

Regular Programming for Quantitative Properties of Data Streams^{*}

Rajeev Alur, Dana Fisman, and Mukund Raghothaman

University of Pennsylvania
{alur, fisman, rmukund}@cis.upenn.edu

Abstract. We propose *quantitative regular expressions* (QREs) as a high-level programming abstraction for specifying complex numerical queries over data streams in a modular way. Our language allows the arbitrary nesting of orthogonal sets of combinators: (a) generalized versions of choice, concatenation, and Kleene-iteration from regular expressions, (b) streaming (serial) composition, and (c) numerical operators such as min, max, sum, difference, and averaging. Instead of requiring the programmer to figure out the low-level details of what state needs to be maintained and how to update it while processing each data item, the regular constructs facilitate a global view of the entire data stream splitting it into different cases and multiple chunks. The key technical challenge in defining our language is the design of typing rules that can be enforced efficiently and which strike a balance between expressiveness and theoretical guarantees for well-typed programs. We describe how to compile each QRE into an efficient streaming algorithm. The time and space complexity is dependent on the complexity of the data structure for representing terms over the basic numerical operators. In particular, we show that when the set of numerical operations is sum, difference, minimum, maximum, and average, the compiled algorithm uses constant space and processes each symbol in the data stream in constant time outputting the cost of the stream processed so far. Finally, we prove that the expressiveness of QREs coincides with the streaming composition of regular functions, that is, MSO-definable string-to-term transformations, leading to a potentially robust foundation for understanding their expressiveness and the complexity of analysis problems.

1 Introduction

In a diverse range of applications such as financial tickers, data feeds from sensors, network traffic monitoring, and click-streams of web usage, the core computational problem is to map a stream of data items to a numerical value. Prior research on stream processing has focused on designing space-efficient algorithms for specific functions such as computing the average or the median of a sequence of values, and integrating stream processing in traditional data management software such as relational databases and spreadsheets. Our goal is orthogonal, namely, to

^{*} This research was partially supported by NSF Expeditions award CCF 1138996.

provide high-level programming abstractions for the modular specification of complex queries over data streams, with a mix of numerical operators such as sum, difference, min, max, and average, supported by automatic compilation into an efficient low-level stream processing implementation.

To motivate our work, suppose the input data stream consists of transactions at a bank ATM, and consider how the following query f can be expressed in a natural, modular, and high-level manner: “On average, how much money does Alice deposit into her account during a month?” We can express this query naturally as a composition of two queries: the query f_1 that maps the input stream to a stream consisting of only the transactions corresponding to the deposits by Alice, and the query f_2 that computes the average of the sum of deposits during each month. This form of filtering, and cascaded composition of stream processors, is common in many existing stream processing languages (such as ActiveSheets [35]) and systems (such as Apache Storm), and our proposal includes a *streaming composition* operator to express it: $f_1 \gg f_2$.

Now let us turn our attention to expressing the numerical computation f_2 . An intuitive decomposition for specifying f_2 is to break up the input stream into a sequence of substreams, each corresponding to the transactions during a single month. We can write a function f_3 that maps a sequence of transactions during a month to its cumulative sum. The desired function f_2 then splits its input stream into substreams, each matching the input pattern of f_3 , applies f_3 to each substream, and combines the results by averaging. In our proposed calculus, f_2 is written as *iter-avg*(f_3). This is exactly the “quantitative” generalization of the Kleene-* operation from regular expressions—a declarative language for specifying patterns in strings that is widely used in practical applications and has a strong theoretical foundation. However, we are not aware of any existing language for specifying quantitative properties of sequences that allows such a regular iteration over chunks of inputs.

As in regular expressions, iterators in our language can be nested, and one can use different aggregation operators at different levels. For example, to specify the modified query, “On an average, during a month, what is the maximum money that Alice deposits during a single day?” in the program for the original query f , we can essentially replace the computation f_3 for processing month-substreams by *iter-max*(f_4), where the function f_4 maps a sequence of transactions during a *single day* to its sum. Analogous to the iteration, the generalization of the concatenation operation *split-op*(f, g), splits the input stream in two parts, applies f to the first part, g to the second part, and combines the results using the arithmetic combinator *op*. If we want to process streams in which no individual withdrawal exceeds a threshold value in a special manner, our program can be *f else g*, where the program f is written to process streams where all withdrawals are below the threshold and g handles streams with some withdrawal above the threshold. These core regular constructs, *else*, *split*, and *iter*, are natural as they are the analogs of the fundamental programming constructs of conditionals, sequential composition, and iteration, but also allow the programmer a global view of the entire stream.

We formalize this idea of regular programming by designing the language of *quantitative regular expressions* for processing data streams (section 2). Each quantitative regular expression (QRE) maps a (regular) subset of data streams to cost values. The language constructs themselves are agnostic to the cost types and combinators for combining cost values. The design of the language is influenced by two competing goals: on one hand, we want as much expressiveness as possible, and on the other hand, we want to ensure that every QRE can be automatically compiled into a streaming algorithm with provably small space and time complexity bounds. For the former, the same way as deterministic finite automata and regular languages serve as the measuring yardstick for design choices in formalizing regular expressions, the most suitable class of functions is the recently introduced notion of *regular functions* [7]. Unlike the better known and well-studied formalism of weighted automata, whose definition is inherently limited to costs with two operations that form a commutative semiring [21], this class is parametrized by an arbitrary set of cost types and operations. It has both a machine-based characterization using *streaming string-to-term transducers* [7] and a logic-based characterization using *MSO-definable string-to-term transformations* [16]. While this class is *not* closed under streaming composition, in section 4, we show that QREs define exactly streaming compositions of *regular functions*. This expressiveness result justifies the various design choices we made in defining QREs.

In section 3, we show how to compile a QRE into an efficient streaming algorithm. The implementation consists of a set of interacting machines, one for each sub-expression. To process operators such as *split* and *iter*, the algorithm needs to figure out how to split the input stream. Since the splitting cannot be deterministically computed in a streaming manner, the algorithm needs to keep track of all potential splits. The typing rules are used to ensure that the number of splits under consideration are proportional to the size of the expression rather than the length of the input stream. While the compilation procedure is generic with respect to the set of cost types and operators, the exact time and space complexity of the algorithm is parametrized by the complexity of the data structure for representing cost terms. For example, every term that can be constructed using numerical constants, a single variable x , and operations of \min and $+$, is equivalent to a term in the canonical form $\min(x + a, b)$, and can therefore be summarized by two numerical values allowing constant-time updates. When we have both numerical values and sequences of such values, if we know that the only aggregate operator over sequences is averaging, then the sequence can be summarized by the sum of all values and the length (such a succinct representation is not possible if, for instance, we allow mapping of a sequence of values to its median). In particular, we show that when a QRE is constructed using the operations of sum, difference, minimum, maximum, and average, the compiled streaming algorithm has time complexity that is linear in the length of the data stream (that is, constant time for processing each symbol) and constant space complexity (in terms of the size of the QRE itself, both complexities are

polynomial). This generalizes the class of queries for which efficient streaming algorithms have been reported in the existing literature.

2 Quantitative Regular Expressions

$f, g, \dots ::= \varphi ? \lambda$ $\epsilon ? x$	Basic functions
$op(f_1, f_2, \dots, f_k)$ $f[x/g]$	Cost operations
$f \text{ else } g$ $\textit{split}(f \langle x \rangle g)$ $\textit{split}(f \langle x \rangle g)$ $\textit{iter}[x](f[g_1, g_2, \dots, g_k])$ $\textit{iter}\langle x \rangle(f[g_1, g_2, \dots, g_k])$	Regular operators
$f \gg g$	Streaming composition

Fig. 2.1. List of expression combinators.

2.1 Preliminaries

The data and cost types. We first fix a set $\mathcal{T} = \{T_1, T_2, \dots\}$ of types, each of which is non-empty. Typical examples might include the sets \mathbb{R} , \mathbb{Z} and \mathbb{N} of real numbers, integers and natural numbers respectively, the set $\mathbb{B} = \{\textit{true}, \textit{false}\}$ of boolean values, and the set \mathbb{M} of multisets of real numbers.

Another example is $D_{\textit{bank}} = \mathbb{R} \cup \{\textit{end}_d, \textit{end}_m\}$, indicating the transactions of a customer with a bank. The symbol \textit{end}_d indicates the end of a working day, \textit{end}_m indicates the passage of a calendar month, and each real number $x \in \mathbb{R}$ indicates the deposit (withdrawal if negative) of x dollars into the account.

Data predicates. For each $D \in \mathcal{T}$, let Φ_D be a non-empty collection of predicates over D . In the case of $D_{\textit{bank}}$, an example choice is $\Phi_{D_{\textit{bank}}} = \{d = \textit{end}_d, d = \textit{end}_m, d \in \mathbb{R}, d \geq 0, d < 0, \dots\}$. We require the following:

1. Φ_D be closed under Boolean connectives: for each $\varphi_1, \varphi_2 \in \Phi_D$, $\varphi_1 \vee \varphi_2, \varphi_1 \wedge \varphi_2, \neg \varphi_1 \in \Phi_D$,
2. whether a data value d satisfies a predicate φ is decidable, and
3. there is a quantity $\textit{time}_{\varphi\textit{-sat}}$ such that the satisfiability of each predicate φ is mechanically decidable in time $\leq \textit{time}_{\varphi\textit{-sat}}$. Satisfiability of predicates would typically be determined by an SMT solver [20].

We refer to a set of predicates satisfying these properties as an *effective boolean algebra*.

Data streams, data languages, and symbolic regular expressions. For each $D \in \mathcal{T}$, a *data stream* is simply an element $w \in D^*$, and a *data language* L is a subset of D^* .

Symbolic regular expressions provide a way to identify data languages L . Our definitions here are mostly standard, except for the additional requirement of *unambiguous parseability*, which is for uniformity with our later definitions of function combinators. They are *symbolic* because the basic regular expressions are predicates over D . Through the rest of this paper, the unqualified phrase “regular expression” will refer to a “symbolic” regular expression.

Consider two non-empty languages $L_1, L_2 \subseteq D^*$. They are *unambiguously concatenable* if for each stream $w \in L_1 L_2$, there is a unique pair of streams $w_1 \in L_1$ and $w_2 \in L_2$ such that $w = w_1 w_2$. The language L is *unambiguously iterable* if L is non-empty and for each stream $w \in L^*$, there is a unique sequence of streams $w_1, w_2, \dots, w_k \in L$ such that $w = w_1 w_2 \dots w_k$.

We write $\llbracket r \rrbracket \subseteq D^*$ for the language defined by the symbolic regular expression r . *Symbolic unambiguous regular expressions* are inductively defined as follows:

1. ϵ is a regular expression, and identifies the language $\llbracket \epsilon \rrbracket = \{\epsilon\}$.
2. For each predicate $\varphi \in \Phi$, φ is a regular expression, and $\llbracket \varphi \rrbracket = \{d \in D \mid \varphi(d) \text{ holds}\}$.
3. For each pair of regular expressions r_1, r_2 , if $\llbracket r_1 \rrbracket$ and $\llbracket r_2 \rrbracket$ are disjoint, then $r_1 + r_2$ is a regular expression and $\llbracket r_1 + r_2 \rrbracket = \llbracket r_1 \rrbracket \cup \llbracket r_2 \rrbracket$.
4. For each pair of regular expressions r_1, r_2 , if $\llbracket r_1 \rrbracket$ and $\llbracket r_2 \rrbracket$ are unambiguously concatenable, then $r_1 r_2$ is a regular expression which identifies the language $\llbracket r_1 r_2 \rrbracket = \llbracket r_1 \rrbracket \llbracket r_2 \rrbracket$.
5. When r is a regular expression such that $\llbracket r \rrbracket$ is unambiguously iterable, then r^* is also a regular expression, and identifies the language $\llbracket r^* \rrbracket = \llbracket r \rrbracket^*$.

Example 1. In the bank transaction example, the languages $\llbracket \mathbb{R}^* \text{end}_d \rrbracket$ and D_{bank}^* are unambiguously concatenable: the only way to split a string which matches $\mathbb{R}^* \text{end}_d \cdot D_{bank}^*$ is immediately after the first occurrence of end_d . On the other hand, observe that \mathbb{R}^* and $\mathbb{R}^* \text{end}_d$ are not unambiguously concatenable: the string $2, \text{end}_d \in \llbracket \mathbb{R}^* \rrbracket \cdot \llbracket \mathbb{R}^* \text{end}_d \rrbracket$ can be split as $2, \text{end}_d = (\epsilon)(2, \text{end}_d)$, where $\epsilon \in \llbracket \mathbb{R}^* \rrbracket$ and $2, \text{end}_d \in \llbracket \mathbb{R}^* \text{end}_d \rrbracket$, and can also be split as $2, \text{end}_d = (2)(\text{end}_d)$, where $2 \in \llbracket \mathbb{R}^* \rrbracket$ and $\text{end}_d \in \llbracket \mathbb{R}^* \text{end}_d \rrbracket$.

Similarly, the language $\llbracket \mathbb{R}^* \text{end}_d \rrbracket$ is unambiguously iterable: the only viable split for any matching string is immediately after each occurrence of end_d . The language $\llbracket \mathbb{R} \rrbracket$ is also unambiguously iterable, but observe that $\llbracket \mathbb{R}^* \rrbracket$ is not.

Given a regular expression r_1 and r_2 , let *min-terms* be the set of distinct character classes formed by predicates from Φ [18]. The first three claims below can be proved by straightforward extensions to the traditional regular expression-to-NFA translation algorithm [12,13,33]. The final claim is proved in [34].

Theorem 1. *Given unambiguous regular expressions r_1 and r_2 , the following problems can be decided in time $\text{poly}(|r_1|, |r_2|, |\text{min-terms}|) \text{time}_{\varphi\text{-sat}}$:*

1. Are $\llbracket r_1 \rrbracket$ and $\llbracket r_2 \rrbracket$ disjoint?

2. Are $\llbracket r_1 \rrbracket$ and $\llbracket r_2 \rrbracket$ unambiguously concatenable?
3. Is $\llbracket r_1 \rrbracket$ unambiguously iterable?
4. Are r_1 and r_2 equivalent, i.e. is $\llbracket r_1 \rrbracket = \llbracket r_2 \rrbracket$?

Example 2. Consider again the case of the customer and the bank. We picked $D_{bank} = \mathbb{R} \cup \{end_d, end_m\}$ and $\Phi_{D_{bank}} = \{d = end_d, d = end_m, d \in \mathbb{R}, d \geq 0, d < 0\}$. The transactions of a single day may then be described by the regular expression $r_{day} = (d \in \mathbb{R})^* \cdot (d = end_d)$, the transactions of a week, by the regular expression $r_{week} = r_{day}^7$. Months which involve only deposits are given by the regular expression $((d \geq 0) + (d = end_d))^* \cdot (d = end_m)$.

An important restriction for the predicates is that they are only allowed to examine individual data values. The language $w \in \mathbb{R}^*$ of monotonically increasing sequences can therefore not be expressed by a symbolic regular expression. Without this restriction, all problems listed in theorem 1 are undecidable [17].

Cost terms. Let $\mathcal{G} = \{op_1, op_2, \dots\}$ be a collection of operations over values of various types. Each operation is a function of the form $op : T_1 \times T_2 \times \dots \times T_k \rightarrow T$.

Example 3. Over the space of types $\mathcal{T}_s = \{\mathbb{Z}, \mathbb{B}\}$, a simple choice of operators is $\mathcal{G}_s = \{+, \min, \max, x < 7\}$ where $+, \min, \max : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$, are the usual operations of addition, minimum and maximum respectively, and the unary operator $x < 7 : \mathbb{Z} \rightarrow \mathbb{B}$ determines whether the input argument is less than 7.

Over the space of types $\mathcal{T}_m = \{\mathbb{R}, \mathbb{M}\}$ where \mathbb{M} is the set of multisets of real numbers, the chosen operations might be $\mathcal{G}_m = \{\text{ins}, \text{avg}, \text{mdn}\}$, where $\text{ins} : \mathbb{M} \times \mathbb{R} \rightarrow \mathbb{M}$ is insertion into sets, defined as $\text{ins}(A, x) = A \cup \{x\}$, and $\text{avg}, \text{mdn} : \mathbb{M} \rightarrow \mathbb{R}$ return the average and median of a multiset of numbers respectively.

Let $X = \{x_1, x_2, \dots\}$ be a sequence of parameters, and each parameter x_i be associated with a type $T_i \in \mathcal{T}$. We use $x : T$ when we want to emphasize that the type of x is T . Parameters in X can be combined using cost operations from \mathcal{G} to construct *cost terms*:

$$\tau ::= x \mid t \in T \mid op(\tau_1, \tau_2, \dots, \tau_k)$$

We will only consider those terms τ that are well-typed and *single-use*, i.e. where each parameter x occurs at most once in τ . For example, $\min(x, y)$ is a well-typed single-use term when both parameters are of type \mathbb{R} . On the other hand, the term $\min(x, x + y)$ is not single-use. Each term is associated with the set $Param(\tau) = \{x_{t_1}, x_{t_2}, \dots, x_{t_k}\}$ of parameters appearing in its description, and naturally encodes a function $\llbracket \tau \rrbracket : T_1 \times T_2 \times \dots \times T_k \rightarrow T$ from parameter valuations \bar{v} to costs $\llbracket \tau \rrbracket(\bar{v})$, where T_i is the type of the parameter x_i .

2.2 Quantitative regular expressions and function combinators

Informally, for some data and cost domains $D, C \in \mathcal{T}$, QREs map data streams $w \in D^*$ to terms $f(w)$ over the cost domain C . Formally, given the data domain

$D \in \mathcal{T}$, cost domain $C \in \mathcal{T}$, and the list of parameters $X = \langle x_1, x_2, \dots, x_k \rangle$, each QRE f identifies a function $\llbracket f \rrbracket : D^* \rightarrow (T_1 \times T_2 \times \dots \times T_k \rightarrow C)_\perp$.¹ Observe that whether $\llbracket f \rrbracket(w, \bar{v})$ is defined depends only on the data stream w , and not on the parameter valuation \bar{v} .²

In addition to the data domain D , cost domain C , and the parameters of interest $X = \langle x_1, x_2, \dots \rangle$, each QRE f is therefore also associated with a regular expression r , describing the subset of data streams over which it is defined: $\llbracket r \rrbracket = \{w \in D^* \mid \llbracket f \rrbracket(w) \neq \perp\}$. We will represent this by saying that f is of the form $\text{QRE}(r, X, C)$, or even more succinctly as $f : \text{QRE}(r, X, C)$.

We will now inductively define quantitative regular expressions. A summary of the syntax can be found in table 2.1.

Basic functions. If $\varphi \in \Phi_D$ is a predicate over data values $d \in D$, and $\lambda : D \rightarrow C$ is an operation in \mathcal{G} , then $\varphi ? \lambda : \text{QRE}(\varphi, \emptyset, C)$ is defined as follows:

$$\llbracket \varphi ? \lambda \rrbracket(w, \bar{v}) = \begin{cases} \lambda(w) & \text{if } |w| = 1 \text{ and } \varphi(w) \text{ is true, and} \\ \perp & \text{otherwise.} \end{cases}$$

For each data domain D and parameter x of type C , $\epsilon ? x$ is a $\text{QRE}(\epsilon, \{x\}, C)$. If the input stream w is empty, then it produces the output v_x , where v_x is the assignment to x in the parameter valuation \bar{v} :

$$\llbracket \epsilon ? x \rrbracket(w, \bar{v}) = \begin{cases} v_x & \text{if } w = \epsilon, \text{ and} \\ \perp & \text{otherwise.} \end{cases}$$

Example 4. In the bank transaction example, if we wish to count the number of transactions made by the customer in a month, we would be interested in the functions $d \in \mathbb{R} ? 1$ and $d \notin \mathbb{R} ? 0$, where 0 and 1 are the functions returning the constant values 0 and 1 respectively.

Cost operations. Consider two competing banks, which given the transaction history w of the customer, yield interest amounts of $f_1(w)$ and $f_2(w)$ respectively. The customer wishes to maximize the interest received: given the binary operation $\max : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ from \mathcal{G} , we are therefore interested in the $\text{QRE} \max(f_1, f_2)$.

More generally, pick an operator $op : T_1 \times T_2 \times \dots \times T_k \rightarrow C$ and expressions $f_1 : \text{QRE}(r_1, X_1, T_1)$, $f_2 : \text{QRE}(r_2, X_2, T_2)$, \dots , $f_k : \text{QRE}(r_k, X_k, T_k)$. If the domains are equal, $\llbracket r_1 \rrbracket = \llbracket r_2 \rrbracket = \dots = \llbracket r_k \rrbracket$, and the parameter lists are pairwise-disjoint, $X_i \cap X_j = \emptyset$, for all $i \neq j$, then $op(f_1, f_2, \dots, f_k)$ is an expression of the form $\text{QRE}(r_1, \bigcup_i X_i, C)$.

¹ Note our convention of describing partial functions $f : A \rightarrow B$ as total functions $f : A \rightarrow B_\perp$, where $B_\perp = B \cup \{\perp\}$ and $\perp \notin B$ is the undefined value. The *domain* $\text{Dom}(f)$ of f is given by $\text{Dom}(f) = \{a \in A \mid f(a) \neq \perp\}$.

² We will be flexible in our use of function application, and freely use both the uncurried form $\llbracket f \rrbracket(w, \bar{v})$ and the partial application $\llbracket f \rrbracket(w)$ which maps parameter valuations to costs.

Given an input stream $w \in D^*$, if $\llbracket f_i \rrbracket(w)$ is defined for each i , then:

$$\begin{aligned} \llbracket op(f_1, f_2, \dots, f_k) \rrbracket(w, \bar{v}) &= op(int_1, int_2, \dots, int_k), \text{ where} \\ &\text{for each } i, int_i = \llbracket f_i \rrbracket(w, \bar{v}). \end{aligned}$$

Otherwise, $\llbracket op(f_1, f_2, \dots, f_k) \rrbracket(w, \bar{v}) = \perp$.

Substitution. Informally, the expression $f[x/g]$ substitutes the result of applying g to the input stream into the parameter x while evaluating f .

Example 5. In the bank transaction example, the bank may determine interest rates by a complicated formula *rate* of the form $\text{QRE}(D_{bank}^*, \emptyset, \mathbb{R})$, which is possibly a function of the entire transaction history of the customer. Given the monthly interest rate $p \in \mathbb{R}$, we can define a formula *earnings* of the form $\text{QRE}(D_{bank}^*, \{p\}, \mathbb{R})$ which returns the total earnings of the customer's account. The QREs *rate* and *earnings* thus encode functions $\llbracket rate \rrbracket : D_{bank}^* \rightarrow \mathbb{R}$ and $\llbracket earnings \rrbracket : D_{bank}^* \times \mathbb{R} \rightarrow \mathbb{R}$ respectively. The QRE *earnings*[$p/rate$] plugs the result of the rate computation into the earning computation, and maps the transaction history to the total interest earned.

Formally, let f and g be of the form $\text{QRE}(r_f, X_f, T_f)$ and $\text{QRE}(r_g, X_g, T_g)$ respectively and with equal domains $\llbracket r_f \rrbracket = \llbracket r_g \rrbracket$, let $x \in X_f$ be of type T_g , and let $(X_f \setminus \{x\}) \cap X_g = \emptyset$. Then, $f[x/g]$ is of the form $\text{QRE}(r_f, (X_f \setminus \{x\}) \cup X_g, T_f)$. If $\llbracket f \rrbracket(w)$ and $\llbracket g \rrbracket(w)$ are both defined, then:

$$\llbracket f[x/g] \rrbracket(w, \bar{v}) = \llbracket f \rrbracket(w, \bar{v}[x/\llbracket g \rrbracket(w, \bar{v})]).$$

where $\bar{v}[x/t]$ replaces the value of x in \bar{v} with t . Otherwise, $\llbracket f[x/g] \rrbracket(w, \bar{v})$ is undefined.

Conditional choice. If f and g are QREs with disjoint domains then $f \text{ else } g$ is a QRE defined as follows:

$$\llbracket f \text{ else } g \rrbracket(w, \bar{v}) = \begin{cases} int_f & \text{if } int_f \neq \perp, \text{ and} \\ int_g & \text{otherwise,} \end{cases}$$

where $int_f = \llbracket f \rrbracket(w, \bar{v})$ and $int_g = \llbracket g \rrbracket(w, \bar{v})$ respectively.

Example 6. The expression $isPositive = d \geq 0 ? true \text{ else } d < 0 ? false$ examines a single customer transaction with the bank, and maps it to *true* if it was a deposit, and *false* otherwise.

The bank rewards program may reserve a higher reward rate for each transaction made in a “deposit-only” month, and a lower reward rate for other months. Say we have expressions *hiReward* and *loReward* which compute the total customer rewards in qualifying and non-qualifying months respectively. The expression $reward = hiReward \text{ else } loReward$ then maps the transactions of an arbitrary month to the total reward earned.

Observe that the choice of *isPositive* depends only on the next customer transaction. On the other hand, the choice in *reward* depends on the *entire* list of transactions made in that month, and resolving this choice requires a global view of the data stream.

Concatenation. In the bank transaction example, say we have the transactions of two consecutive months, $w \in r_{month}^2$, and that the QRE *count* returns the number of transactions made in a single month.

To express the minimum number of transactions for both months, a natural description is *split-min(count, count)*, where the combinator *split-min* splits the input string w into two parts w_1 and w_2 , applies *count* to each part, and combines the results using the min operator. Similarly, the expression *split-plus(count, count)* would total the number of transactions made in both months. One reasonable choice for function concatenation is therefore to have a k -ary combinator *split-op*, for each k -ary operator $op \in G$.

We instead choose to have a pair of uniform *binary, operator-agnostic* combinators *split*: *split(f [x] g)* and *split(f <x> g)*. They split the given input stream into two parts, applying f to the prefix and g to the suffix, and use parameter x to pass the result from f to g in the case of $[x]$ and vice-versa in the case of $\langle x \rangle$ (see figure 2.2). Therefore the sub-expressions f and g themselves determine how the intermediate results are combined. The *split-op* combinators are just a special case of this more general construct (see example 7). We now formalize this intuition.

Let $f : \text{QRE}(r_f, X_f, T_f)$ and $g : \text{QRE}(r_g, X_g, T_g)$ be a pair of QREs whose domains $\llbracket r_f \rrbracket$ and $\llbracket r_g \rrbracket$ are unambiguously concatenable. Let $x : T_f$ be a parameter in X_g and let $X_f \cap (X_g \setminus \{x\}) = \emptyset$. Then *split(f [x] g)* is a QRE of the form $\text{QRE}(r_f \cdot r_g, X, T_g)$ where $X = X_f \cup (X_g \setminus \{x\})$. Given an input stream w , if there exist sub-streams w_f, w_g such that $w = w_f w_g$ and such that both $\llbracket f \rrbracket(w_f)$ and $\llbracket g \rrbracket(w_g)$ are defined, then:

$$\begin{aligned} \llbracket \text{split}(f [x] g) \rrbracket(w, \bar{v}) &= \llbracket g \rrbracket(w_g, \bar{v}_g), \text{ where} \\ \bar{v}_g &= \bar{v}[x/int_f], \text{ and} \\ int_f &= \llbracket f \rrbracket(w_f, \bar{v}). \end{aligned}$$

Otherwise, $\llbracket \text{split}(f \langle x \rangle g) \rrbracket(w, \bar{v}) = \perp$.

Example 7. We promised earlier that for each cost domain operator op , the *split-op* combinator was a special case of the more general *split* combinator just defined. We illustrate this in the bank transaction example by constructing a QRE for *split-min(count, count)*, where *count* is the expression counting the number of transactions in a single month.

The desired function may be expressed by the QRE *split(count [p] minCount)*, where the QRE $minCount(p) = \min(p, count)$.³ The *split* combinator splits the

³ Note that this QRE is pedantically ill-formed, because the first argument to the min operator is a parameter p , and the second argument is a QRE *count*. The parameter p on the left should be read as the QRE $r_{month} ? p$, where $r_{month} = Dom(count)$.

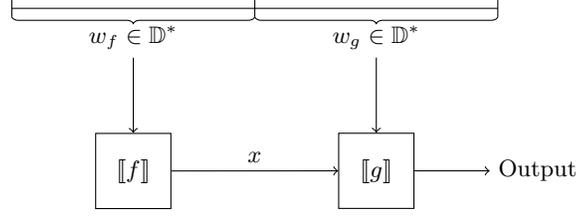


Fig. 2.2. The split combinator, $\mathit{split}(f \langle x \rangle g)$, divides the input data string w into two parts, $w = w_f w_g$, applies f to w_f and g to w_g , and substitutes the result of f into the parameter x of the term generated by g . The expression $\mathit{split}(f \langle x \rangle g)$ is similar except for the direction of the f - g data flow.

input data string into a prefix and a suffix, and propagates the output of count on the prefix into the parameter p of the QRE $\mathit{minCount}$ on the suffix.

In general, given appropriate QREs f and g , and a binary operator op , the expression $\mathit{split-op}(f, g)$ can be expressed as $\mathit{split}(f \langle p \rangle g')$, where $g' = op(\mathit{Dom}(f) ? p, g)$ and p is a new temporary parameter.

The definition of the left version of the split combinator, $\mathit{split}(f \langle x \rangle g)$ is similar to the definition of $\mathit{split}(f \langle x \rangle g)$, with just the direction of the f - g information flow reversed, and is given in the appendix.

Iteration. Our next combinator is iteration, the analog of Kleene-* from regular expressions. The definition of the combinator is similar to that of split , to permit more general forms of iteration than just those offered a hypothetical $\mathit{iter-op}$ operator.

Consider an expression $f : \text{QRE}(r, \{x\}, T)$, such that $\llbracket r \rrbracket$ is unambiguously iterable, and $x : T$. Then $\mathit{iter}[x](f)$ is an expression of the form $\text{QRE}(r^*, \{x\}, T)$. The expression $\mathit{iter}[x](f)$ divides the input stream w into a sequence of substreams, w_1, w_2, \dots, w_k , such that $\llbracket f \rrbracket(w_i)$ is defined for each i . $\llbracket \mathit{iter}[x](f) \rrbracket(w)$ is defined if this split exists and is unique. In that case, for each $i \in \{0, 1, 2, \dots, k\}$, define int_i as follows:

1. $\mathit{int}_0 = v_x$, and
2. for each $i \geq 1$, $\mathit{int}_i = \llbracket f \rrbracket(w_i, \{x \mapsto \mathit{int}_{i-1}\})$.

Finally, $\llbracket \mathit{iter}[x](f) \rrbracket(w, \bar{v}) = \mathit{int}_k$.

Example 8. In the bank transaction example, consider the query $\mathit{currBal}$ which maps the transaction history w of the customer to her current account balance. We can write $\mathit{currBal} = \mathit{iter-plus}(tx)$, where the expression $tx = (d \in \mathbb{R} ? d) \mathit{else} (d \notin \mathbb{R} ? 0)$ records the change in account balance as a result of a single transaction.

To desugar $\mathit{currBal}$ to use the parametrized version of the iteration combinator, we adopt a similar approach as in example 7. We first write the expression $tx' = \mathit{plus}(p, tx)$. Observe that tx' is a QRE dependent on a single parameter p ,

the previous account balance, and returns the new balance. Then, we can write: $currBal = iter-plus(tx) = iter[p](tx')$.

A similar approach works to desugar $iter-op(f)$, for any associative cost operator op with the identity element 0. We first write $f' = op(r_f ? p, f)$, for some new parameter p of type T_f . Then, observe that $iter-op(f) = (iter[p](f'))[p/0]$.

Remark 1. We have presented here a simplified version of the iteration combinator, which is sufficient for most examples. The full combinator, used in the proof of expressive completeness, allows f to depend on multiple parameters (instead of just a single parameter as above), and multiple values are computed in each substream w_i . This full combinator and the symmetric left-iteration combinator are defined in the full paper.

Streaming composition. In the bank transaction example, suppose we wish to compute the minimum balance over the entire account history. Recall that the stream w is a sequence of transactions, each data value indicating the deposit / withdrawal of some money from the account. The expression $currBal$ from example 8 maps the transaction history of the customer to her current balance. It is also simple to express the function $minValue = iter-min(d \in \mathbb{R} ? d)$ which returns the minimum of a stream of real numbers. We can then express the minimum account balance query using the streaming composition operator: $minBal = currBal \gg minValue$.

Informally, the streaming composition $currBal \gg minValue$ applies the expression $currBal$ to each prefix w_1, w_2, \dots, w_i of the input data stream $w = w_1, w_2, \dots$, thus producing the account balance after each transaction. It then produces an intermediate data stream w' by concatenating the results of $currBal$. This intermediate stream w' is supplied to the function $minValue$, which then produces the historical minimum account balance.

Formally, let f and g be of the form $QRE(r, \emptyset, T_f)$ and $QRE(T_f^*, X, T_g)$ respectively. Note that f produces results independent of any parameter, and that g is defined over all intermediate data streams $w' \in T_f^*$. Then $f \gg g : QRE(D^*, X, T_g)$. Given the input stream $w = w_1, w_2, \dots, w_k$ and parameter valuation \bar{v} , for each i such that $1 \leq i \leq k$, define int_i as follows:

1. If $\llbracket f \rrbracket(w_1, w_2, \dots, w_i, \bar{v}_\emptyset) \neq \perp$, then $int_i = \llbracket f \rrbracket(w_1, w_2, \dots, w_i, \bar{v}_\emptyset)$, where \bar{v}_\emptyset is the empty parameter valuation.
2. Otherwise, $int_i = \epsilon$.

Then $\llbracket f \gg g \rrbracket(w, \bar{v})$ is given by:

$$\llbracket f \gg g \rrbracket(w, \bar{v}) = \llbracket g \rrbracket(int_1, int_2, \dots, int_k, \bar{v}).$$

2.3 Examples

Analyzing a regional rail system. Consider the rail network of a small city. There are two lines, the airport line, al , between the city and the airport, and

the suburban line, sl , between the city and the suburbs. The managers of the network want to determine the average time for a traveler in the suburbs to get to the airport.

The data is a sequence of event tuples $(line, station, time)$, where $line \in \{al, sl\}$ indicates the railway line on which the event occurs, $station \in \{air, city, suburb\}$ indicates the station at which the train is stopped, and $time \in \mathbb{R}$ indicates the event time.

We first write the expression t_{sc} which calculates the time needed to travel from the suburbs to the city. Let $\varphi_{ss}(d) = (d.line = sl \wedge d.station = suburb)$ and $\varphi_{sc}(d) = (d.line = sl \wedge d.station = city)$ be the predicates indicating that the suburban line train is at the suburban station and at the city station respectively. Consider the expression:

$$t_{sc-drop} = \textit{split-plus}(\neg\varphi_{ss}^* \cdot \varphi_{ss} \cdot \neg\varphi_{sc}^* ? 0, \varphi_{sc} ? time)$$

that maps event streams of the form $\neg\varphi_{ss}^* \cdot \varphi_{ss} \cdot \neg\varphi_{sc}^* \cdot \varphi_{sc}$ to the time at which the train stops at the city station. The expression $t_{sc-pickup}$ that maps event streams to the pickup time from the suburban station can be similarly expressed:

$$t_{sc-pickup} = \textit{split-plus}(\neg\varphi_{ss}^* ? 0, \varphi_{ss} ? time, \neg\varphi_{sc}^* \cdot \varphi_{sc} ? 0).$$

Our goal is to express the commute time from the suburbs to the city. This is done by the expression: $t_{sc} = \textit{minus}(t_{sc-drop}, t_{sc-pickup})$.

The QRE t_{ca} that expresses the commute time from the city to the airport can be similarly constructed. We want to talk about the total travel time from the suburbs to the airport: $t_{sa} = \textit{split-plus}(t_{sc}, t_{ca})$. Our ultimate goal is the average travel time from the suburbs to the airport. The following QRE solves our problem: $t_{avg} = \textit{iter-avg}(t_{sa})$.

Rolling data telephone plans. Now consider a simple telephone plan, where the customer pays 50 dollars each month for a 5 GB monthly download limit. If the customer uses more than this amount, she pays 30 dollars extra, but otherwise, the unused quota is added to her next month's limit, up to a maximum of 20 GB.

The data domain $D_{tel} = \mathbb{R} \cup \{end_m\}$, where $d \in \mathbb{R}$ indicates a download of d gigabytes and end_m indicates the end of the billing cycle and payment of the telephone bill. Given the entire browsing history of the customer, we wish to compute the download limit for the current month.

First, the QRE $totalDown = \textit{iter-plus}(d \in \mathbb{R} ? d)$ takes a sequence of downloads and returns the total data downloaded. The browsing history of a single month is given by the regular expression $r_m = \mathbb{R}^* \cdot end_m$, and the QRE $monthDown = \textit{split-plus}(totalDown, end_m ? 0)$ maps data streams $w \in \llbracket r_m \rrbracket$ to the total quantity of data downloaded.

The following expression gives the unused allowance available at the end of the month, in terms of $initLim$, the download limit at the beginning of the month: $unused(initLim) = (r_m ? initLim) - monthDown$, and the QRE

$rollover(initLim) = \min(\max(used(initLim), 0), 20)$ gives the amount to be added to the next month's limit.

The QRE $nextQuota(initLim) = rollover(initLim) + 5$ provides the download limit for the next month in terms of $initLim$ and the current month's browsing history. Finally, the QRE $quota = iter[initLim](nextQuota)[initLim/5]$ maps the entire browsing history of the customer to the download limit of the current month.

Aggregating weather reports. Our final example deals with a stream of weather reports. Let the data domain $D_{wth} = \mathbb{R} \cup \{autEqx, sprEqx, newYear, \dots\}$, where the symbols $autEqx$ and $sprEqx$ represent the autumn and spring equinoxes. Here a number $d \in \mathbb{R}$ indicates a temperature reading of $d^\circ\text{C}$. We wish to compute the average winter-time temperature reading. For this query, let winter be defined as the time between an autumn equinox and the subsequent spring equinox.

The regular expression $r_{summ} = \mathbb{R}^* \cdot autEqx$ captures a sequence of temperature readings made before the start of winter, and the expression $summer = r_{summ} ? \emptyset$ maps the summer readings to the empty set. The QRE $collect = iter\text{-}union(d \in \mathbb{R} ? \{d\})$ collects a sequence of temperature readings into a set and so we can write QRE $winter = split\text{-}union(collect, (sprEqx ? \emptyset))$.

The QRE $year = split\text{-}union(summer, winter)$ constructs the set of winter-time temperatures seen in a data stream. The average winter-time temperature over all years is then given by: $avgWinter = avg(iter\text{-}union(year))$.

3 Compiling QREs into Streaming Evaluation Algorithms

In this section, we show how to compile a QRE f into a streaming algorithm M_f that computes $\llbracket f \rrbracket(w, \bar{v})$. Recall that the partial application $\llbracket f \rrbracket(w) : T_1 \times T_2 \times \dots \times T_k \rightarrow T$ is a term which maps parameter valuations \bar{v} to cost values c . Therefore, the complexity of expression evaluation depends on the complexity of performing operations on terms. This in turn depends on the choice of cost types \mathcal{T} , cost operations \mathcal{G} , and on the number of parameters appearing in f . Let $time_{\varphi\text{-}eval}(f)$ be the maximum time needed to evaluate predicates appearing in f on data values, and let $time_\tau$ (resp. mem_τ) be the maximum time (resp. memory) needed to perform an operation $op \in \mathcal{G}$ on terms $\tau_1, \tau_2, \dots, \tau_k$. Then we have:

Theorem 2. *Every QRE f can be compiled into a streaming algorithm M_f which processes each data item in time $poly(|f|)time_\tau time_{\varphi\text{-}eval}(f)$ and which consumes $poly(|f|)mem_\tau$ memory.*

Since the size of all intermediate terms produced, $|\tau|$, is bounded by $|w|poly(|f|)$, this is also an upper bound on $time_\tau$ and mem_τ . However, depending on the cost types \mathcal{T} and the cost operators \mathcal{G} , they might be smaller. For instance, in the case of real-valued terms over $\mathcal{G} = \{*, +, \min, \max, \mathbf{ins}, \mathbf{avg}\}$, the term simplification procedure of subsection 3.2 guarantees that $time_\tau = poly(|f|)$ and $mem_\tau = O(|f|)$.

- Theorem 3.** 1. For every choice of cost types \mathcal{T} , and cost operations, \mathcal{G} , $time_\tau = |w|poly(|f|)$ and $mem_\tau = |w|poly(|f|)$.
2. If $\mathcal{T} = \{\mathbb{R}, \mathbb{M}\}$, where \mathbb{M} is the set of multisets of real numbers, and $\mathcal{G} = \{*, +, \min, \max, \mathbf{ins}, \mathbf{avg}\}$, then $time_\tau = poly(|f|)$ and $mem_\tau = poly(|f|)$, independent of the length of the data stream $|w|$.

3.1 Overview

We construct, by structural induction on the QRE f , an *evaluator* M_f that computes the function $\llbracket f \rrbracket$. Let us first assume that each element $w_i \in d$ of the data stream is annotated with its index i , so the input to the streaming evaluator is a sequence of pairs (i, w_i) . Now consider the evaluator $M_{split(f [x] g)}$ when processing the stream w , as shown in figure 3.1. $M_{split(f [x] g)}$ forwards each element of the stream to both sub-evaluators M_f and M_g . After reading the prefix w_{pre} , M_f reports that $\llbracket f \rrbracket(w_{pre})$ is defined, and returns the term produced. We are therefore now interested in evaluating g over the suffix beginning at the current position of the stream. Each evaluator therefore also accepts input signals of the form (START, i) , which indicates positions of the input stream from which to start processing the function. While evaluating $split(f [x] g)$, there may be multiple prefixes, such as w_{pre} and w'_{pre} , for which $\llbracket f \rrbracket$ is defined. M_g gets a start signal after each of these prefixes is read, and may therefore be simultaneously evaluating g over multiple suffixes of the data stream. We refer to the suffix beginning at each start signal as a *thread* of evaluation.

After reading the data stream, M_g may report a result, i.e. that $\llbracket g \rrbracket$ is defined on some thread. Recall the semantics of the split operator: $split(f [x] g)$ is now defined on the entire data stream, and the result is $\tau_f[x/\tau_g]$, where τ_f and τ_g are respectively the results of evaluating $\llbracket f \rrbracket$ and $\llbracket g \rrbracket$ on the appropriate substrings. The compound evaluator therefore needs to “remember” the result τ_f reported by M_f after processing w_{pre} —a key part of the complexity analysis involves bounding the amount of auxiliary state that needs to be maintained. Next, since M_g may be processing threads simultaneously, it needs to uniquely identify the thread which is currently returning a result. Result signals are therefore triples of the form (RESULT, i, τ) , indicating that the thread beginning at index i is currently returning the result τ . On receiving the result τ_g from M_g at the end of the input stream, the compound evaluator reconciles this with the result τ_f earlier obtained from T_f , and itself emits the result $\tau_f[x/\tau_g]$. The response time of each evaluator therefore depends on the time $time_\tau$ required to perform basic operations over terms.

Finally, let us consider the auxiliary state that $M_{split(f [x] g)}$ needs to maintain during processing. It maintains a table of results Th_g reported by M_f (the subscript g indicates that the thread is currently being processed by M_g). Each time M_f reports a result $(\text{RESULT}, i, \tau_f)$, it adds the triple (i, j, τ_f) to Th_g : this indicates that the thread of $M_{split(f [x] g)}$ beginning at index i received the result τ_f from M_f at index j . $M_{split(f [x] g)}$ then sends a start signal to M_g . If left unoptimized, the number of entries in Th_g is therefore the number of start signals sent to M_g , which is, in the worst case, $O(|w|)$.

The evaluator M_g therefore emits kill signals of the form (KILL, i) , indicating that the thread beginning at index i will not be producing any more results, and that parent evaluators may recycle auxiliary state as necessary. Note that a kill signal (KILL, i) is a strong prediction about the fate of a thread: it is the assertion that for all future inputs, $\llbracket g \rrbracket$ is undefined for the suffix of the data stream beginning at index i . Because of the unambiguity requirements on QREs, it follows that each function evaluator M_f will always have at most $O(|f|)$ active threads. Such claims can be immediately lifted to upper bounds on the response times and memory consumption of the evaluator, $\text{poly}(|f|)\text{time}_\tau$ and $\text{poly}(|f|)\text{mem}_\tau$.

In summary, the evaluator accepts two types of signals: start signals of the form (START, i) and data signals of the form (SYMBOL, i, d) . It emits two types of signals as output: result signals of the form (RESULT, i, τ) and kill signals of the form (KILL, i) . The input fed to the evaluator satisfies the guarantee that no two threads ever simultaneously produce a result (input validity). In return, the evaluator is guaranteed to report results correctly, and eagerly kill threads.

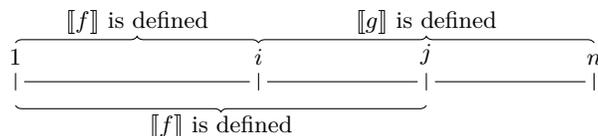


Fig. 3.1. Processing $\text{split}(f [x] g)$ over an input data stream w . There are two prefixes $w_{pre} = w_1 w_2 \dots w_i$ and $w'_{pre} = w_1 w_2 \dots w_j$ of w for which $\llbracket f \rrbracket$ is defined. $\llbracket g \rrbracket$ is defined for the suffix $w_{suff} = w_{i+1} w_{i+2} \dots w_n$.

The input-output requirements of function evaluators are formally stated in the full paper. The construction is similar to the streaming evaluation algorithm for DReX [6]. The major differences are the following:

1. QREs map data streams to terms, while DReX maps input strings to output strings. Even if the top-level QRE is parameter-free, sub-expressions might still involve terms, and intermediate results involve storing terms. We thus pay attention to the computational costs of manipulating terms: this includes the time time_τ to perform basic operations on them, and the memory mem_τ needed to store terms.
2. The *iter* combinator, as defined in this paper, is conceptually different from both the iteration and the chained sum combinators of DReX.

3.2 Succinct representation of terms

Recall that the evaluation time is parametrized by mem_τ and time_τ , the maximum memory and time required to perform operations on terms. It is therefore important to be able to succinctly represent terms. To prove the second

part of theorem 3, we now present a simplification procedure for terms over $\mathcal{G} = \{*, +, \min, \max, \text{ins}, \text{avg}\}$ so that mem_τ and $time_\tau$ are both bounded by $poly(|f|)$, independent of the length of the input stream. The representation we develop must support the following operations:

1. Construct the term $op(\tau_1, \tau_2, \dots, \tau_k)$, for each operator $op \in \mathcal{G}$, and appropriately typed terms $\tau_1, \tau_2, \dots, \tau_k$,
2. given terms τ_1 and τ_2 , and an appropriately typed $x \in Param(\tau_1)$, construct the term $\tau_1[x/\tau_2]$, and
3. given a term τ and a parameter valuation \bar{v} , compute $\llbracket \tau \rrbracket(\bar{v})$.

Intuitively, the simplification procedure *simpl* compiles an arbitrary copyless input term τ into an equivalent term $simpl(\tau)$ of bounded size. Consider, for example, a “large” term such as $\tau(x) = \min(\min(x, 3) + 2, 9)$. By routine algebraic laws such as distributivity, we have $\tau(x) = \min(\min(x + 2, 3 + 2), 9) = \min(x + 2, 5, 9) = \min(x + 2, 5)$.

Proposition 4. *For each input τ , *simpl* runs in time $O(|\tau|^2)$ and returns an equivalent term of bounded size: $simpl(\tau)$ is equivalent to τ , and $|simpl(\tau)| = O(|Param(\tau)|)$.*

We apply *simpl* to every intermediate term produced by the streaming algorithm. It follows that $mem_\tau = O(|X_f|) = O(|f|)$, where X_f is the set of parameters appearing in the description of f , and $time_\tau = O(poly(|X_f|)) = O(poly(|f|))$, and this establishes theorem 3.

A first attempt at designing *simpl* might be to repeatedly apply algebraic laws until terms reach a normal form. For example, the term $\min(x + 2, 5) + \min(y + 8, 7)$ could be mechanically simplified into the equivalent term $\min(12, x + 9, y + 13, x + y + 10)$. Notice however, that this simplification potentially involves a constant for each *subset* of parameters, thus producing terms of size $O(2^{|f|})$ (but still independent of the stream length $|w|$).

The simplification routine instead only propagates constants and does not attempt to completely reduce the term to a normal form. The term $\min(x + 2, 5) + \min(y + 8, 7)$ would therefore be left unchanged.

Consider the set of elements $A = \{3, 4, x, y + 3\}$. The only operation over multisets is insertion and average. A is therefore represented by the pair $(x + y + 10, 4)$, indicating the sum of the elements in A and the number of elements in A respectively. The average $\text{avg}(A)$ would be represented by the term $(x + y + 10)/4 = x/4 + y/4 + 2.5$. The simplified term may therefore also contain division terms of the form x/n and is drawn from the following grammar:

$$\begin{aligned} \tau_{\mathbb{R}} ::= & c \text{ (for } c \in \mathbb{R}) \mid x \mid x/n \text{ (for } n \in \mathbb{N}) \\ & \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 - \tau_2 \\ & \mid \min(\tau_1, \tau_2) \mid \max(\tau_1, \tau_2) \\ \tau_{\mathbb{M}} ::= & (\tau_{\mathbb{R}}, n) \end{aligned}$$

The procedure $simpl(\tau)$ works as follows:

1. If τ is a constant c or a parameter x , then return τ .
2. Otherwise, if $\tau = op(\tau_1, \tau_2)$, then compute $\tau'_1 = simpl(\tau_1)$ and $\tau'_2 = simpl(\tau_2)$. Output $prop-const(op(\tau'_1, \tau'_2))$.

The procedure $prop-const(\tau)$ performs constant propagation, and is essentially a large case-analysis with various pre-applied algebraic simplification laws. For example,

$$\begin{aligned}
 &\text{if } \tau = \tau_1 + \tau_2, \\
 &\quad \tau_1 = c_1, \text{ for some constant } c_1, \text{ and} \\
 &\quad \tau_2 = \min(\tau'_2, \tau''_2), \\
 &\text{then } prop-const(\tau) = \min(simpl(c_1 + \tau'_2, c_1 + \tau''_2)).
 \end{aligned}$$

The full case-analysis is described in the full paper.

The key observation is that such constant propagation is sufficient to guarantee small terms: if all non-trivial sub-terms contain at least one parameter, and each parameter appears at most once in each term, then there are at most a bounded number of constants in the simplified term, and thus a bounded number of leaves in the term-tree, and the term is therefore itself of a bounded size. Proposition 4 follows.

3.3 Why the unambiguity and single-use restrictions?

We first consider the unambiguity rules we used while defining QREs. While defining $op(f_1, f_2)$, we require the sub-expressions to have equal domains. Otherwise, the compound expression is only defined for data streams in the intersection of $Dom(f_1)$ and $Dom(f_2)$. Efficiently determining whether strings match regular expressions with intersection is an open problem (see [32] for the state of the art).

Now consider $split(f [x] g)$ without the unambiguous concatenability requirement. For a stream w with two splits $w = w_1w_2 = w'_1w'_2$, such that all of $\llbracket f \rrbracket(w_1)$, $\llbracket f \rrbracket(w'_1)$, $\llbracket g \rrbracket(w_2)$ and $\llbracket g \rrbracket(w'_2)$ are defined. Since we are defining functions and not relations, the natural choice is to leave $split(f [x] g)$ undefined for w . In the compound evaluator, we can no longer assume input validity, as two threads of T_g will report a result after reading w . Furthermore, the compound evaluator has to perform non-trivial bookkeeping and not report *any* result after reading w . Requiring unambiguous concatenability is a convenient way to avoid these issues.

We conjecture that non-regular functions are expressible if the single-use restrictions are relaxed. Furthermore, lifting the single-use restrictions makes the term simplification procedure of subsection 3.2 more complicated. Consider the “copyful” term $\tau = \min((x + y), (x + y) + z)$. Observe that terms are now best represented as DAGs, and consider applying constant propagation to the expression $\tau + 3$: this results in the term $\min(x + (y + 3), (x + y) + z + 3)$, thus causing the shared node $x + y$ of the term DAG to be split. It is not clear that the constant propagation procedure $prop-const$ does not cause a large blow-up in the size of the term being represented, because of reduced sharing.

4 The Expressiveness of Quantitative Regular Expressions

We now study the expressive power of QREs. The recently introduced formalism of *regular functions* [7] is parametrized by an arbitrary set of cost types and operations over cost values. Regular functions can be equivalently expressed both by the operational model of *streaming string-to-term transducers (SSTTs)*, and as logical formulas mapping strings to terms in monadic second-order (MSO) logic. In this section, we show that QREs are expressively equivalent to the streaming composition of regular functions. This mirrors similar results from classical language theory, where regular languages can be alternately expressed by finite automata, by regular expressions and as formulas in MSO.

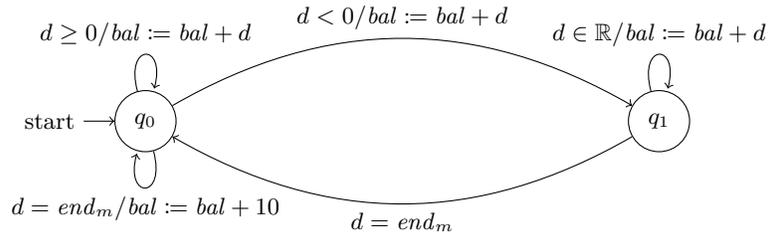


Fig. 4.1. The bank gives the customer a \$10 reward for each month in which no withdrawal is made. The current account balance is computed by the SSTT S_{bal} . The machine maintains a single register bal , the state q_0 indicates that no withdrawal has been made in the current month, and the machine moves to q_1 if at least one withdrawal has been made.

See figure 4.1 for an example of an SSTT. Informally, an SSTT maintains a finite state control, and a finite set of typed registers. Each register holds a term τ of the appropriate type, and register contents are updated during each transition. The main restrictions are: (a) transitions depend only on the current state, and not on the contents of the registers, (b) register updates are *copyless*: the update $x := x + y$ is allowed, but not the update $x := y + y$, and (c) at each point, the term held by each register is itself single-use.

We first describe the translation from QREs to the streaming composition of SSTTs. The construction proceeds in two steps: we first rewrite the given QRE as a streaming composition of composition-free QREs (subsection 4.1), and then we translate each composition-free QRE into an equivalent SSTT (subsection 4.2). In subsection 4.3, we describe the proof of expressive completeness: each SSTT can also be expressed by an equivalent QRE. For SSTTs with multiple registers, data may flow between the registers in complicated ways. The main part of the SSTT-to-QRE translation procedure is analysis of these data flows, and the

imposition of a partial order among data flows so that an inductive construction may be performed. Definitions and omitted proofs may be found in the full paper.

4.1 A normal form for QREs

Our main goal is to convert each QRE into the streaming composition of SSTTs. We first rewrite the given QRE as the streaming composition of several QREs, each of which is itself composition-free (theorem 5). We use the term QRE_{\gg} to highlight that the QRE does not include occurrences of the streaming composition operator. In subsection 4.2 we translate each of these intermediate QRE_{\gg} -s into a single SSTT.

Theorem 5. *For each QRE e , there exists a k and QRE_{\gg} -s f_1, f_2, \dots, f_k , such that e is equivalent to $f_1 \gg f_2 \gg \dots \gg f_k$.*

The proof is by induction on the structure of e . The claim clearly holds if e is a basic expression or itself of the form $f \gg g$. We now handle each of the other cases in turn.

Consider the expression $e = op(f \gg g, h)$. The idea is to produce, after reading each element w_i of the input stream w , the pair $int_i = (\llbracket f \rrbracket(w_1, w_2, \dots, w_i), w_i)$. We can now apply g to the first element and h to the second element of each pair in the stream of intermediate values $int = int_1, int_2, \dots$, and use the cost operator op to produce the final result.

We are therefore interested in the operator *tuple*, which produces the output $(t_1, t_2) \in T_1 \times T_2$ for each pair of input values $t_1 \in T_1$ and $t_2 \in T_2$. Observe that g cannot be directly applied to elements of the intermediate data stream, because they are pairs, and only the first element of this pair is of interest to g . Instead, we write the expression *project*₁(g), which is identical to g except that each data predicate $\varphi(d)$ is replaced with $\varphi(d.first)$, and each atomic function $\lambda(d)$ is replaced with $\lambda(d.first)$. Similarly, *project*₂(h) is identical to h except that it looks at the second element in each input data pair.

Now, observe that the original expression $e = op(f \gg g, h)$ is equivalent to the expression *tuple*($f, last$) \gg $op(\text{project}_1(g), \text{project}_2(h))$, where the expression *last* simply outputs the last element of the input stream: $last = \text{split}(D^* [x] (true ? d))$. The remaining cases are presented in the full paper.

4.2 From QREs to SSTTs

Theorem 6. *Let f be a QRE_{\gg} over a data domain D . There exists an SSTT S_f such that $\llbracket f \rrbracket = \llbracket S_f \rrbracket$.*

The proof proceeds by structural induction on f . The challenging cases are those of *split* and *iter*. Consider the case $f = \text{split}(g [x] h)$. Just as in figure 3.1, it is not possible to determine, before seeing the entire stream w , where to split the stream into $w = w_{pre}w_{suff}$ such that both $\llbracket g \rrbracket(w_{pre})$ and $\llbracket h \rrbracket(w_{suff})$ are defined. However, it is known that SSTTs are closed under *regular lookahead*, a powerful primitive operation by which the automaton can make a transition

not just based on the next input symbol, but based on a regular property of the entire (as yet unseen) suffix. The detailed proof of theorem 6, including a formal definition of regular lookahead, may be found in the full paper.

4.3 From SSTTs to QREs

Theorem 7. *Let \mathcal{S} be an SSTT. There exists a $QRE_{\gg} f$ such that $\llbracket \mathcal{S} \rrbracket = \llbracket f \rrbracket$.*

Proof outline. The proof follows the idea in [33] for constructing a regular expression from a given DFA. Suppose the DFA has n states $\{q_1, q_2, \dots, q_n\}$. Let $R_{i,j}^k$ denote the set of input strings w that take the DFA from state q_i to state q_j without going through any intermediate state numbered higher than k (the origin and target states i and j can be of index greater than k).

The regular expressions corresponding to the sets $R_{i,j}^k$ can be defined inductively. The base case when $k = 0$ corresponds to a single transition. For the inductive definition we have $R_{i,j}^k = R_{i,k}^{k-1} (R_{k,k}^{k-1})^* R_{k,j}^{k-1} \cup R_{i,j}^{k-1}$. Given the accepting states are $F = \{f_1, f_2, \dots, f_\ell\}$ and the initial state is 1 then the regular expression $R_{1,f_1}^n + R_{1,f_2}^n + \dots + R_{1,f_\ell}^n$ recognizes the same language as the given DFA.

We note that this construction works with unambiguous regular expressions. That is, whenever R and R' above are combined using \cup their domain is disjoint, when they are concatenated they are unambiguously concatenable, and when R is iterated it is unambiguously iterable.

Generalizing the idea to SSTTs. Let \mathcal{R} be the set of regular expressions used in the construction above. To extend this idea to SSTTs we would like to create for every $R \in \mathcal{R}$ and every variable x of a given SSTT \mathcal{S} a quantitative regular expression $f_{[x,R]}$ such that for $w \in R$ the value of variable x when processing of w by \mathcal{S} terminates is given by $f_{[x,R]}(w)$. If the SSTT has m variables $\{x_1, \dots, x_m\}$ we can work with vectors of QREs $\mathbf{V}_R = (f_{[1,R]}, \dots, f_{[m,R]})$ where $f_{[i,R]}$ abbreviates $f_{[x_i,R]}$. In order to use the same inductive steps for building these vectors of QREs, given \mathbf{V}_R and $\mathbf{V}_{R'}$ we need to be able to calculate $\mathbf{V}_{R \cdot R'}$, \mathbf{V}_{R^*} and $\mathbf{V}_{R \cup R'}$ whenever R and R' are combined in a respective manner in the construction above. The major difficulty is calculating \mathbf{V}_{R^*} . This is since variables may flow to one another in complicated forms. For instance, on a transition where variable x is updated to $\max(x, y)$ and variable y is updated to $z + 1$, the QRE for iterating this transition, zero, one or two times looks very different. Fortunately, there is a finite number of such variations, i.e. we can define terms that should capture each variation so that repeating the loop more than m times will yield a previously encountered term. The situation, though, is more complicated since there may be several loops on the same state, and in each such loop the variables flow may be different, and we will need to account for all loops consisting of arbitrary long concatenations of these paths. Thus, we will consider regular expressions of the form $R[i, j, k, \theta]$ where θ describes the flow of variables as defined below, and we will need to find some partial order on such regular expressions so that we can

always compute $\mathbf{V}_{R''}$ using previously computed \mathbf{V}_R and $\mathbf{V}_{R'}$, and whenever these are combined they do not violate the unambiguity requirements.

Variable flows. The *variable-flow* (in short *flow*) of a stream $w \in R_{i,j}^k$ is a function $\theta : [1..m] \rightarrow [1..m]$ such that $\theta(i) = j$ if x_j depends on x_i . By the copyless restriction there is at most one variable x_j which may depend on x_i and we assume without loss of generality that every variable flows into some other variable. We use Θ to denote the set of all flows. For $\theta_1, \theta_2 \in \Theta$ we use $\theta_1 \cdot \theta_2$ (or simply $\theta_1\theta_2$) to denote the flow function $\theta(i) = \theta_2(\theta_1(i))$. Note that if $w = w_1w_2$ and w_1, w_2 have flows θ_1, θ_2 respectively, then w has flow $\theta_1\theta_2$. We use θ^0 to denote the identity flow, θ^1 to denote θ and for $k > 1$ we use θ^k to denote $\theta \cdot \theta^{k-1}$. We say that a flow θ is *idempotent* if $\theta^k = \theta$ for every $k \in \mathbb{N}$. We use Θ_I to denote the set of all idempotent flows. We say that a flow θ is *normal* if $\theta(i) = j$ implies $i \geq j$ (i.e. the variables flow only upwards). Given an SSTT \mathcal{S} with set of states Q we can transform it to an SSTT \mathcal{S}_N where all updates on the edges are normal by letting the states of \mathcal{S}_N be $Q \times P_n$ where P_n are all the permutations of $[1..n]$. The SSTT \mathcal{S}_N remembers in the state the permutation needed to convert the flow to that of the original \mathcal{S} . Let Θ_N be the set of normal flows. It can be shown that concatenations of normal flows is a normal flow. Thus, we consider henceforth only normal flows.

For $\theta \in \Theta_N$ and $i, j, k \in [1..n]$, let $R[i, j, k, \theta]$ denote the set of all streams w such that when \mathcal{S} processes w starting in state i , it reaches state j without passing through any state indexed greater than k and the overall flow is θ . We use \mathcal{R}^Θ to denote the set of all such regular expressions. We are now ready to define the vectors \mathbf{V}_R corresponding to $R \in \mathcal{R}^\Theta$ for R 's that appear in the inductive construction.

5 Related Work

Data Management Systems. Traditional database management systems focus on efficient processing of queries over static data. When data is updated frequently, and queries need to be answered incrementally with each update, ideally without reprocessing the entire data set (due to its large size), the resulting data management problem of *continuous queries* has been studied in the database literature (see [9] for a survey, [1] for an example system, and [8] for CQL, an extension of the standard relational query language SQL for continuous queries). The literature on continuous queries assumes a more general data model compared to ours: there can be multiple data streams that encode relational data and queries involve, in addition to aggregation statistics, classical relational operators such as join. Solutions involve maintaining a window of the stream in memory and answer queries only approximately. A recent project on continuous queries is ActiveSheets [35] that integrates stream processing in spreadsheets and provides many high-level features aimed at helping end-users. In recent years, there is also increased focus on evaluating queries over data streams in a distributed manner, and systems such as Apache Storm (see <http://storm.apache.org>) and Twitter

Heron [27] facilitate the design of distributed algorithms for query evaluation. There is also extensive literature on querying XML data using languages such as XPath and XQuery and their extensions [15,26,28].

Many of these query languages support *filtering* operation that maps an input data stream to an output stream that can be fed as an input to another query. The streaming composition operation in QREs is inspired by this. The novelty in our work lies in the regular constructs for modular specification of numerical queries by exploiting the structure in the sequence.

Streaming Algorithms. Designing efficient streaming algorithms has been an active area of research in theoretical computer science (see [29,2] for illustrative results and [30] for a comprehensive survey). Such algorithms are designed for specific computational problems (for example, finding the k -median) using tools such as approximation and randomization. While we have considered only simple aggregation operators such as sum and averaging which have obvious streaming algorithms for exact computation, the complexity in QRE queries is due to the nesting of regular constructs and aggregation operations. Since our evaluation algorithm is oblivious to the set of cost combinators and the data structure used to summarize terms to be able to compute desired aggregates, our results are orthogonal and complementary to the literature on streaming algorithms.

String Transformations. Domain-specific languages for string manipulation such as sed, AWK, and Perl are widely used to query and reformat text files. However, these languages are Turing complete and thus do not support any algorithmic analysis. In recent years, motivated by applications to verification of string sanitizers and string encoders, there is a renewed interest in designing languages based on automata and transducers [3,24,17,19]. While such languages limit expressiveness, they have appealing theoretical properties such as closure under composition and decidable test for functional equivalence, that have been shown to be useful in practical applications. Symbolic automata and transducers introduced the idea of using unary predicates from a decidable theory, supported by modern SMT solvers [20], to process strings over unbounded or large alphabets [36], and we use the same idea for defining symbolic regular expressions. Furthermore, checking typing rules regarding the domains of QREs relies on the constructions on symbolic automata.

The work most relevant to this paper is the design of the language DReX, a declarative language that can express all regular string-to-string transformations [6,7]. In DReX, the only cost type is strings and the only operation is string concatenation, and thus, quantitative regular expressions can be viewed as a generalization of DReX. In the design of QREs, the key new insights are: (a) the introduction of parameters to pass values across chunks during iteration, (b) the clear separation between the regular constructs and cost combinators, and (c) the inclusion of the streaming composition operation. The compilation of QREs into a single-pass streaming algorithm generalizes the evaluation algorithm for DReX,

and its analysis now needs to be parametrized by the design of efficient data structures for representing terms for different choices of cost combinators.

The single-use restriction is common in theory of transducers to ensure that the output grows only linearly with the input [23,22]. Parameters in QREs are conceptually similar to attribute grammars in which attributes are used to pass information across non-terminals during parsing of programs [31]. Our goal is quite different, namely, specification of space-efficient streaming algorithms resulting in different design choices: rules are regular (and not context-free), there are no tests on attributes, and even though the splitting of the stream using *split* and *iter* imparts a hierarchical structure to the input stream as parse-trees do, in the QRE $op(f, g)$, the computation of f and g can impart *different* hierarchical structures to the same input stream.

Quantitative Analysis. The notion of *regularity* for mapping strings to cost values is introduced in [5] using the model of *cost register automata* and shown to coincide with MSO-definable graph transformations [16] using the theory of tree transducers [22,4]. Section 4 shows that the expressiveness of QREs without the streaming composition operator coincides with this class using the model of streaming string-to-tree transducers. Note that in this model, the control-flow does not involve any tests on the registers, but cost values can be combined by arbitrary operations. In contrast, models such as register machines and data automata allow tests, but analyzability typically limits the set of arithmetic operations allowed on data values (for example, in data languages, only equality over data values is allowed) [25,10,11].

There is a growing literature in formal methods on extending algorithms for verification and synthesis of finite-state systems from temporal correctness requirements to quantitative properties [14,21]. Typically, the system is modeled as a state-transition graph with costs associated with transitions, the cost of an execution is an aggregation of costs of transitions it contains (for example, maximum or limit-average for an infinite execution), and the analysis problem corresponds to checking that the minimum or maximum of the costs of all executions does not exceed a threshold. In contrast, we are analyzing a single execution, but QREs are significantly more expressive than the properties considered in quantitative verification literature.

6 Conclusion

Contributions. In summary, we have argued that suitably generalized versions of the classical regular operators, in conjunction with arithmetic operators for combining costs, and the streaming composition operation, provide an appealing foundation for specifying quantitative properties of data streams in a modular fashion. This paper makes the following contributions:

1. The idea of **regular programming** allows the programmer to specify the processing of a data stream in a modular way by considering different cases

and breaking up the stream into substreams using the constructs of *else*, *split*, and *iter*.

2. The language of **quantitative regular expressions** integrates regular constructs and the streaming composition operation, with cost types and combinator in a generic manner, and with a set of typing rules designed to achieve a trade-off between expressiveness and efficiency of evaluation and analysis.
3. The compilation of quantitative regular expressions into an efficient streaming implementation, and in particular, an **incremental, constant-space, linear-time, single pass streaming algorithm** for QREs with the numerical operators of sum, difference, minimum, maximum, and averaging.
4. **Expressiveness results** establishing the relationship between QREs and the class of regular functions defined using the machine model of cost register automata and MSO-definable string-to-tree transformations.

Future Work. We are currently working on an implementation of the proposed language for specifying flow-based routing policies in network switches. We also want to explore applications to the quantitative monitoring of executions of cyber-physical systems, and in particular, for analyzing simulations of hybrid systems models for robustness measures [37]. In terms of design of space-efficient data structures, in this paper we have considered only the operations of sum, difference, minimum, maximum, and averaging, for which each term can be summarized in constant space. In future work, we want to consider more challenging aggregate operators such as medians and frequency moments, for which sub-linear-space streaming algorithms are possible only if answers can be approximate.

References

1. D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.
2. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the 28th Annual Symposium on Theory of Computing*, STOC '96, pages 20–29. ACM, 1996.
3. R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages*, POPL '11, pages 599–610. ACM, 2011.
4. R. Alur and L. D'Antoni. Streaming tree transducers. In A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, editors, *Automata, Languages, and Programming*, volume 7392 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2012.
5. R. Alur, L. D'Antoni, J. Deshmukh, M. Raghothaman, and Y. Yuan. Regular functions and cost register automata. In *28th Annual Symposium on Logic in Computer Science*, pages 13–22, 2013.
6. R. Alur, L. D'Antoni, and M. Raghothaman. DRex: A declarative language for efficiently evaluating regular string transformations. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, POPL '15, pages 125–137. ACM, 2015.

7. R. Alur, A. Freilich, and M. Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the 23rd Annual Conference on Computer Science Logic and the 29th Annual Symposium on Logic in Computer Science*, CSL-LICS '14, pages 9:1–9:10. ACM, 2014.
8. A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In G. Lausen and D. Suciu, editors, *Database Programming Languages*, volume 2921 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2004.
9. S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, Sept. 2001.
10. H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411(4–5):702–715, 2010.
11. M. Bojańczyk, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data trees and XML reasoning. *Journal of the ACM*, 56(3):13:1–13:48, May 2009.
12. R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2):149–153, February 1971.
13. A. Brüggemann-Klein. Regular expressions into finite automata. In *LATIN '92*, volume 583 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 1992.
14. K. Chatterjee, L. Doyen, and T. Henzinger. Quantitative languages. *ACM Transactions on Computational Logic*, 11(4):23:1–23:38, July 2010.
15. Y. Chen, S. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE '06*. IEEE Computer Society, 2006.
16. B. Courcelle. Monadic second-order definable graph transductions: A survey. *Theoretical Computer Science*, 126(1):53–75, 1994.
17. L. D'Antoni and M. Veanes. Equivalence of extended symbolic finite transducers. In *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 624–639. Springer, 2013.
18. L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *Proceedings of the 41st Symposium on Principles of Programming Languages*, POPL '14, pages 541–553. ACM, 2014.
19. L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar. Fast: A transducer-based language for tree manipulation. In *Proceedings of the 35th Conference on Programming Language Design and Implementation*, PLDI '14, pages 384–394. ACM, 2014.
20. L. de Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*, 54(9):69–77, Sept. 2011.
21. M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer, 1st edition, 2009.
22. J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Information and Computation*, 154(1):34–91, 1999.
23. J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31(1):71–146, 1985.
24. P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11. USENIX Association, 2011.
25. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
26. C. Koch. XML stream processing. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 3634–3637. Springer, 2009.

27. S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250. ACM, 2015.
28. B. Mozafari, K. Zeng, and C. Zaniolo. High-performance complex event processing over XML streams. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 253–264. ACM, 2012.
29. I. Munro and M. Paterson. Selection and sorting with limited storage. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, pages 253–258. IEEE Computer Society, 1978.
30. S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.
31. J. Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.
32. G. Rosu. An effective algorithm for the membership problem for extended regular expressions. In *Foundations of Software Science and Computational Structures*, volume 4423 of *Lecture Notes in Computer Science*, pages 332–345. Springer, 2007.
33. M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012.
34. R. Stearns and H. Hunt. On the equivalence and containment problems for unambiguous regular expressions, grammars, and automata. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, pages 74–81. IEEE Computer Society, 1981.
35. M. Vaziri, O. Tardieu, R. Rabbah, P. Suter, and M. Hirzel. Stream processing with a spreadsheet. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2014.
36. M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual Symposium on Principles of Programming Languages*, pages 137–150. ACM, 2012.
37. A. Zutshi, S. Sankaranarayanan, J. Deshmukh, J. Kapinski, and X. Jin. Falsification of safety properties for closed loop control systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, pages 299–300. ACM, 2015.

A Summary of Quantitative Regular Expressions

This appendix summarizes the combinators used to construct QREs. Recall that each QRE f is associated with a regular expression r , describing the set $\llbracket r \rrbracket \subseteq D^*$ of data streams over which it is defined, the cost domain C , and a sequence of parameters $X = \langle x_1, x_2, \dots, x_k \rangle$ over which it is dependent. We say that f is of the form $\text{QRE}(r, X, C)$, or simply that $f : \text{QRE}(r, X, C)$.

A.1 Basic functions

$\varphi ? \lambda$. Defined if for some data domain D and cost domain C , $\varphi \in \Phi_D$ and $\lambda : D \rightarrow C$ is in \mathcal{G} . Then $\varphi ? \lambda : \text{QRE}(\varphi, \emptyset, C)$.

$$\llbracket \varphi ? \lambda \rrbracket(w, \bar{v}) = \begin{cases} \lambda(w) & \text{if } |w| = 1 \text{ and } \varphi(w) \text{ is true, and} \\ \perp & \text{otherwise.} \end{cases}$$

$\epsilon ? x$. Defined for each parameter $x \in X$, data domain D and cost domain C , if x is of type C . Then $\epsilon ? x : \text{QRE}(\epsilon, \{x\}, C)$.

$$\llbracket \epsilon ? x \rrbracket(w, \bar{v}) = \begin{cases} v_x & \text{if } w = \epsilon, \text{ and} \\ \perp & \text{otherwise.} \end{cases}$$

Remark 2. We will often write QREs of the form $r ? \tau$, which map data streams matching the regular expression r to some output term τ . This is just syntactic sugar and can be expressed using the remaining combinators.

A.2 Cost operations

$op(\mathbf{f}_1, \mathbf{f}_2, \dots, \mathbf{f}_k)$. Defined for each operator $op : T_1 \times T_2 \times \dots \times T_k \rightarrow T$ in \mathcal{G} , when $k > 0$, and if:

1. for each i , $f_i : \text{QRE}(r_i, X_i, T_i)$,
2. all the domains are equal: for all i, j , $\llbracket r_i \rrbracket = \llbracket r_j \rrbracket$, and
3. the parameters are pairwise-disjoint: for all $i \neq j$, $X_i \cap X_j = \emptyset$.

$op(f_1, f_2, \dots, f_k) : \text{QRE}(r_1, \bigcup_i X_i, T)$. For each w, \bar{v} , if $\llbracket f_i \rrbracket(w, \bar{v}) \neq \perp$ for all i , then:

$$\llbracket op(f_1, f_2, \dots, f_k) \rrbracket(w, \bar{v}) = op(int_1, int_2, \dots, int_k), \text{ where} \\ \text{for each } i, int_i = \llbracket f_i \rrbracket(w, \bar{v}).$$

Otherwise, $\llbracket op(f_1, f_2, \dots, f_k) \rrbracket(w, \bar{v}) = \perp$.

Substitution, $f[x/g]$. Defined if $f : \text{QRE}(r_f, X_f, T_f)$, $g : \text{QRE}(r_g, X_g, T_g)$, $x : T_g$, $x \in X_f$ and if:

1. the domains are equal: $\llbracket r_f \rrbracket = \llbracket r_g \rrbracket$, and
2. $(X_f \setminus \{x\}) \cap X_g = \emptyset$.

Then $f[x/g] : \text{QRE}(r_f, (X_f \setminus \{x\}) \cup X_g, T_f)$. For each w, \bar{v} , if $\llbracket g \rrbracket(w) \neq \perp$, then:

$$\llbracket f[x/g] \rrbracket(w, \bar{v}) = \llbracket f \rrbracket(w, \bar{v}[x/\llbracket g \rrbracket(w, \bar{v})]).$$

Otherwise, $\llbracket f[x/g] \rrbracket(w, \bar{v}) = \perp$. See figure A.1.

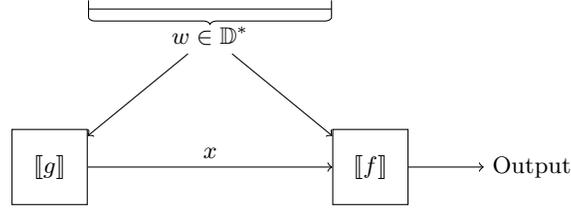


Fig. A.1. The “substitute” combinator $f[x/g]$ substitutes the result of g into the parameter x while evaluating f .

A.3 Regular combinators

Choice, $f \text{ else } g$. Defined if for some data domain D and cost domain C ,

1. r_f and r_g are regular expressions over D ,
2. $f : \text{QRE}(r_f, X_f, C)$ and $g : \text{QRE}(r_g, X_g, C)$, and
3. $\llbracket r_f \rrbracket \cap \llbracket r_g \rrbracket = \emptyset$.

Then, $f \text{ else } g : \text{QRE}(r_f + r_g, X_f \cup X_g, C)$. For each w, v , if $\text{int}_f = \llbracket f \rrbracket(w, \bar{v})$ and $\text{int}_g = \llbracket g \rrbracket(w, \bar{v})$, then:

$$\llbracket f \text{ else } g \rrbracket(w, \bar{v}) = \begin{cases} \text{int}_f & \text{if } \text{int}_f \neq \perp, \text{ and} \\ \text{int}_g & \text{otherwise.} \end{cases}$$

(Right) Concatenation $\text{split}(f [x] g)$. $\text{split}(f [x] g)$ is defined if $f : \text{QRE}(r_f, X_f, T_f)$, $g : \text{QRE}(r_g, X_g, T_g)$, and

1. $\llbracket r_f \rrbracket$ and $\llbracket r_g \rrbracket$ are unambiguously concatenable,
2. $X_f \cap (X_g \setminus \{x\}) = \emptyset$, and
3. $x \in X_g$ is of type T_f .

Then, $\mathit{split}(f \langle x \rangle g) : \text{QRE}(r_f \cdot r_g, X, T_g)$, where $X = X_f \cup (X_g \setminus \{x\})$. Given a string w and parameter valuation \bar{v} , if there exist unique streams w_f, w_g such that:

1. $w = w_f w_g$,
2. $\mathit{int}_f = \llbracket f \rrbracket(w_f, \bar{v}) \neq \perp$, and
3. $\mathit{int}_g = \llbracket g \rrbracket(w_g, \bar{v}[x/\mathit{int}_f]) \neq \perp$,

then $\llbracket \mathit{split}(f \langle x \rangle g) \rrbracket(w, \bar{v}) = \mathit{int}_g$. Otherwise, $\llbracket \mathit{split}(f \langle x \rangle g) \rrbracket(w, \bar{v}) = \perp$.

Left concatenation $\mathit{split}(f \langle x \rangle g)$. $\mathit{split}(f \langle x \rangle g)$ is defined if $f : \text{QRE}(r_f, X_f, T_f)$, $g : \text{QRE}(r_g, X_g, T_g)$, and

1. $\llbracket r_f \rrbracket$ and $\llbracket r_g \rrbracket$ are unambiguously concatenable,
2. $(X_f \setminus \{x\}) \cap X_g = \emptyset$, and
3. $x \in X_f$ is of type T_g .

Then, $\mathit{split}(f \langle x \rangle g) : \text{QRE}(r_f \cdot r_g, X, T_f)$, where $X = (X_f \setminus \{x\}) \cup X_g$. Given a string w and parameter valuation \bar{v} , if there exist unique streams w_f, w_g such that:

1. $w = w_f w_g$,
2. $\mathit{int}_g = \llbracket g \rrbracket(w_g, \bar{v}) \neq \perp$, and
3. $\mathit{int}_f = \llbracket f \rrbracket(w_f, \bar{v}[x/\mathit{int}_g]) \neq \perp$,

then $\llbracket \mathit{split}(f \langle x \rangle g) \rrbracket(w, \bar{v}) = \mathit{int}_f$. Otherwise, $\llbracket \mathit{split}(f \langle x \rangle g) \rrbracket(w, \bar{v}) = \perp$.

(Right) Iteration, $\mathit{iter}[x](f[g_1, g_2, \dots, g_k])$. See figure A.2 for a visualization of the *iter* combinator.

Let $f : \text{QRE}(r_f, X, T)$, $x : T$, and for each i , $1 \leq i \leq k$, $g_i : \text{QRE}(r_i, \emptyset, T_i)$, such that:

1. the domains are identical: $\llbracket r_f \rrbracket = \llbracket r_1 \rrbracket = \llbracket r_2 \rrbracket = \dots = \llbracket r_k \rrbracket$,
2. $\llbracket r_f \rrbracket$ is unambiguously iterable,
3. $X = [x, y_1, y_2, \dots, y_k]$, and
4. for each i , y_i has the same type as the return type of g_i , i.e. $y_i : T_i$.

In this case, $\mathit{iter}[x](f[g_1, g_2, \dots, g_k])$ is defined, and is of the form $\text{QRE}(r_f^*, X, T)$.

Given an input stream w and a parameter valuation \bar{v} , if there exists a unique sequence of streams, $w = w_1, w_2, \dots, w_n$, such that $w_i \in \llbracket r_f \rrbracket$ for each i , then for each j , $0 \leq j \leq n$, we define the parameter valuation \bar{v}_j as follows:

1. $\bar{v}_0 = \bar{v}$, and
2. for each $j \geq 1$, define \bar{v}_j as:
 - (a) $v_j x = \llbracket f \rrbracket(w_j, \bar{v}_{j-1})$, and
 - (b) for all i , $v_{j y_i} = \llbracket g_i \rrbracket(w_j, \bar{v}_\emptyset)$, where \bar{v}_\emptyset is the empty parameter valuation.

Then, $\llbracket \mathit{iter}[x](f[g_1, g_2, \dots, g_k]) \rrbracket(w, \bar{v}) = v_n x$. Otherwise, if no such split exists, then $\llbracket \mathit{iter}[x](f[g_1, g_2, \dots, g_k]) \rrbracket(w, \bar{v}) = \perp$.

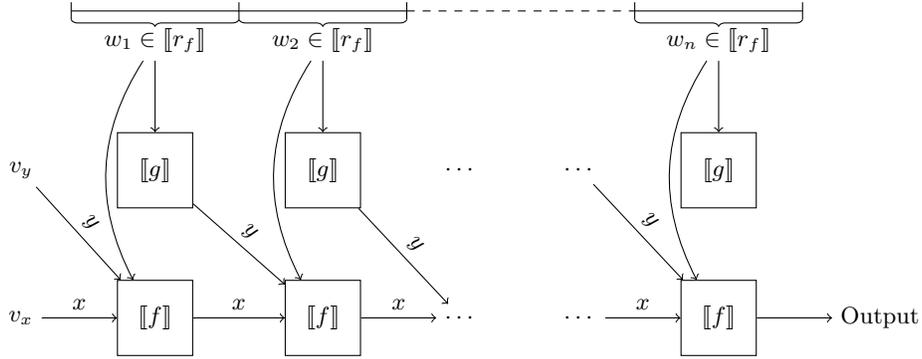


Fig. A.2. The working of the RCE $\text{iter}[x](f[g])$. The input string w is split into chunks, w_1, w_2, \dots, w_n , and the functions f and g are applied to each chunk. Note that apart from the first iteration, subsequent iterations are independent of the original parameter valuation (x, y) . The expression $\text{litersum}nxf g$ is similar, with just the direction of data flow reversed.

Left iteration, $\text{iter}(f)$. Let $f : \text{QRE}(r_f, X, T)$, $x : T$, and for each i , $1 \leq i \leq k$, $g_i : \text{QRE}(r_i, \emptyset, T_i)$, such that:

1. the domains are identical: $\llbracket r_f \rrbracket = \llbracket r_1 \rrbracket = \llbracket r_2 \rrbracket = \dots = \llbracket r_k \rrbracket$,
2. $\llbracket r_f \rrbracket$ is unambiguously iterable,
3. $X = [x, y_1, y_2, \dots, y_k]$, and
4. for each i , y_i has the same type as the return type of g_i , i.e. $y_i : T_i$.

In this case, $\text{iter}[x](f[g_1, g_2, \dots, g_k])$ is defined, and is of the form $\text{QRE}(r_f^*, X, T)$.

Given an input stream w and a parameter valuation \bar{v} , if there exists a unique sequence of streams, $w = w_1, w_2, \dots, w_n$, such that $w_i \in \llbracket r_f \rrbracket$ for each i , then for each j , $0 \leq j \leq n$, we define the parameter valuation \bar{v}_j as follows:

1. $\bar{v}_n = \bar{v}$, and
2. for each $j < n$, define \bar{v}_j as:
 - (a) $v_j x = \llbracket f \rrbracket(w_{j+1}, \bar{v}_{j+1})$, and
 - (b) for all i , $v_{j y_i} = \llbracket g_i \rrbracket(w_{j+1}, \bar{v}_\emptyset)$, where \bar{v}_\emptyset is the empty parameter valuation.

Then, $\llbracket \text{iter}[x](f[g_1, g_2, \dots, g_k]) \rrbracket(w, \bar{v}) = v_n x$. Otherwise, if no such split exists, then $\llbracket \text{iter}[x](f[g_1, g_2, \dots, g_k]) \rrbracket(w, \bar{v}) = \perp$.

A.4 Streaming composition, $f \gg g$

Defined if for some data domain D , cost domains T_f, T_g , and regular expression r over data streams in D^* , $f : \text{QRE}(r, \emptyset, T_f)$ and $g : \text{QRE}(T_f^*, X, T_g)$. $f \gg g : \text{QRE}(D^*, X, T_g)$. Given the input stream $w = w_1, w_2, \dots, w_k$ and parameter valuation \bar{v} , for each i such that $1 \leq i \leq k$, define int_i as follows:

1. If $\llbracket f \rrbracket(w_1, w_2, \dots, w_i, \bar{v}_\emptyset) \neq \perp$, then $int_i = \llbracket f \rrbracket(w_1, w_2, \dots, w_i, \bar{v}_\emptyset)$, where \bar{v}_\emptyset is the empty parameter valuation.
2. Otherwise, $int_i = \epsilon$.

Then $\llbracket f \gg g \rrbracket(w, \bar{v})$ is given by:

$$\llbracket f \gg g \rrbracket(w, \bar{v}) = \llbracket g \rrbracket(int_1, int_2, \dots, int_k, \bar{v}).$$

B Compiling QREs into Streaming Algorithms

B.1 The formal specification of an expression evaluator

As described in subsection 3.1, the input alphabet of function evaluators is $IN = (\{\text{START}\} \times \mathbb{N}) \cup (\{\text{SYMBOL}\} \times \mathbb{N} \times D)$. After reading each input symbol, the evaluator emits a finite set of responses drawn from $OUT = (\{\text{RESULT}\} \times \mathbb{N} \times \text{TERM-REP}) \cup (\{\text{KILL}\} \times \mathbb{N})$, where TERM-REP is the space of terms.

Given an evaluator input sequence $in = in_1 in_2 \dots in_n$, let $w_{in} \in D^*$ be the sequence of data symbols in in . We refer to every suffix $in_i in_{i+1} \dots in_n$ of in , where $in_i = (\text{START}, j)$ is a start signal, as a *thread* of in . Finally, *active threads* are those $(\text{START}, j)(\text{SYMBOL}, k, d_k) \dots$ for which the evaluator has not yet emitted a kill signal, (KILL, j) .

Example 9. For evaluation input sequence $in = (\text{SYMBOL}, 4, a), (\text{START}, 5), (\text{SYMBOL}, 5, b), (\text{START}, 6), (\text{SYMBOL}, 6, a)$, the data stream $w_{in} = aba$, and there are two threads: $(\text{START}, 5)(\text{SYMBOL}, 5, b), \dots$ and $(\text{START}, 6), (\text{SYMBOL}, 6, a)$.

Informally, an input sequence in is a valid evaluator input to a quantitative regular expression f if there is always at most one thread for which f is defined.

Definition 8 (Validity of evaluator inputs). Given a QRE f , an input sequence $in = in_1 in_2 \dots in_n \in IN^*$ is a *valid evaluator input* if: (a) the sequence of indices, $i_1 i_2 \dots i_n \in \mathbb{N}$ is monotonically increasing, (b) each pair of distinct data signals, (SYMBOL, i, d_i) and (SYMBOL, j, d_j) , have distinct indices $i \neq j$, and (c) for each prefix pre of in , there is at most one thread th_{pre} for which $\llbracket f \rrbracket$ is defined.

Example 10. Consider the QRE $f = D^* ? 0$, which maps every sequence of data values to the constant output 0. Notice that the input sequence from example 9, $in = (\text{SYMBOL}, 4, a)(\text{START}, 5)(\text{SYMBOL}, 5, b)(\text{START}, 6)(\text{SYMBOL}, 6, a)$, is *not* a valid evaluator input for f because f is defined both threads $(\text{START}, 5), \dots$ and $(\text{START}, 6)$. On the other hand, the prefix $(\text{SYMBOL}, 4, a)(\text{START}, 5)(\text{SYMBOL}, 5, b)$ is a valid evaluator input.

The following claim follows immediately from the definition of input validity:

Proposition 9. *There are no adjacent start signals (START, i) in every valid evaluator sequence in to a quantitative regular expression f .*

Definition 10. An evaluator M computes the QRE f if:

1. (Correctness of results). After reading every valid input sequence in , it emits a result (RESULT, i, t) exactly when there is a thread th such that $\llbracket f \rrbracket(w_{th}) = t$, and
2. (Eagerness of kills). After reading every data signal (SYMBOL, i, d) , it emits a kill signal (KILL, j) for all those active threads $th = (\text{START}, j) \cdots$ such that for all suffixes $w \in D^*$, $\llbracket f \rrbracket(w_{th}w)$ is undefined.

While constructing evaluators for the various combinators, we guarantee that: (a) the number of active threads is always at most a linear function of the expression size $|f|$, and (b) the evaluator constructed responds to each input symbol in time $O(\text{poly}(|f| \text{time}_\tau \text{time}_{\varphi\text{-eval}}(f)))$ and requires at most $O(\text{poly}(|f| \text{mem}_\tau))$ memory, where $\text{time}_{\varphi\text{-eval}}(f)$ is the maximum time required to evaluate the predicates appearing in f on data values.

B.2 Basic evaluators

Evaluators for $\varphi ? \lambda$. If the data predicate φ is satisfiable, then we construct the following evaluator $M_\varphi ? \lambda$. The evaluator maintains two sets $Th, Th' \subseteq \mathbb{N}$ of threads, both initialized to \emptyset . On receiving the signal (START, i) , the evaluator updates $Th := Th \cup \{i\}$, and does not emit any responses. On receiving the input signal (SYMBOL, i, d) :

1. For each thread $j \in Th'$, it emits the kill signal (KILL, j) .
2. If d satisfies φ , then:
 - (a) for each $j \in Th$, it emits the response $(\text{RESULT}, j, \lambda(d))$,
 - (b) updates $Th' := Th$, and then updates $Th := \emptyset$.
3. Otherwise, when d does not satisfy φ , it emits the kill signal (KILL, j) for each thread $j \in Th$, and then updates $Th := \emptyset$ and $Th' := \emptyset$.

The following observation follows from proposition 9. It follows that the evaluator constructed above responds to each input signal in constant time.

Proposition 11. *If φ is a satisfiable data predicate, then the evaluator $M_\varphi ? \lambda$ satisfies the following invariant while processing each valid input sequence: $|Th|, |Th'| \leq 1$.*

Observe that Th is the set of all those threads th such that $w_{th} = \epsilon$, and Th' is the set of all those threads th' such that $w_{th'} = d$ for some d which satisfies φ . We can therefore show that:

Lemma 12. *When φ is satisfiable, the evaluator $T_\varphi ? \lambda$ computes $\varphi ? \lambda$.*

Otherwise, in the case that φ is unsatisfiable, we construct the following evaluator M_\perp . On every start signal (START, i) , the evaluator responds with (KILL, i) . For every input symbol (SYMBOL, i, d) , the evaluator does not emit any responses.

Lemma 13. *The evaluator M_\perp correctly evaluates $\varphi ? \lambda$ for every unsatisfiable data predicate φ and atomic transformation λ .*

Evaluators $\epsilon ? x$. The evaluator $M_{\epsilon ? x}$ maintains a set $Th \subseteq \mathbb{N}$ of unkilld threads, initialized to \emptyset , and with the invariant that $|Th| \leq 1$. On receiving the signal (START, i) , it returns the result (RESULT, i, x) , and updates $Th := Th \cup \{i\}$. On receiving the signal (SYMBOL, i, d) , it emits a kill signal (KILL, j) for every thread index $j \in Th$ and then resets $Th := \emptyset$.

It is clear that the evaluator correctly reports results and eagerly kills threads. From the invariant $|Th| \leq 1$, it also follows that the evaluator processes every input signal in constant time and consumes at most a constant amount of memory.

B.3 State-free evaluators

Evaluators for $op(f_1, f_2, \dots, f_k)$ and $f[x/g]$. The evaluators for $op(f_1, f_2, \dots, f_k)$ and $f[x/g]$ are conceptually similar and both very simple. Start and symbol signals are sent directly to the sub-evaluators, and the compound evaluator itself maintains no auxiliary state. We therefore present them before constructing more complicated evaluators for the remaining combinators.

Given evaluators M_1, M_2, \dots, M_k for f_1, f_2, \dots, f_k respectively, and M_f and M_g for f and g , the evaluators $M_{op(f_1, f_2, \dots, f_k)}$ and $M_{f[x/g]}$ function as follows.

1. Each input signal, (START, i) or (SYMBOL, i, d) , is forwarded directly to each of the sub-evaluators.
2. On receiving a result, $(\text{RESULT}, j, \tau_1)$, from M_1 , (necessarily simultaneously with results $(\text{RESULT}, j, \tau_2)$, $(\text{RESULT}, j, \tau_3)$, \dots , $(\text{RESULT}, j, \tau_k)$ from M_2, M_3, \dots, M_k respectively) $M_{op(f_1, f_2, \dots, f_k)}$ emits the result $(\text{RESULT}, j, op(\tau_1, \tau_2, \dots, \tau_k))$. Similarly, on receiving the results $(\text{RESULT}, j, \tau_f)$ and $(\text{RESULT}, j, \tau_g)$ from M_f and M_g , $M_{f[x/g]}$ outputs $\tau_f[x/\tau_g]$.
3. When $T_{op(f_1, f_2, \dots, f_k)}$ receives a kill signal (KILL, j) from T_1 , it itself kills the thread beginning at index j by emitting (KILL, j) . Similarly, when $T_{f[x/g]}$ receives a kill signal (KILL, j) from T_f , it itself emits (KILL, j) . Kill signals from the remaining sub-evaluators have to arrive simultaneously with kills from T_1 and T_f because of equality of domains and eagerness of kills, and can therefore be ignored.

It is clear that $T_{op(f_1, f_2, \dots, f_k)}$ and $T_{f[x/g]}$ correctly evaluate the $op(f_1, f_2, \dots, f_k)$ and $f[x/g]$ respectively, and process each input signal in time $t_1 + t_2 + \dots + t_k + time_\tau$ and $t_f + t_g + time_\tau$ respectively.

Evaluators for $f \text{ else } g$. We now present evaluators for conditional choice. The evaluator $M_{f \text{ else } g}$ is nearly stateless: it only maintains the set of active threads of T_f and T_g so that kills may be issued at appropriate times.

Given a pair of QREs f and g with disjoint domains $Dom(f) \cap Dom(g) = \emptyset$, the evaluator $M_{f \text{ else } g}$ for $f \text{ else } g$ is constructed from M_f and M_g as follows. It maintains two sets $Th_f, Th_g \subseteq \mathbb{N}$ of active threads, both initialized to \emptyset . On receiving (START, i) , the evaluator updates $Th_f := Th_f \cup \{i\}$ and $Th_g := Th_g \cup \{i\}$

and sends the signal (START, i) to both M_f and M_g . On receiving (SYMBOL, i, d) , it forwards the signal to both M_f and M_g .

In every case, the responses from M_f and M_g are processed as follows:

1. If the response is a result, (RESULT, j, τ) , then $M_f \text{ else } g$ itself emits the result (RESULT, j, τ) .
2. If the response is a kill signal, (KILL, j) , then it updates $Th_f := Th_f \setminus \{j\}$ if the response came from M_f , and otherwise (in the case that it came from M_g), it updates $Th_g := Th_g \setminus \{j\}$. If $j \notin Th_f \cup Th_g$, then $M_f \text{ else } g$ emits the kill signal (KILL, j) .

Every valid input sequence in to $M_f \text{ else } g$ is also a valid input sequence to M_f and to M_g . Hence, M_f and M_g report results correctly. Therefore, every time $M_f \text{ else } g$ reports a result, it corresponds to an actual result of $f \text{ else } g$. The converse is also true: every result for $\llbracket f \text{ else } g \rrbracket$ corresponds to either $\llbracket f \rrbracket$ or $\llbracket g \rrbracket$ being defined, and from the induction hypothesis it follows that $M_f \text{ else } g$ emits the result.

Every element of Th_f (resp. Th_g) corresponds to an active thread of M_f (resp. M_g). It follows that $M_f \text{ else } g$ eagerly kills threads. Putting the observations together, we have:

Proposition 14. *The evaluator $M_f \text{ else } g$ computes $f \text{ else } g$.*

Also observe that $M_f \text{ else } g$ maintains at most $n_f + n_g$ active threads, where n_f, n_g are the number of active threads of M_f and M_g respectively, and responds to each input signal in time $t_f + t_g + O(n_f + n_g)$, where t_f, t_g are the response times of M_f and M_g respectively.

Evaluator for $f \gg g$. We now construct $M_f \gg g$ given evaluators M_f and M_g for the sub-expressions. The evaluator maintains a boolean variable $isStarted$, initialized to *false*, an index $lastIndex_g$, and a term τ_g to store the last result of M_g . Both $lastIndex_g$ and τ_g are initialized arbitrarily.

On receiving (START, i) , $M_f \gg g$:

1. updates $isStarted := true$, and
2. forwards the start signal to both sub-evaluators M_f and M_g .

On receiving an input symbol (SYMBOL, i, d) , the evaluator forwards it to M_f .

If M_f responds with a result, (RESULT, j, c) , then $M_f \gg g$ sends the signal (SYMBOL, i, c) to M_g . Otherwise, if M_f does not produce a result, and $isStarted$ is *true*, then it produces the output $(\text{RESULT}, lastIndex_g, \tau_g)$.

When M_g responds with a result, (RESULT, j, τ) , it first updates $lastIndex_g := j$ and $\tau_g := \tau$. Then it produces the output $(\text{RESULT}, lastIndex_g, \tau_g)$.

Kill signals from M_f are ignored, and we are guaranteed to never receive a kill signal from M_g .

Proposition 15. *$M_f \gg g$ computes $f \gg g$.*

B.4 Stateful evaluators: $\mathit{split}(f \langle x \rangle g)$ and $\mathit{split}(f \leftarrow^x g)$

We have already described the evaluator for $\mathit{split}(f \langle x \rangle g)$ in the motivation. Formally, the evaluator maintains two sets $Th_f \subseteq \mathbb{N}$ of threads which are being processed by M_f and $Th_g \subseteq \mathbb{N} \times \mathbb{N} \times \text{TERM-REP}$ of threads being processed by M_g , where TERM-REP is the space of terms. Both sets are initialized to \emptyset . We now describe the evaluator behavior.

On receiving a start signal, (START, i) , the compound evaluator updates $Th_f := Th_f \cup \{i\}$ and sends (START, i) to M_f . On receiving a data signal, (SYMBOL, i, d) , the compound evaluator repeats the same signal to both sub-evaluators.

On receiving a result $(\text{RESULT}, j, \tau_f)$ from M_f , the compound evaluator updates $Th_g := Th_g \cup \{(j, i, \tau_f)\}$ and sends the signal (START, i) to M_g . On receiving a kill signal (KILL, j) from M_f , it updates $Th_f := Th_f \setminus \{j\}$. If there is no element of the form $(j, j', \tau_f) \in Th_f$, for $j' \in \mathbb{N}$, then the compound evaluator emits a kill signal (KILL, j) .

On receiving a result $(\text{RESULT}, j, \tau_g)$ from M_g , then for each element $(j', j, \tau_f) \in Th_g$ (by the unambiguity rules, this must exist and be uniquely defined), the compound evaluator does the following:

1. If evaluating $\mathit{split}(f \langle x \rangle g)$, it emits $\tau_f[x/\tau_g]$, and
2. otherwise, if it is evaluating $\mathit{split}(f \langle x \rangle g)$, then it emits $\tau_g[x/\tau_f]$.

On receiving a kill signal (KILL, j) from M_g , it finds the element $rec = (j', j, \tau_f) \in Th_g$, for some j' . Again, by the unambiguity rules, we assert that such an element exists and is unique. Next, it updates $Th_g := Th_g \setminus \{rec\}$, and finally, if $j' \notin Th_f$, then the compound evaluator itself emits the kill signal (KILL, j') .

It is easy to see that every result (RESULT, j, τ) emitted by the evaluator $M_{\mathit{split}(f \langle x \rangle g)}$ just constructed corresponds to $\llbracket \mathit{split}(f \langle x \rangle g) \rrbracket$ being defined on the thread beginning at index j . Conversely, if $\llbracket \mathit{split}(f \langle x \rangle g) \rrbracket$ is defined for the thread beginning at index j , then, because the input to the sub-evaluators is valid and by the induction hypothesis, it follows that the correct result (RESULT, j, τ) is emitted.

Every kill signal (KILL, j) emitted by the compound evaluator corresponds to a previously active thread, and it can be shown that this thread can no longer produce any results. Conversely, every time a thread j of $\mathit{split}(f \langle x \rangle g)$ becomes inactive, it has to be the case that j is an inactive thread of M_f , and for every index j' at which M_f reported a result, j' is an inactive thread of M_g . It follows that the thread beginning at index j is already killed by the compound evaluator. Formalizing this argument yields:

Proposition 16. *The evaluators $M_{\mathit{split}(f \langle x \rangle g)}$ and $M_{\mathit{split}(f \leftarrow^x g)}$ correctly compute $\mathit{split}(f \langle x \rangle g)$ and $\mathit{split}(f \leftarrow^x g)$ respectively.*

Observe that $|Th_f|$ and $|Th_g|$ are both bounded by the number of active threads of M_f and M_g respectively. Thus, the number of active threads of the compound evaluator is always bounded by $O(|f| + |g|)$, the memory usage is bounded by $O(|f| + |g| \text{mem}_\tau)$ and the response time is bounded by $O(\text{poly}(|\mathit{split}(f \langle x \rangle g)| \text{time}_\tau))$.

B.5 Function iteration

The evaluator $M_{iter(f, g_2, g_3, \dots, g_k)}$ is a more involved version of that for *split*. The evaluators for left-iteration are symmetric, and we will not discuss it here.

The evaluator maintains a set $Th \subseteq \mathbb{N} \times \mathbb{N} \times \text{TERM-REP}^k$ of threads being processed by M_f . Each element $(i, j, \tau_1, \tau_2, \dots, \tau_k)$ indicates that the compound evaluator thread beginning at index i most recently encountered a result at index j , and the terms $\tau_1, \tau_2, \dots, \tau_k$ summarize the intermediate results. The set Th is initialized to \emptyset . The evaluator behaves as follows.

On receiving a start signal, (START, i) , the compound evaluator updates $Th := Th \cup \{(i, i, x_1, x_2, \dots, x_k)\}$, sends the start signal (START, i) to each sub-evaluator, and responds with the result (RESULT, i, x_1) . Every data signal, (SYMBOL, i, d) is forwarded to the sub-evaluators.

On receiving a result $(\text{RESULT}, j, \tau_f)$ from M_f , $(\text{RESULT}, j, \tau_2)$ from M_{g_2} , $(\text{RESULT}, j, \tau_3)$ from M_{g_3} , \dots , $(\text{RESULT}, j, \tau_k)$ from M_{g_k} , all necessarily simultaneously, it does the following.

1. It finds the element $(j', j, \tau'_1, \tau'_2, \dots, \tau'_k) \in Th$, updates $(j', i, \tau_1, \tau_2, \dots, \tau_k)$, where $\tau_1 = \tau_f[x_1/\tau'_1, x_2/\tau'_2, \dots, x_k/\tau'_k]$.
2. It sends the signal (START, i) to each sub-evaluator. We assert that the sub-evaluators do not respond, because of unambiguity.
3. It emits the result $(\text{RESULT}, j', \tau_1)$.

On receiving a kill signal (KILL, j) , it first populates $kill\text{-}ring = \{(j', j, \tau) \in Th \mid \forall j' \in \mathbb{N}, \tau \in \text{TERM-REP}\}$. Next, it updates $Th := Th \setminus kill\text{-}ring$, and finally, for every j' such that $(j', j, \tau) \in kill\text{-}ring$ and $(j', j'', \tau') \notin Th$, it emits the kill signal (KILL, j') .

Correctness follows from similar arguments as for the iteration combinator of DReX.

B.6 Succinct representation of terms

We describe the procedure $prop\text{-}const(\tau)$:

1. If $\tau = \text{ins}((\tau_1, n), \tau_2)$, then return $(prop\text{-}const(\tau_1 + \tau_2), n + 1)$.
2. If $\tau = \text{avg}(\tau', n)$, then return $prop\text{-}const(\tau'/n)$.
3. Otherwise, let $\tau = op(\tau_1, \tau_2)$. Refer to table 1.
 - (a) If τ_1 is a constant c_1 , then return the value in the row corresponding to τ_2 and column corresponding to op .
 - (b) Otherwise, if τ_2 is a constant c_2 , then return the value in the row corresponding to τ_1 and column corresponding to op .
 - (c) Otherwise, both τ_1 and τ_2 are non-trivial terms. Simply return $op(\tau_1, \tau_2)$.

Table 1. Constant propagation rules for terms over $*$, $+$, \min and \max . The notation $\llbracket c_1 + c_2 \rrbracket$ (and $\llbracket op(c_1, c_2) \rrbracket$ in general) refers to eagerly adding the constants $c_1 \in \mathbb{R}$ and $c_2 \in \mathbb{R}$ and returning the result.

τ	$\cdot * c$	$\cdot + c$	$\min(\cdot, c)$	$\max(\cdot, c)$
c'	$\llbracket c' * c \rrbracket$	$\llbracket c' + c \rrbracket$	$\llbracket \min(c', c) \rrbracket$	$\llbracket \max(c', c) \rrbracket$
x	$x * c$	$x + c$	$\min(x, c)$	$\max(x, c)$
x/n'	$(x/n') * c$	$(x/n') + c$	$\min(x/n', c)$	$\max(x/n', c)$
$\tau_1 * \tau_2$	$\tau_1 * \text{simpl}(\tau_2 * c)$	$(\tau_1 * \tau_2) + c$	$\min(\tau_1 * \tau_2, c)$	$\max(\tau_1 * \tau_2, c)$
$\tau_1 + \tau_2$	$\text{simpl}(\tau_1 * c) + \text{simpl}(\tau_2 * c)$	$\tau_1 + \text{simpl}(\tau_2 + c)$	$\min(\tau_1 + \tau_2, c)$	$\max(\tau_1 + \tau_2, c)$
$\tau_1 - \tau_2$	$\text{simpl}(\tau_1 * c) - \text{simpl}(\tau_2 * c)$	$\tau_1 - \text{simpl}(\tau_2 - c)$	$\min(\tau_1 - \tau_2, c)$	$\max(\tau_1 - \tau_2, c)$
$\min(\tau_1, \tau_2)$	$\min(\text{simpl}(\tau_1 * c), \text{simpl}(\tau_2 * c))$	$\min(\text{simpl}(\tau_1 + c), \text{simpl}(\tau_2 + c))$	$\min(\tau_1, \text{simpl}(\tau_2, c))$	$\min(\text{simpl}(\max(\tau_1, c)), \text{simpl}(\max(\tau_2, c)))$
$\max(\tau_1, \tau_2)$	$\max(\text{simpl}(\tau_1 * c), \text{simpl}(\tau_2 * c))$	$\max(\text{simpl}(\tau_1 + c), \text{simpl}(\tau_2 + c))$	$\max(\text{simpl}(\min(\tau_1, c)), \text{simpl}(\min(\tau_2, c)))$	$\max(\tau_1, \text{simpl}(\tau_2, c))$

C Expressiveness of Quantitative Regular Expressions

C.1 A normal form for QREs

We complete the proof of theorem 5. Recall that $last = split(D^* [x] (true? d))$ outputs the last element of the input data stream, $project_1(f)$ modifies the QRE f to accept sequences of pairs and only examine the first element of each input pair, and $project_2(f)$ modifies f to accept sequences of pairs and only examine the second element of each input pair.

Cost operations. Consider the case when $e = op(f \gg g, h)$. Let $op : T_1 \times T_2 \rightarrow T$, $f : QRE(r_f, \emptyset, T_f)$, $g : QRE(T_f^*, X_g, T_1)$, and $h : QRE(D^*, X_h, T_2)$. Then observe that e is equivalent to

$$tuple(f, last) \gg op(project_1(g), project_2(h)).$$

The case where $e = op(f, g \gg h)$ is similar, and the construction can be generalized to the case where op is of k -arity.

Substitution. Observe that $e = (f \gg g)[x/h]$ is equivalent to

$$tuple(f, last) \gg (project_1(g)[x/project_2(h)]),$$

and that $e' = f[x/g \gg h]$ is equivalent to:

$$tuple(g, last) \gg project_2(f)[x/project_1(h)].$$

Choice. Consider $e = (f \gg g) \text{ else } h$. The expression is equivalent to

$$tuple(f, last) \gg (project_1(g) \text{ else } project_2(h)).$$

The case for $e' = f \text{ else } (g \gg h)$ is symmetrical.

Concatenation. Consider $e = split((f \gg g) [x] h)$. Let $f : QRE(r_f, X_f, T_f)$. Define $T'_f = T_f \cup \{undef\}$, i.e. T_f extended with an *undef* symbol. Next, let $f' = f \text{ else } (r_f^c ? undef)$ so that f' is defined over every stream w .

Let \tilde{g} be the extension of g which given a stream $int \in T_f \cup \{undef\}$, produces $g(int')$, where int' is the restriction of int to symbols in T_f . Observe that e is equivalent to

$$tuple(f', last) \gg split(\tilde{g} [x] project_2(h)).$$

A similar construction works for $split(f [x] (g \gg h))$, and for the left versions.

Iteration. Because the domain of $f \gg g$ is D^* , its domain is not unambiguously iterable, and cannot appear in the body of an iteration operator.

C.2 Streaming string to term transducers

Terms with variables. In Section 2 we defined the terms over a set of types \mathcal{T} and a set of operators G . We denote them here \mathbb{T} , and recall their definition:

$$\mathbb{T} ::= x \mid op(\tau_1, \dots, \tau_k)$$

Here $x \in X$ is a parameter, $op : T_1 \times \dots \times T_k \rightarrow T$ and $\tau_i \in \mathbb{T}$ are of type T_i . We use parameters to define a function with respect to some unknown inputs. We extend terms to include also a given set of *variables* X as follows:

$$\mathbb{T}_X ::= x \mid x \mid op(\tau_1, \dots, \tau_k) \mid \tau[x/\tau']$$

here $x \in X$ is a parameter, $x \in X$ is a variable, $\tau, \tau', \tau_1, \tau_2, \dots, \tau_k \in \mathbb{T}_X$, and τ_i is of type T_i .

We use variables in the definition of streaming transducers. Similar to automata states, the variables of our transducers are updated when the transducer processes a data element (unlike the parameters, which are not updated, but supply initial values). A function $\eta : X \rightarrow \mathbb{T}_X$ is termed an *update* function. We use $\mathcal{Y}_{\mathbb{T}}$ to denote the set of update functions.

Streaming string-to-term transducers. A *symbolic streaming string to term transducer (SSTT)* over \mathcal{T}, G is a tuple $\mathcal{S} = \langle \mathbb{D}, \Phi, X, X, Q, q_0, \eta_0, \Delta, \rho \rangle$ where:

- $\mathbb{D} \in \mathcal{T}$ is the *data domain*,
- Φ is an effective Boolean algebra over \mathbb{D} as defined in section 2.1
- X is a finite set of *parameters* and X is a finite set of *variables*,
- Q is a finite set of *states* and q_0 is the *initial state*,
- $\eta_0 : X \rightarrow X \cup T$ is the *initial valuation*,
- $\Delta : Q \times \Phi \rightarrow Q \times \mathcal{Y}_{\mathbb{T}}$ is a partial *transition function*,
- $\nu : Q \rightarrow \mathbb{T}_X$ is a partial *output function*.

An SSTT is *deterministic* if for every q, φ, φ' such that both $\Delta(q, \varphi)$ and $\Delta(q, \varphi')$ are defined, $\varphi \wedge \varphi'$ is unsatisfiable. We use $\delta(q, \varphi)$ and $\rho(q, \varphi)$ for the first and second components of $\Delta(q, \varphi)$, respectively.

The copylessness restriction. Given a variable $x \in X$ and a term $\tau \in \mathbb{T}_X$, we use $Occ(x, \tau)$ to denote the number of occurrences of the variable x in the term τ . We use $Vars(\tau)$ to denote the set of variables that occur in the term τ . A term $\tau \in \mathbb{T}_X$ is *copyless* if every variable $x \in X$ appears at most once in τ . An update function $\eta : X \rightarrow \mathbb{T}_X$ is *copyless* if no variable appears in the update of two different variables; formally, if for every $x \neq y \in X$ we have $Vars(\eta(x)) \cap Vars(\eta(y)) = \emptyset$. An SSTT is *copyless* if for every $q \in Q$ and $\varphi \in \Phi$ the update function $\rho(q, \varphi)$ is copyless. We use only deterministic copyless SSTTs, thus henceforth the term SSTT refers to a deterministic copyless SSTT.

Semantics A configuration of an SSTT \mathcal{S} is an element of $\{\perp\} \cup (Q \times \mathcal{Y}_{\mathbb{T}})$. The initial configuration is (q_0, η_0) . Assume the current configuration of the SSTT is (q, η) and it reads the data element $d \in \mathbb{D}$. Since \mathcal{S} is deterministic there exists at most one predicate φ such that $\varphi(d)$ holds and $\Delta(q, \varphi)$ is defined. If there is no such φ the subsequent configuration is \perp . Otherwise, let φ be this predicate. Then the subsequent configuration is (q', η') where $q' = \delta(q, \varphi)$ and $\eta'(x) = \rho(q, \varphi)(x)[x_1/\eta(x_1), \dots, x_m/\eta(x_m)]$ assuming $\mathbf{X} = \{x_1, \dots, x_m\}$. The subsequent configuration of \perp is \perp . A run of an SSTT on a data stream $w = d_1, d_2, \dots, d_n$ is a sequence of configurations $(q_0, \eta_0)(q_1, \eta_1) \dots (q_n, \eta_n)$ such that (q_0, η_0) is the initial configuration and for every $0 \leq i \leq n$, (q_i, η_i) is the subsequent configuration of (q_{i-1}, η_{i-1}) with respect to d_i . The output of the SSTT on w is the parametrized term $\eta_n(\nu(q_n))$ if $\nu(q_n)$ is defined and is undefined otherwise. An SSTT \mathcal{S} therefore induces a partial function $\llbracket \mathcal{S} \rrbracket : \mathbb{D}^* \rightarrow (T_1 \times \dots \times T_k \rightarrow T)_{\perp}$ exactly as a QRE does. Together, with an initial valuation to the parameters, the SSTT returns a value in the output type T . The symbolic DFA $D_{\mathcal{S}} = \langle \mathbb{T}, Q, q_0, \delta, F \rangle$ obtained from the SSTT by removing the variables updates and setting F to the domain of ν characterizes the domain of \mathcal{S} .

C.3 From QREs to SSTTs

SSTTs with regular lookahead. An SSTT makes its decisions based on the prefix of the data stream read so far. An SSTT with regular look-ahead can make its decisions also based on whether the remaining suffix of the input data stream belongs to some regular language. To do that, it inspects the states visited by a symbolic lookahead DFA R that reads the data stream w in reverse. Formally if $w = d_1, d_2, \dots, d_n$, we assume $R = (\mathbb{D}, S, s_0, \delta, F)$ reads the reverse of w , i.e. the data stream d_n, \dots, d_2, d_1 . Assume that the run of R on the reverse of w is $s_0 \xrightarrow{d_n} s_1 \xrightarrow{d_{n-1}} s_2 \xrightarrow{d_{n-2}} \dots \xrightarrow{d_1} s_n$. We say that R generates the labeling $s_0, s_1, s_2, \dots, s_n$ on w . An SSTT with regular look-ahead reads both the given data stream and the labeling of R on the given input data stream. It is shown in [4] that regular look-ahead does not increase the expressivity of SSTTs:

Proposition 17 (Closure under regular lookahead [4]). *Let S be an SSTT with regular look-ahead. There exists an SSTT S' without regular look ahead such that $\llbracket S \rrbracket = \llbracket S' \rrbracket$.*

Proof of theorem 6. The proof is by induction over the structure of f . We build an SSTT $\mathcal{S}_f = (\mathbb{D}, \Phi, X, \mathbf{X}, Q, q_0, \eta_0, \Delta, \nu)$ as follows. In the induction steps we often assume SSTTs $\mathcal{S}_i = (\mathbb{D}, \Phi, X_i, \mathbf{X}_i, Q_i, q_{0i}, \eta_{0i}, \Delta_i, \nu_i)$ with disjoint sets of states and variables. For many induction steps the transition relation is that of a product construction. We thus define $\delta_{\otimes}((q_1, q_2, \dots, q_k), d) = (\delta_1(q_1, d), \delta_2(q_2, d), \dots, \delta_k(q_k, d))$ and use it throughout.

- For a base function $\varphi ? \lambda$ where $\varphi \in \Phi$ and $\lambda : \mathbb{D} \rightarrow T$ is an atomic function, we build a copyless SSTT from scratch as follows. \mathcal{S} has two states q_0 and q_1 , where q_0 is the initial state. Its set of parameters is empty. Its set of variables

is $X = \{x\}$. The initial valuation function maps x to 0.⁴ It has one transition from q_0 to q_1 labeled $\varphi/x := \lambda$. The output functions maps q_1 to x . Clearly \mathcal{S} is copyless.

- For a base function $\epsilon ? x$, where x is a parameter, we build a copyless SSTT from scratch as follows. \mathcal{S} has one state q_0 which is the initial state. Its set of parameters is $\{x\}$. Its set of variables is $\{x\}$. The initial valuation function maps x to x . It has no transitions. The output functions maps q_0 to x . Clearly \mathcal{S} is copyless.
- For $op(f_1, f_2, \dots, f_k)$ we combine SSTTs $\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_k$ for f_1, f_2, \dots, f_k , respectively using a product construction. Thus, the states of \mathcal{S} are $Q_1 \times Q_2 \times \dots \times Q_k$. The initial state is (q_{01}, \dots, q_{0k}) and the transition relation is δ_{\otimes} . The set of parameters is $X_1 \cup \dots \cup X_k$. The variables of \mathcal{S} are $X_1 \cup \dots \cup X_k$. The initial valuation function maps $x \in X_i$ to $\eta_i(x)$. For a transition from (q_1, \dots, q_k) on φ the variable update function ρ maps a variables $x \in X_i$ to the term $\rho_i(q_i, \varphi)(x)$ the update function of \mathcal{S}_i maps x to. The output function ν maps a state (q_1, \dots, q_k) to $op(\nu_1(q_1), \dots, \nu_k(q_k))$. Note that given $\mathcal{S}_1, \dots, \mathcal{S}_k$ are copyless, since variables of different transducers are mixed only at the output function, \mathcal{S} is copyless as well.
- For $f = f_1 \text{ else } f_2$ we again combine the SSTTs for \mathcal{S}_1 and \mathcal{S}_2 using a product construction. The states are $Q_1 \times Q_2$. The initial state is (q_{01}, q_{02}) . The transition relation is δ_{\otimes} . The parameters are $X_1 \cup X_2$, and the variables are $X_1 \cup X_2$. The initial valuation maps $x \in X_1$ to $\eta_1(x)$ and $x \in X_2$ to $\eta_2(x)$ (since the set X_1 and X_2 are disjoint, this is well defined). The variables update function ρ maps $((q_1, q_2), \varphi)(x)$ to $\rho_1(q_1, \varphi)(x)$ if $x \in X_1$ and to $\rho_2(q_2, \varphi)(x)$ otherwise. The output function maps a state (q_1, q_2) to $\nu_1(q_1)$ if it is different than \perp and to $\nu_2(q_2)$ otherwise. Since \mathcal{S} the does not mix variables beyond what is in \mathcal{S}_1 or \mathcal{S}_2 , \mathcal{S} is copyless.
- For $f = \text{split}(f_1 [x] f_2)$ we use SSTTs with regular look-ahead to answer whether the suffix of the data stream read so far is in $Dom(f_2)$. Let $R = (\mathbb{D}, S_2, s_{02}, \delta_R, F)$ be a symbolic DFA for the *reverse* of $Dom(f_2)$. Let \mathcal{S}_1 and \mathcal{S}_2 be copyless SSTTs for f_1 and f_2 . We build a copyless SSTT with regular lookahead R as follows. The states are $Q_1 \cup Q_2$. The initial state is q_{01} . The parameters are $X_1 \cup X_2 \setminus \{x\}$. The variables are $X_1 \cup X_2 \cup \{x\}$. The initial valuation maps variables of X_1 as \mathcal{S}_1 does and variables of X_2 as \mathcal{S}_2 does. Assume a state q_1 such that $q_1 \in Q_1$ and a next data element to be read (d, s) . (Recall that an SSTT with regular lookahead reads a pair (d, s) where d is the next data element and s the state of the satellite DFA.) Let φ be the predicate for which $\varphi(d)$ is satisfiable and $\Delta_1(q_1, \varphi)$ is defined. Let $q'_1 = \delta_1(q_1, \varphi)$ and $s' = \delta_R(s, \varphi)$. If $q'_1 \notin Dom(\nu_1)$ or $s' \notin F$ the next state is q'_1 . The variable update in such a case updates variables in X_1 following \mathcal{S}_1 's update, and keeps variables in X_2 unchanged. If $q'_1 \in Dom(\nu_1)$ and $s \in F$ the next state is q_{02} . The variable update in such a case updates the designated variable x to $\nu_1(\delta_1(q_1, \varphi))$, sets all variables of X_1 with the default value 0 and keeps the other variables of X_2 unchanged. From state q_2 such

⁴ We use 0 for a default value of the cost domain.

that $q_2 \in Q_2$, on reading data element (d, s) the next state is $\delta(q_2, \varphi)$ for φ such that $\varphi(d)$ holds and $\Delta(q_2, \varphi)$ is defined. The variable update in such a case updates the variables of \mathbf{X}_2 as \mathcal{S}_2 does and keeps the other variables unchanged. The output function maps a state q_2 for $q_2 \in Q_2$ to $\nu_2(q_2)$ and a state q_1 for $q_1 \in Q_1$ to \perp . Note that if $w = w_1 w_2$ such that $w_1 \in \text{Dom}(f_1)$ and $w_2 \in \text{Dom}(f_2)$, the variables are updated according to \mathcal{S}_1 as long as w_1 is read and according to \mathcal{S}_2 when w_2 is read. On the transition, the designated variable x of \mathbf{X}_2 gets the output of \mathcal{S}_1 , which may combine several variables of \mathbf{X}_1 , but in this case those other variables are set to a default value. Thus \mathcal{S} is copyless as well.

- For $f = \textit{split}(f_1 \langle x \rangle f_2)$ we also use an SSTT with regular lookahead. Here however, f_1 relies on the value of f_2 which is not available when \mathcal{S}_1 processes w_1 , the first part of the data stream. However, \mathcal{S}_1 is set to receive a parameter x that under *split* corresponds to the output of f_2 on w_2 , the second part of the data stream. The construction thus proceeds as in the case of *split*, with the following two differences: (a) when \mathcal{S}_2 effectively takes control, the designated variable x of \mathbf{X}_2 is updated to $\nu_1(q_1)$ where q_1 is the state \mathcal{S}_1 arrived at, all other variables in \mathbf{X}_1 are set to their default value and variables of \mathbf{X}_2 retain their values, and (b) the output function maps a state q_2 for $q_2 \in Q_2$ to $x[\nu_2(q_2)]$ and to \top otherwise. Here as well, the only mix of variables in \mathcal{S} that does not already exists in \mathcal{S}_1 or \mathcal{S}_2 is on the transition to \mathcal{S}_2 , in which the designated variable x mixes some variables of \mathbf{X}_1 , but all these variables are set to their default value. Thus \mathcal{S} is copyless as well.
- For *iter*(f) we again use an SSTT with regular lookahead. The idea relies on that of *split*(\cdot), but considers a DFA R for $\text{Dom}(f)^*$ for the regular look ahead.
- For *iter*(f) we again use an SSTT with regular lookahead. The idea relies on that of *split*(\cdot), but considers a DFA R for $\text{Dom}(f)^*$ for the regular look ahead.
- For $f = f_1[x/f_2]$ we build an SSTT using a product construction of the SSTTs for \mathcal{S}_1 and \mathcal{S}_2 for f_1 and f_2 , that processes f_1 and f_2 in different set of variables, as in *op*(f_1, f_2) for instance. The output function here maps state (q_1, q_2) to $\nu(q_1)[x/\nu(q_2)]$.

C.4 From SSTTs to QREs

We now complete the proof of theorem 7.

Base. For the base case we consider regular expressions of the form $R[i, i, 0, \theta_{id}]$ and $R[i, j, 0, \theta]$. For R of the former type we have $\mathbf{V}_R = (f_1, \dots, f_m)$ where $f_i = \epsilon ? x$. For R of the second type, we have $f_i = (\varphi_1 ? \tau_{1i} \textit{else} \varphi_2 ? \tau_{2i} \cdots \textit{else} \varphi_\ell ? \tau_{\ell i})$ assuming there are ℓ transitions from i to j with flow θ and labels $\varphi_1, \dots, \varphi_\ell$, respectively and on transition labeled φ_j the update of x_i is given by τ_{ji} .

Concatenation. Let $\mathbf{V}_R = (f_1, \dots, f_m)$ and $\mathbf{V}_{R'} = (g_1, \dots, g_m)$ be the vectors of QRE corresponding to some $R = R[i, k, k, \theta]$ and $R' = R[k, j, k, \theta']$. Then $\mathbf{V}_{R \cdot R'} = (h_1, \dots, h_m)$ where $h_i = \text{split}(R?0 [z] z + g_i)[x_1/f'_1, \dots, x_m/f'_m]$ and $f'_i(z) = \text{split}(f_i [z] z + R'?0)$.

Choice. Let $\mathbf{V}_R = (f_1, \dots, f_m)$ and $\mathbf{V}_{R'} = (g_1, \dots, g_m)$ be the vectors of QRE corresponding to some $R, R' \in \mathcal{R}^\Theta$ such that $R \cap R' = \emptyset$. Then $\mathbf{V}_{R \uplus R'} = (h_1, \dots, h_m)$ where $h_i = f_i \text{ else } g_i$. In the rest of this proof we will use $\mathbf{V}_R \uplus \mathbf{V}'_R$ to denote $\mathbf{V}_{R \uplus R'}$.

Iteration. Given $\Theta_N = \{\theta_1, \theta_2, \dots, \theta_t\}$ and given $\mathbf{V}_{R_1}, \mathbf{V}_{R_2}, \dots, \mathbf{V}_{R_t}$ for $R_i = R[k, k, k-1, \theta_i]$ and some $\theta \in \Theta^N$ we would like to calculate \mathbf{V}_{R^*} for $R = R[k, k, k-1, \theta]$. If θ is idempotent, and $\mathbf{V}_R = (f_1, \dots, f_m)$ we define $\mathbf{V}_{R^*} = (f_1^*, \dots, f_m^*)$ where given the support of f_i is $\{x_i\} \cup \{x_{i_1}, \dots, x_{i_\ell}\}$ we let $f_i^* = \text{iter}(f_i, f_{i_1}, \dots, f_{i_\ell})$.

If θ is not idempotent, we cannot simply apply *iter* as is. Since for non-idempotent flows the next iteration depends not only on x_i and the result of the other f_j 's on the current block, but also on the result of the other f_j 's on previous blocks. Such values cannot be passed via substitution. So we may try to concatenate the path to itself until it stabilizes. Indeed for any normal flow θ there exists $k \leq m$ such that θ^k is idempotent. The difficulty here is that we need to account for all possible paths from k to k simultaneously, all their possible flows, and all flows obtained by concatenating an arbitrary long sequence of such paths to each other. To cope with these challenges we use the notion of *support* of a flow, with which we can define a partial order on normal flows, that we can use to compute \mathbf{V}_{R^*} inductively.

The *support* of a flow θ , denoted $\text{Supp}(\theta)$, is the set of variables that flow into themselves, i.e. the set $\{i \mid \theta(i) = i\}$. We use $\theta_1 \sim \theta_2$ to denote that $\text{Supp}(\theta_1) = \text{Supp}(\theta_2)$; $\theta_1 \sqsubset \theta_2$ to denote that $\text{Supp}(\theta_2) \subset \text{Supp}(\theta_1)$; and $\theta_1 \sqsubseteq \theta_2$ to denote that $\theta_1 \sim \theta_2$ or $\theta_1 \sqsubset \theta_2$. Let θ be the flow of an arbitrary long concatenation of flows $\theta_1 \theta_2 \dots \theta_\ell$. It can be shown that this concatenation can be decomposed uniquely to blocks by the first time the result of the concatenation has some flow $\theta' \sim \theta$. There may be a remaining block with flow $\theta' \sqsubset \theta$, and there should be a first time where the results of the concatenation of several such blocks has flow θ .⁵ This is depicted as follows:

$$\underbrace{\theta_1 \theta_2 \dots \theta_{i_1}}_{\theta'_1 \sim \theta} \underbrace{\theta_{i_1+1} \dots \theta_{i_2}}_{\theta'_2 \sim \theta} \underbrace{\theta_{i_2+1} \dots \theta_{i_3}}_{\theta'_3 \sim \theta} \dots \underbrace{\theta_{i_{k+1}} \theta_{i_{k+2}} \dots \theta_\ell}_{\theta'_{k+1} \sqsubset \theta}$$

$$\underbrace{\theta'_1 \theta'_2 \dots \theta'_j}_{=\theta} \underbrace{\theta'_{j+1}}_{\sim \theta} \underbrace{\theta'_{j+2}}_{\sim \theta} \dots \underbrace{\theta'_k}_{\sim \theta} \underbrace{\theta'_{k+1}}_{\sqsubset \theta}$$

⁵ This follows from the following basic properties on flows: (a) $\theta_1, \theta_2 \in \Theta_N$ implies $\theta_1 \sqsubset \theta_1 \theta_2$ and $\theta_2 \sqsubseteq \theta_1 \theta_2$, (b) $\theta_L, \theta_R \in \Theta_I$ and $\theta_L \sim \theta_R$ implies $\theta_L \theta_R = \theta_L$, and (c) for all $\theta \in \Theta_N$ there exist $\theta' \in \Theta_I$ and $j \in \mathbb{N}$ such that $\theta' \sim \theta$ and $\forall k \geq j, \theta^k = \theta'$.

To compute the iteration of a vector \mathbf{V}_R we would thus like to consider the following restrictions on flows. For a given $\theta \in \Theta$ we define the following *flow restrictions*:

- θ^T : the overall flow is θ .
- θ^F : the overall flow is θ , and no prefix has flow θ .
- θ^E : the overall flow is θ' for some $\theta' \sim \theta$, and no prefix has flow $\sim \theta$.

We thus enhance the set of considered regular expressions to consider the above restrictions. For $\theta \in \Theta_N$, $\theta^R \in \{\theta^E, \theta^F, \theta^T\}$ and $i, j, k \in [1..n]$, let $R[i, j, k, \theta^R]$ denote the set of all data streams w such that when \mathcal{S} processes w starting in state i , it reaches state j without passing through any state indexed greater than k and the overall flow is consistent with the flow restriction θ^R . We use $\mathcal{R}_{\text{EFT}}^\Theta$ to denote the set of all such regular expressions. Note that $R[i, j, k, \theta^T] = R[i, j, k, \theta]$, thus we can safely refine \mathcal{R}^Θ to $\mathcal{R}_{\text{EFT}}^\Theta$ only for the case of iteration.

Back to our starting point, given \mathbf{V}_θ is the vector of QRE corresponding to $R[k, k, k-1, \theta]$ and we have such vectors for all $\theta \in \Theta^N$, we would like to calculate \mathbf{V}_{θ^*} corresponding to $(R[k, k, k-1, \theta])^*$. We will do so using the intermediate vectors \mathbf{E}_θ and \mathbf{F}_θ corresponding to $R[k, k, k-1, \theta^E]$ and $R[k, k, k-1, \theta^F]$, respectively. Define

$$E_\theta = \left\{ (\theta_1, \theta_2, \dots, \theta_\ell) \mid \begin{array}{l} \theta_1 \theta_2 \dots \theta_\ell \sim \theta, \ell < m, \\ \forall j < \ell, \theta_1 \theta_2 \dots \theta_j \not\sim \theta \end{array} \right\}$$

$$F_\theta = \left\{ (\theta_1, \theta_2, \dots, \theta_\ell) \mid \begin{array}{l} \theta_1 \theta_2 \dots \theta_\ell = \theta, \ell < m, \\ \forall j < \ell, \theta_1 \theta_2 \dots \theta_j \neq \theta \end{array} \right\}.$$

It follows from the above arguments that

$$\mathbf{E}_\theta = \bigsqcup_{(\theta_1, \theta_2, \dots, \theta_\ell) \in E_\theta} \mathbf{V}_{\theta_1} \mathbf{V}_{\theta_2} \dots \mathbf{V}_{\theta_\ell}$$

$$\mathbf{F}_\theta = \bigsqcup_{(\theta_1, \theta_2, \dots, \theta_\ell) \in F_\theta} \mathbf{E}_{\theta_1} \mathbf{E}_{\theta_2} \dots \mathbf{E}_{\theta_\ell}$$

$$\mathbf{V}_{\theta^*} = \mathbf{F}_\theta \left(\left(\bigsqcup_{\theta' \sim \theta} \mathbf{E}_{\theta'} \right)^m \right)^* \left(\bigsqcup_{\theta' \sim \theta} \mathbf{E}_{\theta'} \right)^{\leq m} \left(\bigsqcup_{\theta' \sqsubseteq \theta} \mathbf{V}_{\theta'} \right)$$

Note that $(\bigsqcup_{\theta' \sim \theta} \mathbf{E}_{\theta'})^m$ is guaranteed to be idempotent. Thus, since we know how to iterate vectors corresponding to idempotent flows, and by concatenation and choice defined previously we can compute \mathbf{V}_{θ^*} inductively.

Output QRE. Assume the output function is defined on states $\{i_1, \dots, i_F\}$. For $1 \leq i \leq F$, and flow θ let $R = R[1, n, i, \theta]$ and let $\mathbf{V}_R = (f_1, \dots, f_m)$ be the vector of QREs corresponding to R . If the update function for i returns h and h depends on $\{x_{j_1}, \dots, x_{j_k}\}$ let $h_{i, \theta} = h[x_{j_1}/f_{j_1}, \dots, x_{j_k}/f_{j_k}]$ and let $h_i = \bigsqcup_{\theta' \in \Theta^N} h_{i, \theta}$. Then construct $f_S = \bigsqcup_{1 \leq i \leq F} h_i$. By construction, f_S is equivalent to the original SSTT \mathcal{S} . This completes the proof of theorem 7.