function from input to output. Given this, the three DTs concatenated are all parallel-consistent. To show the result is restartable, it remains to show that \( \text{G}_{\text{op}} \) is converted to a DT which is restartable. This can be done by defining the simulation relation: To define \( c \sim [c_1,\ldots,c_k] \) we first require that for every \( i \), either \( c_i(x) = \bot \) for all \( x \in X \) or \( c_i(x) \in \Sigma \cup \{\top\} \) for all \( x \in X \), and the same for \( c \). Then, we require that \( c = \bigcup_{i=1}^k c_i \). Then property (iii) holds immediately, and (i) and (ii) can be shown as well. The size of the concatenation construction is bounded by a quadratic polynomial because we have added additional size equal to the size of \( I_{\alpha_2} \), which is bounded by a linear polynomial. The construction \( I_{\alpha_1} \cdot I_{\alpha_2} \) preserves a linear bound and is restartable because \( I_{\alpha_1} \) and \( I_{\alpha_2} \) are restartable.

Now we explain the iteration implementation. The DT \((I_{\alpha_1} \parallel \mathcal{A}_{\alpha_1}) \cdot \text{G}_{\text{op}} \) takes an input, both saves it and performs a new computation \( \mathcal{A}_{\alpha_1} \), and then produces \( \text{op} \) of the old value and the new value. When this is iterated, we get the desired fold operation; finally, \( \mathcal{A}_{\epsilon\text{psl(init)}} \) provides this with the needed initial value \( \text{init} \).

For iteration, the machine is restartable by a similar argument. First, the \( \parallel \) is only applied to two machines with the same extended language. Then, \( \text{G}_{\text{op}} \) is restartable as we saw for concatenation.
Therefore, the overall machine is restartable. As with \texttt{split}, the size of our construction includes the size of \(A_{a_1}\) but adds a linear size due to inclusion of \(I_{a_1}\), so we preserve a quadratic bound on size. For \(I_{\text{iter}(a_1, \text{init}, \text{op})}\) we can simply take \((I_{a_1})^*\).

\textbf{Parallel combination: combine.} This is the first operation in our language which requires a typing restriction. For 
\(\text{combine}(a_1, \ldots, a_k, \text{op})\), the computation is simple: apply every \(a_i\) to the input stream to get a result, then \texttt{combine} all these results via operation \texttt{op}. The implementation as a DT is 
\[A_{\text{combine}(a_1, \ldots, a_k, \text{op})} := (A_{a_1} \parallel \cdots \parallel A_{a_k}) \cdot G_{\text{op}},\]
where \(G_{\text{op}}\) applies \texttt{op} to the \(k\) output states of the \(\parallel\). For \(I_{\text{combine}(a_1, \ldots, a_k, \text{op})}\), we do the same thing but replace \texttt{op} by the term \(y_1\) (i.e. we use \(G_{y_1} : \{y_1, \ldots, y_k\} \Rightarrow \{y\}\) where \(y\) is the final output variable). The construction for \texttt{combine} is well-defined even if the typing restriction is not satisfied, but does not preserve restartability in that case. We use the non-restartable version in some other constructions.

Let us argue that this construction is restartable. First, the left part of the concatenation is output-consistent by the typing restriction. Second, the \(G_{\text{op}}\) is converted to a restartable data transducer. The size of both \(A_{\text{combine}(a_1, \ldots, a_k, \text{op})}\) and \(I_{\text{combine}(a_1, \ldots, a_k, \text{op})}\) linear in the sizes of the constituent DTs, so these constructions preserve the quadratic and linear bound on size, respectively.

\textbf{Prefix sum: prefix-sum.} The prefix sum \(\text{prefix-sum}(a_1, \text{init}, \text{op})\) is defined only if \(a_1\) is defined (not conflict) on all input. Its value should be \(\text{op}(\text{init}, \alpha_1(\epsilon))\) on the empty string, and then fold \texttt{op} over the outputs of \(a_1\) after that. This is implemented directly using the prefix-sum constructor.

\[A_{\text{prefix-sum}(a_1, \text{init}, \text{op})} := (A_{\text{eps}(\text{init})} \parallel A_{\text{eps}(\text{init})}) \cdot \left(\oplus_{\text{op}} A_{a_1}\right).\]

We need to use \(A_{\text{eps}(\text{init})} \parallel A_{\text{eps}(\text{init})}\) just because the right side of the concatenation has two inputs. It can be seen that \(A_{\text{eps}(\text{init})} \parallel A_{\text{eps}(\text{init})}\) is equivalent to the DT obtained from a data function which returns \((I, I)\), so this concatenation is concatenation with a data function. Restartability is preserved by applying the theorem directly.

\textbf{Fill operations: fill, fill-with.} These operations are ways to \textit{fill in} the values which are \(\bot\) and \(\top\) with other values. This will not preserve restartability, so it is only allowed in top-level queries; however, it is useful to do this in order to get a query defined on all input data words, so that comparison \(\beta_1 \text{ comp } \beta_2\) can be applied. The query \(\text{fill}(a_1)\) always returns the last defined value returned by \(a_1\). For instance, if the sequence of outputs of \(a_1\) is \(\bot, \top, 3, \top, 4, 5, \bot\), the outputs of \(\text{fill}(a_1)\) should be \(\bot, \bot, 3, 3, 4, 5, 5\). The query \(\text{fill-with}(a_1, a_2)\), instead of outputting the last defined value returned by \(a_2\), just outputs the value returned by \(a_2\) if \(a_1\) is not defined. So, if \(a_2\) is the constant always returning 0, the sequence of outputs of \(\text{fill-with}(a_1, a_2)\) should be \(0, 0, 3, 0, 4, 5, 0\).

To accomplish these constructions, we first obtain two DTs \(A_+\) and \(A_-\) which are defined when \(a_1\) is defined and when \(a_1\) is not defined, respectively. We define
\[A_+ := [I_{a_1} \in \mathbb{D}]\]
\[A_- := [I_{a_1} = \bot] \cup [I_{a_1} = \top].\]

Note that the constructions for \([= \bot]\) and \([= \top]\) cause quadratic blowup. The trick here is to use \(I\) in the argument to those constructions, instead of \(A\). By doing so, \(A_+\) and \(A_-\) only have quadratic size. Now, let \(\text{fst}, \text{snd} : \mathbb{D}^2 \rightarrow \mathbb{D}\) be the first and second projections (these are really just the terms \(y_1 \in \text{TM}[Y]\) and \(y_2 \in \text{TM}[Y]\) for \(Y = (y_1, y_2)\)). We then implement these as
\[\text{fill-with}(a_1, a_2) := \text{combine}(A_{a_1}, A_+, \text{fst}) \cup \text{combine}(A_{a_2}, A_-, \text{fst})\]
\[\text{fill}(a_1) := \oplus_{G_{\text{op}}} (\text{combine}(A_{a_1}, A_+, \text{fst}) \parallel A_-),\]
where \( t \) is a term which expresses how to update the fill result based on the previous fill result, and whether \( \mathcal{A}_{\alpha_1} \) is defined or not: if defined, we should take the new defined value, and otherwise, we should take the old fill result.

**Comparison:** \( \leq, \geq, = \). The semantics of \( \beta_1 \comp \beta_2 \) is just to apply \( \comp \): for example if \( \comp \) is \(<\), and if \( y_1 \) and \( y_2 \) are the outputs of \( \beta_1 \) and \( \beta_2 \) (which are always defined), then \( \beta_1 < \beta_2 \) should output \( y_1 < y_2 \) (which is 0 if false, 1 if true). Therefore, this construction is actually equivalent to \( \comp \beta_1, \beta_2, \comp \). so this is how we implement it. We do not need to worry about restartability for temporal queries, and we also don’t define \( I \).

**Boolean operators:** \( \land, \lor, \rightarrow, \leftrightarrow, \neg \). The boolean operators similarly correspond exactly to applying the corresponding operation. For example, \( \varphi_1 \lor \varphi_2 \) is equivalent to \( \comp \varphi_1, \varphi_2, \lor \).

**Past-temporal operators:** \( \circ, \Box, \diamond, S_W, S_S \). Finally, we have the past temporal operators. These have the usual semantics on finite traces: for example \( \circ \varphi_1 \) says that \( \varphi_1 \) was true at the previous item, and is false initially, and \( \diamond \varphi_1 \) says that \( \varphi_1 \) was true at some point in the trace up to this point (including at the present time). The implementation of \( \circ \) uses concatenation while the others all use prefix-sum.

**Resulting theorem.** Our implementations give us the following theorem. In particular, the evaluation of any query on an input data stream requires a quadratic number of registers and a quadratic number of operations per element, independent of the length of the stream.

**Theorem 4.1.** For any well-typed base-level quantitative query \( \alpha \) except at the top level, the compilation described above via the constructions of §3 produces a restartable DT \( \mathcal{A}_\alpha \) of quadratic size in the length of the query. For any well-typed top-level quantitative query \( \beta \) or temporal query \( \varphi \), the compilation produces a DT of quadratic size which implements the semantics.

### 4.3 Case study: cardiac arrhythmia detection

We will present now an application of the QRE-P\( _{\text{Past}} \) monitoring language to the medical domain, in particular for the detection of potentially fatal cardiac arrhythmias. In particular, we consider Implantable Cardioverter Defibrillators (ICDs), which are devices that continually monitor the electrical activity of the heart in order to detect potentially fatal arrhythmias. If a dangerous arrhythmia is detected, then the ICD delivers a powerful electrical shock to the patient in order to restore the normal rhythm of the heart.

The type of fatal arrhythmia that ICDs detect is called *Ventricular Tachycardia* (VT), and its identification is based on the electrical signal that the device records directly from the right ventricle (lower-right chamber of the heart). As a first step, the heart signal is analyzed by the device to determine its *peaks*, which correspond to the ventricular heart beats (contractions). We will assume here that the input signal already includes the information about peaks. More specifically, we suppose that the data items of the input stream are of the form \((b, \text{seq})\), where \(b\) is a Boolean value, and \(\text{seq}\) is a sequence number. The presence (resp., absence) of a heartbeat is indicated with \(b = 1\) (resp., \(b = 0\)). The signal is uniformly sampled and the sampling period is \(T\). So, the timestamp of the data item \((b, \text{seq})\) is equal to \(\text{seq} \cdot T\).

The arrhythmia detection algorithm is based on the length of the heartbeat intervals, and uses several criteria that are called *discriminators* in the medical literature:

- **Initial Rhythm Classification** (IRC): The current interval length and the average of the four most recent interval lengths are both below a threshold \(T_{\text{IRC}}\).
- **Sudden Onset** (SO): This discriminator corresponds to the clinical observation that VT typically occurs suddenly. It quantifies the suddenness of tachycardia using the last nine interval lengths...
\(I_1, \ldots, I_9\) (where \(I_9\) is the last interval length). The onset of tachycardia is considered to be sudden if at least one of the differences \(I_1 - I_9, \ldots, I_8 - I_9\) is greater than a threshold \(T_{SO}\).

- **Rhythm Stability (RS):** The heartbeat interval lengths during VT usually display low variability. The rhythm stability discriminator quantifies variability as the difference between the second longest and the second shortest interval lengths among the last ten heartbeat intervals. If this difference is less than a threshold \(T_{RS}\), then the rate is considered to be stable.

At the top level, the query for VT detection is a Boolean combination (in particular, a conjunction) of the three above discriminators. Each discriminator can be described modularly by specifying a computation that processes a single heartbeat interval, and then composing these subcomputations sequentially a fixed or an arbitrary number of times. This high-level structure of the VT detection algorithm suggests the need for a language that combines Boolean operators with sequence-based pattern matching and quantitative aggregation. The QRE-PAST language provides all these features, and therefore it facilitates the modular description of the VT detection algorithm. The DT framework then provides the constructs for compiling the high-level description into an efficient streaming algorithm with precise bounds on space and per-element time usage: the complexity (for both resources) is constant in the length of the stream and quadratic in the size of the specification.

We start by describing a query that computes the length of a single heartbeat interval. The pattern is \(0^+ \cdot 1\), which describes an arbitrary number of samples that correspond to the absence of a heartbeat followed by a single heartbeat. The query \(f_{0^+1}\) (shown below) matches a heartbeat interval and outputs its length.

\[
\begin{align*}
    f_0 &= \text{atom}(0, \text{seq}) \quad \text{// rate 0} \\
    f_{0^+} &= \text{iter}(f_0, 0, (x,y) \rightarrow 0) \quad \text{// rate 0^+} \\
    f_{0^+} &= \text{split}(f_0, f_{0^+}, (x,y) \rightarrow x) \quad \text{// rate 0^+} \\
    f_1 &= \text{atom}(1, \text{seq}) \quad \text{// rate 1} \\
    f_{0^+1} &= \text{split}(f_{0^+}, f_1, (x,y) \rightarrow |y-x| \cdot T) \quad \text{// rate 0^+1}
\end{align*}
\]

Now, the query \(f_{\text{last}}\) (shown below) outputs the length of the last interval:

\[
\begin{align*}
    f_{\text{prev}} &= \text{iter}(f_{0^+1}, 0, (x,y) \rightarrow 0) \quad \text{// rate (0^+1)^*} \\
    f_{\text{last}} &= \text{split}(f_{\text{prev}}, f_{0^+1}, (x,y) \rightarrow y) \quad \text{// rate (0^+1)^+}
\end{align*}
\]

W.l.o.g. we can assume that the input stream always starts with a 0-tagged data item (absence of heartbeat). Now, the average of the last four intervals is given by the query

\[
f_{\text{last}4} = \text{split}(f_{\text{prev}}, f_{0^+1}, f_{0^+1}, f_{0^+1}, (x,y_1,y_2,y_3,y_4) \rightarrow (y_1 + y_2 + y_3 + y_4)/4)
\]

with rate \((0^+1)^* \cdot (0^+1)^4\). The satisfaction of the IRC discriminator is described by the conjunction, denoted \(\varphi_{IRC}\) of the following two QRE-PAST formulas:

\[
\begin{align*}
    \text{fill-with}(f_{\text{last}}, T_{IRC}) < T_{IRC} \quad & \text{fill-with}(f_{\text{last}4}, T_{IRC}) < T_{IRC}
\end{align*}
\]

Notice that this conjunction can only be satisfied at the occurrence of a heartbeat. Since the discriminators SO and RS are similar to IRC in that they are computed using a fixed number of the most recent heartbeat intervals, we leave it as an exercise to the reader to write the formulas \(\varphi_{SO}\) and \(\varphi_{RS}\). Finally, detection of VT is given by the formula \(\varphi_{IRC} \land \varphi_{SO} \land \varphi_{RS}\).

Using QRE-PAST instead of the QRE (or StreamQRE) language for this application has the advantage that it directly allows the use of tests on aggregates. In [Abbas et al. 2018] this behavior was encoded by annotating the stream with additional information, streaming the intermediate output to another processing stage (using the operation of streaming composition), and then applying tests on the annotated data items.
5 SUCCINCTNESS AND EXPRESSIVENESS

5.1 Comparison with cost-register automata

Cost Register Automata (CRAs) were introduced in [Alur et al. 2013] as a machine-based characterization of the class of regular transductions, which is a notion of regularity that relies on the theory of MSO-definable string-to-tree transductions. The advantage of CRAs over other approaches is that they suggest an obvious algorithm for computing the output in a streaming manner. A CRA has a finite-state control that is updated based only on the tag values of the input data word, and a finite set of write-only registers that are updated at each step using the given operations. The original CRA model is a deterministic machine, whose registers can hold data values as well as functions represented by terms with parameters. Each register update is required to be copyless, that is, a register can appear at most once in the right-hand-side expressions of the updates.

In [Alur et al. 2018], the class of Streamable Regular (SR) transductions is introduced, which has two equivalent characterizations: in terms of MSO-definable string-to-dag (directed acyclic graph) transductions without backward edges, and in terms of possibly copyful CRAs. Since the focus is on streamability, and terms can grow linearly with the size of the input stream, the registers are restricted to hold only values, not terms. This CRA model is equivalent to DTs.

Theorem 5.1. The class of transductions computed by DTs is equal to the class SR.

Proof sketch. It suffices to show semantics-preserving translations from CRAs to DTs and vice versa. Suppose $\mathcal{A} = (Q, X, \Delta, I, F)$ is a CRA. We construct a data transducer $\mathcal{B}$ with states $Q \times X$, that is, one state per state/variable pair of the CRA. For the second part of the theorem, suppose $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ is a data transducer. We construct a CRA $\mathcal{B}$ with states $\{\perp, \star, \top\} \times X$ and variables $X$. A configuration of $\mathcal{B}$ consists of a state in $\{\perp, \star, \top\} \times X$ and an assignment $\mathbb{D}^X$, and therefore specifies uniquely a configuration $\{\perp, \mathbb{D}, \top\}^X$ of $\mathcal{A}$. For a configuration $\{\perp, \mathbb{D}, \top\}^X$ of $\mathcal{A}$ and a letter $a$, the transition $\Delta_a$ can then be modeled with an $a$-labeled transition in the CRA $\mathcal{B}$. □

The benefit of DTs, however, is that they enable succinct modular constructions, whereas CRAs do not. For example, the parallel composition of CRAs requires a product construction, whereas the parallel composition of DTs employs a disjoint union construction. This implies that a multiple parallel composition of CRAs can cause an exponential blowup of the state space, but the corresponding construction on DTs causes only a linear increase in size.

Theorem 5.2. For a fixed $(\mathbb{D}, Op)$, DTs can be exponentially more succinct than unambiguous CRAs.

Proof sketch. Consider the case where $\Sigma$ contains $k$ different tags, and $\mathbb{D}$ is the set of natural numbers. The only operation included in $Op$ is addition. Suppose that $\mathcal{A}_i$ for $i = 1, \ldots, k$ is a data transducer over $\Sigma \times \mathbb{D}$ that outputs the sum of all values if the input contains the $i$-th tag (for some fixed enumeration of $\Sigma$) and 0 otherwise. Notice that $\mathcal{A}_i$ can be implemented with two data variables. Now, $\mathcal{B}$ is the data transducer that combines $\mathcal{A}_1, \ldots, \mathcal{A}_k$ using the $k$-ary addition operation. A CRA that implements the same function as $\mathcal{B}$ needs finite control that remembers which tags have appeared so far. This implies that the CRA needs exponentially many states, and this is true even if unambiguous nondeterminism is allowed. □

5.2 Comparison with finite-state automata

Another perspective on succinctness is to compare DTs with finite automata for expressing regular languages. To simplify this, consider DTs over a singleton data set $\mathbb{D} = \{\star\}$, with no initial states and one final state. Each such DT $\mathcal{A}$ computes a regular languages $\mathcal{L}(\mathcal{A})$. If we further restrict to acyclic DTs, they are exactly as succinct as reversed alternating finite automata (r-AFA). In particular, this implies that acyclic DTs (and hence DTs) are exponentially more succinct than DFAs and NFAs.
An r-AFA [Chandra et al. 1981; Salomaa et al. 2000] consists of \((Q, \Sigma, \delta, I, F)\) where the transition function \(\delta\) assigns to each state in \(Q\) a boolean combination of the previous values of \(Q\). For example, we could assign \(\delta(q_3) = q_1 \land (q_2 \lor \neg q_3)\). An r-AFA is equivalent to an AFA where the input string is read in the opposite order. The translation from DT to r-AFA copies the states, and on each update, sets each state to be equal to the disjunction of the transitions into it, where each transition is the conjunction of the source variables. Thus, the total size of \(\delta\) is bounded by the size of the DT. For the other direction, we first remove negation in the standard way; then, conjunction becomes \(\land\) and disjunction becomes \(\lor\) (multiple transitions with a single target) in the DT.

It is known [Chandra et al. 1981; Fellah et al. 1990] that \(L\) is recognized by a r-AFA with \(n\) states if and only if it is recognized by a DFA with \(2^n\) states. This gives an exponential gap in state complexity between acyclic data transducers and finite automata, both DFAs and NFAs. To see the gap for NFAs, consider a DFA with \(2^n\) states which has no equivalent NFA with a fewer number of states. Acyclic data transducers are a special case, so data transducers are exponentially more succinct than both DFAs (uniformly) and NFAs (in the worst case).

### 5.3 Comparison with general stream-processing programs

Finally, we consider a general model of computation for efficient streaming algorithms. The maintained state consists of a fixed number of Boolean and data variables. The behavior of the algorithm is defined by providing an initialization function, an update function, a distinguished output data variable and a Boolean output flag (which is set when output is present). The initialization and update functions are specified using a loop-free imperative language with the following constructs: assignments, sequential composition, and conditionals. This model captures all efficient (bounded space and per-element processing time) streaming computations over a set of allowed data operations \(\text{Op}\). We write \text{Stream}(\text{Op})\) to denote the class of such efficient streaming algorithms. The problem with the class \text{Stream}(\text{Op})\) is that it is not suitable for modular specifications. As the following theorem shows, it is not closed under the \text{split} combinator.

**Theorem 5.3.** Let \(\Sigma = \{a, b\}\), \(\mathbb{D} = \mathbb{N}\), and let \(\text{Op}\) be the family of operations that includes unary incrementation, unary decrementation, the constant 0, and the binary equality predicate. Define the transductions \(f, g : (\Sigma \times \mathbb{D})^* \rightarrow \mathbb{D}\) as follows:

\[
L(f) = \{ w \in \Sigma^* : |w|_a = 2 \cdot |w|_b \} \\
L(g) = \{ w \in \Sigma^* : |w|_a = |w|_b \}
\]

\[
f(w) = \begin{cases} 
1, & \text{if } w \downarrow \Sigma \in L(f) \\
\bot, & \text{otherwise}
\end{cases} \\
g(w) = \begin{cases} 
1, & \text{if } w \downarrow \Sigma \in L(g) \\
\bot, & \text{otherwise}
\end{cases}
\]

where \(|w|_a\) is notation for the number of \(a\)'s that appear in \(w\). Both \(f\) and \(g\) are streamable functions, but \(h = \text{split}(f, g, (x, y) \rightarrow 1)\) is not.

**Proof Sketch.** Both \(f\) and \(g\) can be implemented efficiently by maintaining two counters for the number of \(a\)'s and the number of \(b\)'s seen so far. With a simple combinatorial argument it can be established that any streaming algorithm that computes \(h\) requires a linear number of bits (in the size of the stream seen so far). Any streaming algorithm over \(\text{Op}\), however, employs a finite number of integer registers whose size (in bits) can grow only logarithmically.

Theorem 5.3 suggests that some restriction on the domains of transductions is necessary in order to maintain closure under modular constructions. We therefore enforce regularity of a generic streaming algorithm by requiring that the values of the Boolean variables depend solely on the input tags. That is, they do not depend on the input data values or the values of the data variables. Under this restriction, a streaming algorithm can be encoded as a DT of roughly the same size.
Theorem 5.4. A streaming algorithm of Stream(Op) that satisfies the regularity restriction can be implemented by a DT over Op. This construction can be performed in linear time and space.

Proof sketch. Consider an arbitrary streaming algorithm of Stream(Op) that satisfies the regularity restriction. Its data variables are encoded as DT variables that are always active. A Boolean variable \( b \) is encoded using two DT variables \( x_b \) and \( x_{\overline{b}} \) as follows: if \( b = 0 \) then \( x_b = \bot \) and \( x_{\overline{b}} = d_\star \), and if \( b = 1 \) then \( x_b = d_\star \) and \( x_{\overline{b}} = \bot \), where \( d_\star \) is any fixed element of \( D \).

6 RELATED WORK

6.1 Quantitative Automata

The literature contains various proposals of automata-based models that are some kind of quantitative extension of classical finite-state automata.

Weighted Automata, which were introduced in 1961 by Schützenberger [Schützenberger 1961] (see also the more recent monograph [Droste et al. 2009]), extend nondeterministic finite-state automata by annotating transitions with weights (which are elements of a semiring) and can be used for the computation of simple quantitative properties. A weighted automaton maps an input string \( w \) to the minimum over costs of all accepting paths of the automaton over \( w \).

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 [Björklund and Schwentick 2010; Bojańczyk et al. 2011; Demri and Lazic 2009; Kaminski and Francez 1994; Neven et al. 2004] studies models that operate on words over an infinite alphabet, which is typically of the form \( \Sigma \times \mathbb{N} \), where \( \Sigma \) 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 [Neven et al. 2004].

The work on Cost Register Automata (CRAs) [Alur et al. 2013; Alur and Raghothaman 2013] and Streaming Transducers [Alur and Černý 2010; Alur and D’Antoni 2012; Alur and Černý 2011] is about models where the control and data registers are kept separate by allowing write access to the registers but no testing. As discussed in §5, DTs are exponentially more succinct than CRAs. The exponential gap arises for the useful construction of performing several subcomputations in parallel and combining their results. DTs recognize the class of streamable regular functions [Alur et al. 2018], which is equivalently defined by QREs, CRAs, and attribute grammars (§5).

The recent work [Bojańczyk et al. 2018] gives a characterization of the first-order (resp., MSO) definable string-to-string transformations using an algebra of functions that operate on objects such as lists, lists of lists, pairs of lists, lists of pairs of lists, and so on. Monitors with finite-state control and unbounded integer registers are studied in [Ferrère et al. 2018] and a hierarchy of expressiveness is established on the basis of the number of available registers. These papers focus on issues related to expressiveness, whereas we focus here on the issues of modularity and succinctness.

6.2 Query languages for runtime monitoring

Runtime monitoring (see the survey [Leucker and Schallhart 2009]) is a lightweight verification technique for testing whether a finite execution trace of a system satisfies a given specification. The specification is translated into a monitor, which executes along with the monitored system: it consumes system events in a streaming manner and outputs the satisfaction or falsification of the specification. A widely used formalism for describing specifications for monitoring is Linear Temporal Logic (LTL) [Havelund and Rosu 2004]. Metric Temporal Logic (MTL) has been used for
monitoring real-time temporal properties [Thati and Roşu 2005]. Signal Temporal Logic (STL), which extends MTL with value comparisons, has been used for online monitoring of real-valued signals [Deshmukh et al. 2017]. However, prior formalisms for monitoring either completely lack quantitative features or they do not allow a rich set of quantitative operations as we do here.

The line of work on synchronous languages [Benveniste et al. 2003] also deal with the processing of data streams. The focus, however, in the design of these languages is the decomposition of the overall computation into logically concurrent tasks. Here, we focus on the control structure for parsing the input stream and applying quantitative aggregators.

7 CONCLUSIONS

Data transducers are a succinct, implementation-level machine model for general streaming computations. They combine finite control (active vs. inactive states) and register updates (e.g. in CRAs) into a single integrated model, with state variables that can be undefined, defined, or conflicted. Our main goal in fine-tuning the details of the design was to ensure succinct union, concatenation, iteration, and parallel composition constructions in §3, along with a few others.

We have also presented a query language, QRE-PAST, which combines elements of StreamQRE [Mamouras et al. 2017] with past-time temporal logic operators [Havelund and Roşu 2004], such that queries compile modularly to DTs with precise complexity guarantees. We have formally justified in §5 that DTs are exponentially more succinct over CRAs, over DFAs and NFAs, and that a certain subset of streaming algorithms given by arbitrary code translates linearly to DTs.

In future work, we hope to explore opportunities for query optimization using the DT model. The DT framework lends itself more easily to this purpose than QREs or unstructured streaming algorithms. In the case of NFAs, bisimulation relations can be used to reduce the size of the automata via a quotient construction, and it seems plausible that an analogous notion can be defined for DTs to reduce the number of variables. Such optimizations could benefit the Java implementation of the StreamQRE language, which is reported in [Mamouras et al. 2017].

REFERENCES


Stéphane Demri and Ranko Lazić. 2009. LTL with the Freeze Quantifier and Register Automata. *ACM Transactions on Computational Logic (TOCL)* 10 (2009), 16:1–16:30.


