Regular Programming over Data Streams

Mukund Raghothaman

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2017

Supervisor of Dissertation	Graduate Group Chairperson
Rajeev Alur Professor of Computer and Information Science	Lyle Ungar Professor of Computer and Information Science

Dissertation Committee
Susan Davidson, Professor of Computer and Information Science
Sanjeev Khanna, Professor of Computer and Information Science
Benjamin Pierce, Professor of Computer and Information Science
Moshe Vardi, Professor of Computational Science, Rice University

Acknowledgements

First and foremost, I would like to thank my advisor, Rajeev Alur. This thesis would not have been possible without his support and guidance.

My thesis committee consisted of Susan Davidson, Sanjeev Khanna, Benjamin Pierce, and Moshe Vardi: I thank them for the careful reading of this document and suggestions for improvement. I would also like to thank Youssef Hamadi and Yi Wei for hosting me at Microsoft Research Cambridge for two exciting and productive summers.

One of the best parts about working with Rajeev is the wonderful group of postdocs and research scientists he assembles. Dana Fisman, Kostas Mamouras, and Arjun Radhakrishna deserve special mention: their keen eye for detail has contributed immeasurably towards making this a more rigorous and better presented thesis. Christos Stergiou, Ashutosh Trivedi, and Jyotirmoy Deshmukh provided great guidance at various stages of this Ph.D. journey. Jyotirmoy was the one who pushed me to follow up what would become my first research paper, and continues to be a rich source of technical problems and an excellent sounding board for ideas.

Over six years and innumerable cups of coffee, Abhishek Udupa and I have become close friends and professional collaborators. The cohort of Rajeev's Ph.D. students—Abhishek, Loris D'Antoni, Salar Moarref, Yifei Yuan, and Nimit Singhania—formed an amazing and lively peer group with which to discuss research problems.

My office-mates in GRW 571—Salar, Nan Zheng, Kai Hong, and Shaheen Jabbari—have each contributed in their little way to the successful completion of this thesis. Arjun Ravi Narayan taught me a lot: from writing code to economics trivia and general wisdom about life. Mike Felker greatly simplified and helped me navigate the bureaucracy associated with being an international Ph.D. student.

Outside the department, Varun Aggarwala, Carin Molenaar, the ECE Lobby from college, and Abhishek Anand were all always there. Finally, I thank my parents, my grandmother, and my brother, without whom I would not even have begun work on this thesis. For them, no words of gratitude will ever be enough.

Mukund Raghothaman
March 8, 2017

ABSTRACT
REGULAR PROGRAMMING OVER DATA STREAMS

Mukund Raghothaman

Rajeev Alur

Data streams arise in a variety of applications, such as feeds from financial markets, event streams from sensors and medical devices, logs produced by long-running programs, click-streams from websites, and packet sequences passing through internet routers. In this thesis, we are concerned with computing quantitative statistics over these streams, and with expressing transformations in the related domain of strings. Many string transformations are instances of simple patterns, such as inserting, deleting and replacing substrings, or applying a function to each element in the stream. Over data streams, the task is usually to compute some simple quantitative statistic, such as counting the number of occurrences of a pattern or the mean time between occurrences of an event.

There has traditionally been limited programming language support for stream processing, and programmers are forced to write low-level code, by manually maintaining state and updating it on seeing each new input element. This sacrifices both ease of expression and amenability to static analysis. We propose a simple, expressive programming model for stream transformations, with strong theoretical foundations and fast evaluation algorithms.

We present two concrete systems: DReX, to express string-to-string transformations, and quantitative regular expressions (QREs) for numerical queries. Both formalisms start with a set of basic functions and a small collection of hierarchically composable *combinators*, analogous to the operations of regular expressions. The operators are simple to describe, and can be used to combine small, easy-to-understand expressions into more complicated expressions.

The functions expressible using DReX and QREs coincide with the class of *regular string transformations*, which is a robust class with multiple characterizations and appealing closure properties (under composition, input reversal, and regular look-ahead). We present a single-pass linear-time evaluation algorithm for function expressions, and study efficient approximate representations of numerical terms, so that some numerical QREs can also be evaluated with sub-linear memory requirements.

This thesis is based on material drawn from the following publications:

1. Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science*, CSL-LICS 2014, pages 9:1–9:10. ACM, 2014.

2. Rajeev Alur, Loris D'Antoni, and Mukund Raghothaman. DReX: A declarative language for efficiently evaluating regular string transformations. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, pages 125–137. ACM, 2015.

3. Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In Peter Thiemann, editor, *Programming Languages and Systems: Proceedings of the 25th European Symposium on Programming*, ESOP 2016, pages 15–40. Springer, 2016.

4. Rajeev Alur, Sanjeev Khanna, Zachary Ives, Konstantinos Mamouras, and Mukund Raghothaman. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In submission, 2016.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Programmers routinely encounter sequential data, traditionally as strings, and more recently as streams, arising in applications such as financial markets, click-streams from websites, event logs produced by programs, and measurements made by sensors and medical devices. When dealing with strings, the programmer typically wishes to perform some simple operation, such as extract, reorder, or delete substrings according to some regular pattern. Over data streams, the task is usually to compute some simple quantitative statistic, such as counting the number of occurrences of a pattern or the mean time between occurrences of an event.

In this thesis, we will introduce high-level abstractions to program queries over data streams. There has been limited language support for these functions: traditional regular expressions do not produce non-boolean outputs, database query languages such as SQL focus on the relational abstraction rather than on the sequential nature of the data, and contemporary stream processing systems consider the orthogonal problem of connecting a topology of individual stream processors, where each stream processor is itself written in low-level code and contains explicit instructions on how to update its state on seeing each new element, i.e. the problem of composing individual stream transformations, each written a low-level language of the programmer's choice.

Writing low-level stateful code is a cumbersome task, and it is difficult to provide programmer assistance for debugging, static analysis, or automatic parallelization. The benefits of high-level abstractions are well-established, for e.g., by successful domain-specific languages such as SQL and regular expressions. In our system, the programmer writes a succinct description of the stream transformation or quantitative query in which they are interested, and we automatically generate efficient query evaluators. The formalism based on the concept of function combinators—constructs which combine functions into larger and more complicated functions. There is a small collection of these core combinators, and each combinator has a simple, intuitive description. Queries written in this style have the advantage of being modular and easy-to-understand, and emphasize *what* needs to be computed over operational details describing *how* the function needs to be computed.

In this thesis, we will address both the logical foundations—what class of queries can we express in our system—and the algorithmic foundations—by showing that queries can be evaluated with low time- and space-complexity.

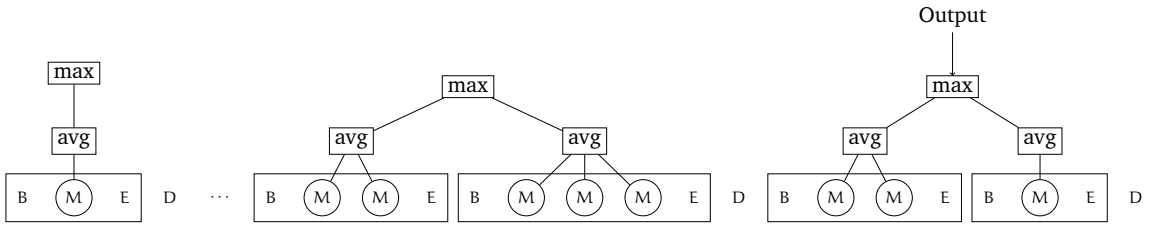


Figure 1.1: Hypothetical EEG event stream from a patient. There are four types of events: (*a*) B indicates the beginning of an episode, (*b*) $M(t, v)$ indicates the measurement of the value v at time t, (*c*) E indicates the end of the episode, and (*d*) D indicates the end of a calendar day. We indicate each episode by drawing a box around its events. The doctor wishes to compute the maximum of the per-episode average measurements for the last day.

Example 1.1. Consider the example of a patient being monitored in a hospital for a condition such as epilepsy. The hypothetical data stream consists of four types of events: (*a*) B, indicating the beginning of an episode, (*b*) EEG sensor readings $M(t, v)$, indicating the timestamp t and value v observed, (*c*) E, indicating the end of an episode, and (*d*) D, indicating the end of a calendar day. The doctor wishes to know if the patient had a severe episode on the last day: because individual measurements during an episode may widely vary, they measure severity by the average measurement during an episode. The query is to find the maximum over all episodes of the last day, of the average measurement during the episode. See figure 1.1.

For simplicity, let us say that the event stream has the form $((B \cdot M^+ \cdot E)^+ \cdot D)^+$. Given a sequence of measurements $w \in M^+$, the expression:

$$e_1 = \text{iter}^+(M(t, v) \mapsto v, \text{avg})$$

extracts the value from each measurement event, and computes the average of these values over the entire sequence. The measurements of a single episode are flanked on the left and on the right by begin- and end-markers, B and E respectively. The expression:

$$e_2 = \text{split}(B, e_1, E, \pi_2)$$

maps event streams of the form $B \cdot M^+ \cdot E$ to the output of e_1 on the sub-sequence of measurements. The input stream w is broken into three parts, $w = w_1 w_2 w_3$, where w_1 is of the form B, w_2 is fed as input to e_1, and w_3 is of the form E. The results from the sub-expressions are combined using the usual second projection operator, $\pi_2(a, b, c) = b$. The third expression:

$$e_3 = \text{split}(\text{iter}^+(e_2, \text{max}), D, \pi_1)$$

maps the events of a single day, $w \in [\![(B \cdot M^+ \cdot E)^+ \cdot D]\!]$ to the severity of the most intense episode. Recall that the doctor was interested in this measurement for the last day:

$$e_4 = \text{split}(((B \cdot M^+ \cdot E)^+ \cdot D)^*, e_3, \pi_2). \qquad \triangle$$

The expressions we have written are inspired by the common notation of regular expressions. Just as with regular expressions, the operations, $\text{split}(e, \mathrm{f}, \text{op})$, $\text{iter}(e, \text{op})$, etc. do not specify the *mechanics* of parsing the input stream, but rather only specify *how to recognize* a viable decomposition: "Split the input stream w into two parts $w = w_1 w_2$ such that both $v_1 = [\![e]\!](w_1)$ and $v_2 = [\![\mathrm{f}]\!](w_2)$ are defined. Output $\text{op}(v_1, v_2)$."

We will consider three concrete questions in this thesis: (*a*) How do we define programming abstractions so that we can write expressions similar to those we have just written? (*b*) What class of queries can we express using this notation? And (*c*) what is the time and space complexity of evaluating queries expressed in this notation? By using high-level combinators such as split, iteration (iter), or global choice (e else f), our hope is that we free the programmer from the burden of explicitly reasoning about state and case analyses based on potentially unseen events. Notably, the only structure we assume inherent in the input data is that it is sequential. The evaluation tree, such as that shown in figure 1.1, is an artifact of the query rather than of the data: different queries can potentially impose different structure on the same input stream.

Example 1.2. Let us consider an example from text processing. Given an email address such as "`user@domain`", we wish to extract the domain name, "`domain`". We can write function expressions in a similar manner as we did for the EEG data stream:

$$e_1 = \text{iter}(\mathrm{x} \neq \texttt{@} \mapsto \epsilon)$$

uniformly maps input streams of the form $(\neg\texttt{@})^*$ to the empty output string ϵ. We are expressing functions $[\![e]\!] : \Sigma^* \to \Gamma^*$: the only way to combine the outputs in this case is by string concatenation, and the concatenation operator, "$\cdot$", is implicit: $\text{iter}(e)$ is shorthand for $\text{iter}(e, \cdot)$.

The second expression,

$$e_2 = \text{iter}(\mathrm{x} \neq \texttt{@} \mapsto \mathrm{x}),$$

is also only defined for strings of the form $(\neg\texttt{@})^*$, but simply echoes such strings unchanged. We wished to extract the domain name from the email address: this is accomplished by the expression,

$$e_3 = \text{split}(e_1, \texttt{@} \mapsto \epsilon, e_2). \qquad \triangle$$

Function combinators. We will present two formalisms: DReX, to express string-to-string transformations, and quantitative regular expressions (QREs), for numerical functions over data streams. In both formalisms, we start with a small collection of basic functions of the form $\varphi \mapsto \lambda$, where $\varphi : \Sigma \to \text{Bool}$ is a single-element predicate, and $\lambda : \Sigma \to \mathbb{D}$ is a single-element transformation: if the input stream w consists of a single element, and $\varphi(w) = \text{true}$, then output $\lambda(w)$.

We then hierarchically combine function expressions using combinators which are very similar to those from regular expressions: e else f is the analogue of union, $\text{split}(e, \mathrm{f}, \text{op})$ is

the analogue of concatenation, and $\text{iter}(e, \text{op})$ is the analogue of Kleene-*. We also include the operation combine:

$$[\![\text{combine}(e, f)]\!](w) = [\![e]\!](w) \cdot [\![f]\!](w),$$

or in the world of numerical quantities, $\text{op}(e, f)$, for some numerical operator $\text{op} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$:

$$[\![\text{op}(e, f)]\!](w) = \text{op}([\![e]\!](w), [\![f]\!](w)).$$

This combinator is the counterpart of the intersection operator in regular expressions. Our ultimate motivation is to obtain a formalism which is easy to understand, efficiently parseable and expressively powerful.

Example 1.3. There are several situations where combine and op are useful constructs. Consider the function $\text{copy} : \Sigma^* \to \Sigma^*$, defined as $\text{copy}(w) = ww$. The identity function can be expressed as $\text{id} = \text{iter}(\text{true} \mapsto x)$, and so we write

$$\text{copy} = \text{combine}(\text{id}, \text{id}).$$

In the case of the patient data stream, we might be interested in mapping episodes $w \in M^+$ to their inter-quartile range. We would then be interested in the QRE:

$$\begin{aligned} \text{iqr} &= s_{0.75} - s_{0.25}, \text{ where} \\ s_{0.75} &= \text{iter}^+(M(t, v) \mapsto v, \text{select}_{0.75}), \text{ and} \\ s_{0.25} &= \text{iter}^+(M(t, v) \mapsto v, \text{select}_{0.25}). \end{aligned}$$

Here, the function $\text{select}_k : \text{MSet}(\mathbb{R}) \to \mathbb{R}$ maps a multiset A to the $k|A|$-th smallest element of A. Observe that in the expression iqr, both operands of the "$-$", $s_{0.75}$ and $s_{0.25}$, are themselves QREs rather than concrete real numbers. $[\![\text{iqr}]\!](w) = [\![s_{0.75}]\!](w) - [\![s_{0.25}]\!](w)$. $\triangle$

1.1 QREs, More Generally

Historically, we developed QREs after introducing DReX and proving most of the results of this thesis for the case of string-to-string transformations. Our motivation in developing QREs was to obtain similar a similar abstraction for numerical functions as DReX is for string-valued functions. Output domains such as the set of real numbers provide the most important applications, justifying the qualifier "*quantitative*" in the phrase "quantitative regular expressions". Primarily, however, QREs are a significant generalization of DReX, where both the output domain and associated cost operations, such as the choice of "op", are arbitrary and chosen by the programmer when constructing the expression. (We will formally state this connection between QREs and DReX in theorem 2.24.)

In general, we assume almost no properties of the output type (except that it is non-empty, for some constructions where we would like variables to be arbitrarily initialized), or of operators $\text{op} : T_1 \times T_2 \times \cdots \times T_k \to T$ (except that they be computable). They need not possess identity elements, or satisfy any interesting algebraic property, such as associativity or

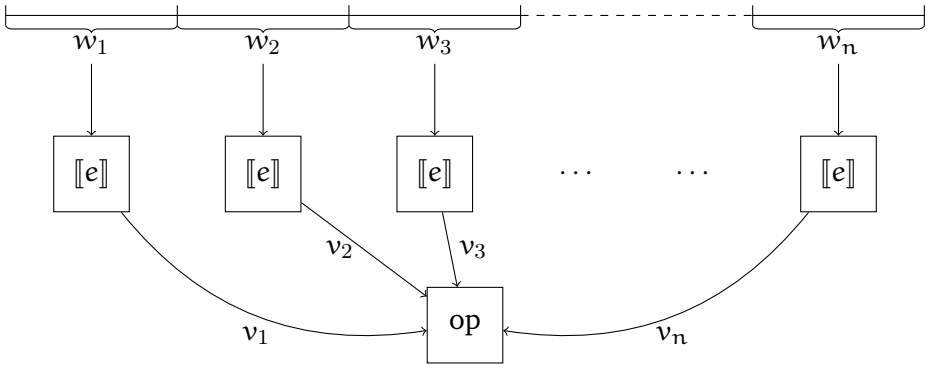


Figure 1.2: The semantics of $\text{iter}(e, \text{op})$. The input stream is divided into pieces, $w = w_1 w_2 \cdots w_n$, such that $v_i = [\![e]\!](w_i)$ is defined for each i. Given an associative operator $\text{op} : \Gamma \times \Gamma \to \Gamma$ with an identity element c, $[\![\text{iter}(e, \text{op})]\!](w) = \text{op}(v_1, v_2, \ldots, v_n)$.

commutativity. This reluctance to assume additional properties is also practically motivated: operators such as median : $\text{MSet}(\mathbb{R}) \to \mathbb{R}$ take a single argument as input, operators such as subtraction, $- : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ are not associative, and even the implicit concatenation operator of DReX, $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$ is non-commutative.

We therefore have to review some implicit assumptions in combinators such as split and iter. Consider the definition of $[\![\text{iter}(e)]\!](w)$. The input string w is divided into patches $w = w_1 w_2 \cdots w_n$, such that $v_i = [\![e]\!](w_i)$ is defined for each i. These values are then combined to produce the result, $[\![\text{iter}(e)]\!](w) = v_1 \cdot v_2 \cdots v_n$. See figure 1.2.

fold$(c, e, \textbf{op})$**.** In particular, when writing $\text{iter}(e, \text{op})$, we assume that op is associative and admits an identity element ϵ, so that binary concatenation $\cdot : \Gamma^* \times \Gamma^* \to \Gamma^*$ can be freely extended to arbitrary arities, $\cdot : \Gamma^* \times \Gamma^* \times \cdots \times \Gamma^* \to \Gamma^*$. A combinator inspired by the "`fold`" operation from functional programming is somewhat more general than $\text{iter}(e)$: Let $e : \Sigma^* \rightsquigarrow \Gamma_e$ be a QRE which maps streams $w \in \Sigma^*$ to values $[\![e]\!](w) : \Gamma_e$, $c : \Gamma$ be an initial value, and $\text{op} : \Gamma \times \Gamma_e \to \Gamma$ be an operation over the cost domains. Then $\text{fold}(c, e, \text{op})$ is a QRE whose semantics are defined as follows. Given an input stream w, it is divided into chunks, $w = w_1 w_2 \cdots w_n$, such that $v_i = [\![e]\!](w_i)$ is defined for each i. Then, each $i \in \{0, 1, 2, \ldots, n\}$, define a_i as follows:

$$
\begin{aligned}
a_0 &= c, \\
a_1 &= \text{op}(a_0, v_1), \\
a_2 &= \text{op}(a_1, v_2), \\
&\cdots \\
a_n &= \text{op}(a_{n-1}, v_n).
\end{aligned}
$$

Then, $[\![\text{fold}(c, e, \text{op})]\!](w) = a_n$. See figure 1.3.

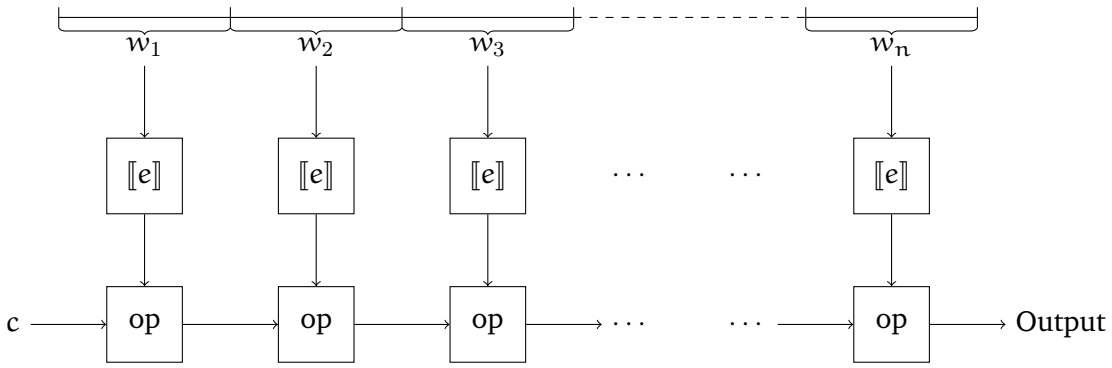


Figure 1.3: The semantics of fold(c, e, op). The input stream is divided into pieces, $w = w_1 w_2 \cdots w_n$, such that $v_i = [\![e]\!](w_i)$ is defined for each i. Starting with the initial value c, the results of subsequent applications of e are "threaded through" using the operator op.

Values vs. terms. A large variety of functions can be expressed using the fold combinator, and it relaxes the assumption that op be associative, but still assumes that op is binary, and requires a "neutral" element c. To work around this, the key idea behind QREs is to make expressions output *terms*, rather than concrete values. Consider, for example, the expressions

$$e_1 = a^n \mapsto n, \text{ and}$$
$$e_2 = a^n \mapsto \min(p, n).$$

Pick the input stream $w = a^{534}$. The first expression e_1 simply returns the length of the stream, $[\![e_1]\!](w) = 534$. On the other hand, the second expression e_2 returns the *syntactic object* "$\min(p, 534)$", where p is an integer-valued parameter. Parameters can be thought of as holes in the evaluation tree, to be filled in later either by the programmer, or some other QRE. For example, if we define

$$e_3 = a^n \mapsto 5, \text{ and}$$
$$e_4 = e_2\{p := e_3\},$$

then the expression e_4 evaluates both e_2 and e_3 on the entire input stream w, and substitutes the value of e_3 into the parameter p in the value returned by e_2. On our input stream of choice, $w = a^{534}$, we have:

$$\begin{aligned}
&[\![e_3]\!](w) = 5, \text{ so that} \\
&[\![e_4]\!](w) = [\![e_2]\!](w)[p := [\![e_3]\!](w)] \\
&\qquad = (\min(p, 534))[p := 5] \\
&\qquad = \min(5, 534) \\
&\qquad = 5.
\end{aligned}$$

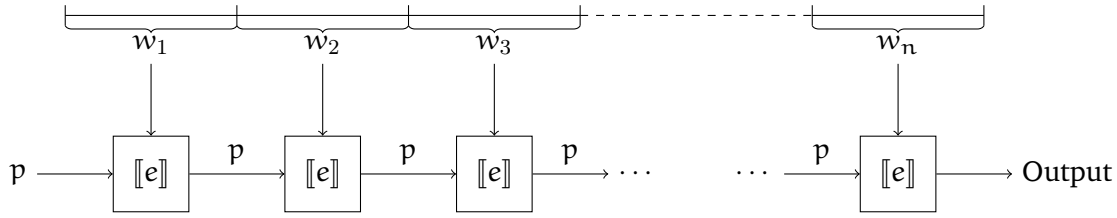


Figure 1.4: The semantics of fold(c, e, op). The input stream is divided into pieces, $w = w_1 w_2 \cdots w_n$, such that $v_i = [\![e]\!](w_i)$ is defined for each i. Starting with the initial value c, the results of subsequent applications of e are "threaded through" using the operator op.

iter$(e \to p)$ **and split**$(e \to^p f)$**.** Once we accept that QREs return representations of syntactic objects called terms, it is relatively straightforward to to devise domain- and operator-agnostic versions of the split and iter combinators. Let $e : \Sigma^* \rightsquigarrow T$ be a QRE[1] which maps input streams $w \in \Sigma^*$ to output terms, $[\![e]\!](w)$, of type T. Then, the expression iter$(e \to p)$ also transforms input streams $w \in \Sigma^*$ into output terms $[\![\text{iter}(e \to p)]\!](w)$ of type T.

The semantics of iter$(e \to p)$ are somewhat similar to fold. The main idea is that intermediate results are accumulated using term substitution rather than by applying any specific operator op. Concretely, the input stream w is first divided into patches $w = w_1 w_2 \cdots w_n$, such that $t_i = [\![e]\!](w_i)$ is defined for each i. We then iteratively define the following sequence of terms:

$$\begin{aligned} t_{a,0} &= p, \\ t_{a,1} &= t_1[p := t_{a,0}], \\ t_{a,2} &= t_2[p := t_{a,1}], \\ &\cdots \\ t_{a,n} &= t_n[p := t_{a,n-1}]. \end{aligned}$$

Finally, we define $[\![\text{iter}(e \to p)]\!](w) = t_{a,n}$. See figure 1.4. Both simpler versions of iter, iter(e, op), and fold(c, e, op) can be desugared into this more general combinator (we will show the explicit desugaring in chapter 2 in examples 2.19 and 2.20).

Similarly, the more general version of the concatenation operator is of the form split$(e \to^p f)$. Here $e : \Sigma^* \rightsquigarrow T_e$, and $f : \Sigma^* \rightsquigarrow T_f$ produce terms of type T_e and T_f respectively. The parameter p is of type T_e, and the intention is to substitute the result t_e produced by T_e into the parameter p of the term t_f produced by T_f. Concretely, let w be the input stream, which can be divided into two pieces, $w = w_e w_f$ such that $t_e = [\![e]\!](w_e)$ and $t_f = [\![f]\!](w_f)$ are both

[1]We intentionally choose a notation $e : \Sigma^* \rightsquigarrow T$ for QREs similar to that used to describe functions, $f : T \to T'$. However, we pick a different arrow, $\rightsquigarrow$, because (*a*) the object actually doing the transformation is $[\![e]\!]$, (*b*) for QREs, the objects produced are actually of type Terms(T), and (*c*) (at least early on,) in chapter 2, $[\![e]\!] : \Sigma^* \times \text{Terms}(T)$ will actually be a relation rather than a function.

$$
\begin{array}{lcll}
e, f, \ldots & ::= & \varphi \mapsto \lambda \mid \epsilon \mapsto t \mid \text{bot} & \text{Basic expressions} \\
& | & e \text{ else } f & \text{Regular combinators} \\
& | & \text{split}(e \to^p f) \mid \text{split}(e \leftarrow^q f) & \\
& | & \text{iter}(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k) & \\
& | & \text{iter}(p_1 \leftarrow e_1, p_2 \leftarrow e_2, \ldots, p_k \leftarrow e_k) & \\
& | & \text{op}(e_1, e_2, \ldots, e_k) & \text{Cost operations} \\
& | & e\{p \coloneqq f\} & \text{Substitution}
\end{array}
$$

Figure 2.9: The syntax of QREs. (Repeated from page 34.)

defined. Then

$$[\![\text{split}(e \to^p f)]\!](w) = t_f[p \coloneqq t_e].$$

As before, the simpler instances of the split combinator, $\text{split}(e, f, \text{op})$ can be desugared to use this more general combinator, as we will see in example 2.18.

See figure 2.9 for the final definition of QREs. As a result of generalizing QREs to output terms rather than concrete values, the combinators split into two separate classes: the parsing combinators such as split, iter and else, and the cost combinators such as $\text{op}(e, f)$ and $e[p \coloneqq f]$. The remaining operators such as $\text{split}(e, f, \text{op})$, $\text{iter}(e, \text{op})$, and $\text{fold}(e, a_0, \text{op})$ are just special instances of our more general split- and iteration-combinators, as we will show in chapter 2. We therefore have a framework which is completely agnostic of instance-specific details such as the cost domain, and the number and arity of individual cost operations.

Streaming composition. Say that the medical sensor occasionally produces out-of-scale readings due to physical limitations, and the manufacturer recommends that all measurements which are numerically greater than 100 be discarded. Rather than rewrite all our previous queries, it is simpler to construct a "predicate QRE" e_0:

$$
\begin{aligned}
e_0 &= \varphi(x) \mapsto x, \text{ where} \\
\varphi(x) &= (x = B) \vee (x = E) \vee (x = M(t, v) \wedge v \leqslant 100).
\end{aligned}
$$

Next, the expression

$$e_{0s} = \text{split}(\Sigma^*, e_0, \pi_2)$$

maps a non-empty stream $w = w'a$ of stream elements to its final value a iff a satisfies the property being tested, and is otherwise undefined.

Consider a stream of elements w, as shown in figure 1.6. When e_{0s} is applied to progressively longer prefixes of w, the sequence of values produced corresponds to exactly those elements which satisfy φ. The QRE e_{0s} therefore behaves like a filter for φ.

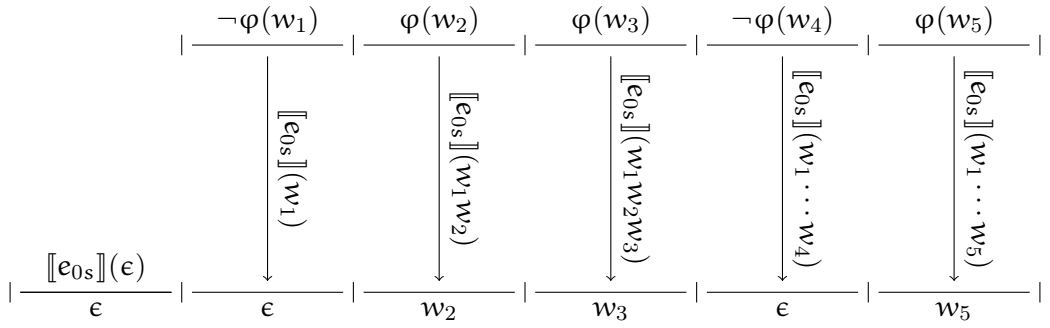


Figure 1.6: The operation of e_{0s} on progressively longer prefixes of w. The expression e_{0s} emits the last element of $w = w'a$ iff a satisfies φ. The sequence of outputs, $[\![e_{0s}]\!](\epsilon) \cdot [\![e_{0s}]\!](w_1) \cdot [\![e_{0s}]\!](w_1w_2) \cdots [\![e_{0s}]\!](w_1w_2 \cdots w_n)$, with ϵ replacing every occurrence of the undefined element $\perp$, consists of only those elements of w which satisfy φ.

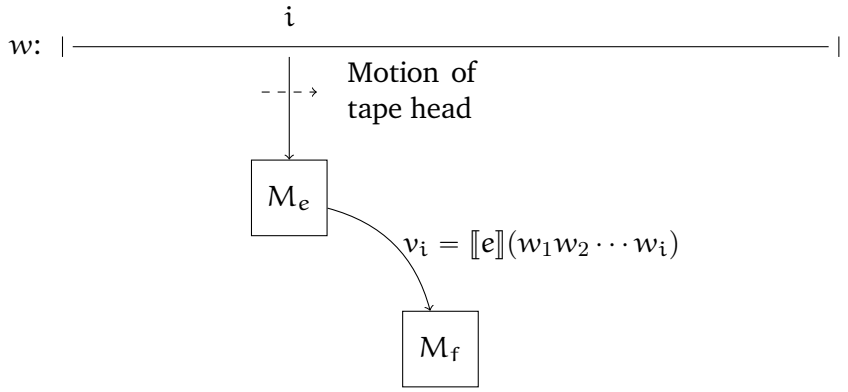


Figure 2.12: Visualizing the streaming composition operation, $e \gg f$. (Repeated from page 42.)

The streaming composition combinator, "$\gg$", allows us to access this sequence of outputs. Given expressions e and f, where f accepts a stream of elements each of which is of the return type of e, $e \gg f$ conceptually makes one left-to-right pass over the input stream w, examining each prefix of w in increasing order of length. Whenever e is defined, the result is fed to the second sub-expression f. See figure 2.12.

Recall that our goal was to ignore out-of-scale readings emitted by the medical sensor, and we wrote the QRE e_{0s} to do the filtering. If e is the actual query of interest, then this is achieved the expression $e' = e_{0s} \gg e$.

While it is not part of the basic QRE calculus of figure 2.9, we have found streaming composition to be an extremely useful operation in some of our later work [20]. Unlike the remaining operators, there is more wide-spread support for this construct outside the text-parsing ecosystem. Long processing pipelines are a natural programming abstraction, it is the primary building block in stream-processing systems such as Apache Storm [1], and also finds application in synchronous programming languages.

1.2 Contributions

In this thesis, we propose DReX expressions and QREs as a practical and theoretically well-understood framework to express functions over data streams. We are easily able to express several non-trivial queries, there is a **fast one-pass evaluation algorithm** for consistent function expressions, and for many numerical queries, we can maintain intermediate results as approximate terms to control memory requirements. QREs and DReX expressions are expressively equivalent to **regular cost functions** and **regular string transformations** respectively. Both classes have several equivalent representations, appealing closure properties, and traditional static analysis problems such as precondition computation and equivalence-checking are decidable for regular string transformations. We are not aware of a prior characterization of these classes of functions by a set of combinators analogous to the characterization of regular languages by regular expressions.

Both DReX and QREs are built around a small core: queries are modular, and the combinators are simple to explain. This makes programming in these formalisms simple, and raises the possibility of new and improved forms of programmer support. This includes better static analysis and debugging tools, and potential for automatic parallelization and query optimization. In comparison, most previous string-manipulating programs do not have a small core calculus, making this kind of programmer assistance difficult or even impossible to provide.

We can broadly divide our technical contributions into two parts, those related to expressiveness, and those related to evaluation algorithms. We will now motivate and describe our results in each of these directions.

1.2.1 Expressiveness

Regular string transformations. What is the class of functions expressible as QREs and as DReX expressions? Given that our formalisms are inspired by regular expressions, we look

at similar results from regular language theory. The study of regular languages has been one of the most prominent successes in computer science: Traditional regular expressions are easy-to-understand, can be efficiently parsed, and are powerful enough for a variety of simple text-processing tasks. Regular languages have robust closure properties, and most analysis questions such as equivalence and language inclusion are decidable. They have multiple equivalent representations, including: (*a*) operational models such as finite automata, (*b*) descriptive characterizations such as regular expressions, and (*c*) as formulas in monadic second-order (MSO) logic.

Therefore, a natural starting point for our expressiveness results is the class of "finite automata extended with outputs", i.e. the class of finite-state transducers. It can be readily seen that unlike in the simpler setting of acceptors, *two-way* finite-state transducers are strictly more powerful than their one-way counterparts (the function $w \mapsto \text{reverse}(w)$ is a simple function in this gap). While the post-image of a regular language under such a transformation need not be regular [6], they are closed under composition [37] and the equivalence of *deterministic* two-way finite state transducers is decidable [64].

Courcelle extended the idea of MSO-definable acceptors to *transformations* defined by logical formulas in MSO [38]. Engelfriet and Hoogeboom later showed that the class of transformations expressible as deterministic two-way finite state transducers coincides with functions definable in MSO [51], and labelled this the class of *regular string transformations*.

Streaming string transducers (SSTs). So far, we have described two equivalent characterizations of regular string transformations, first as the operational model of two-way deterministic finite-state transducers, and second, as transformations defined by logical formulas in MSO. In a sequence of papers [9, 10], Alur and Černý introduced the equivalent one-way operational model of streaming string transducers (SSTs), and this is the characterization which we will use in this thesis.

An SST is a finite-state machine which reads the input string w in one left-to-right pass, but maintains multiple registers to store partially computed chunks of the output which are updated and combined to produce the final result. See figure 3.2.

The expressive power of DReX. Having defined regular string transformations as those which can be implemented by SSTs, our goal in part I of this thesis is the following pair of theorems, which establish that SSTs and DReX expressions are expressively equivalent:

Theorem 4.1 (Closure)**.** *If Σ is a finite input alphabet, then:*

1. *If e is a consistent DReX expression, then we can construct an SST M which computes $[\![e]\!]$.*
2. *If e is a consistent QRE, then we can construct an SSTT M which computes $[\![e]\!]$.*

Theorem 5.1 (Completeness)**.**

1. *For every SST M, we can construct an equivalent consistent DReX expression e.*
2. *For every SSTT M, we can construct an equivalent consistent, well-typed, and single-use QRE e.*

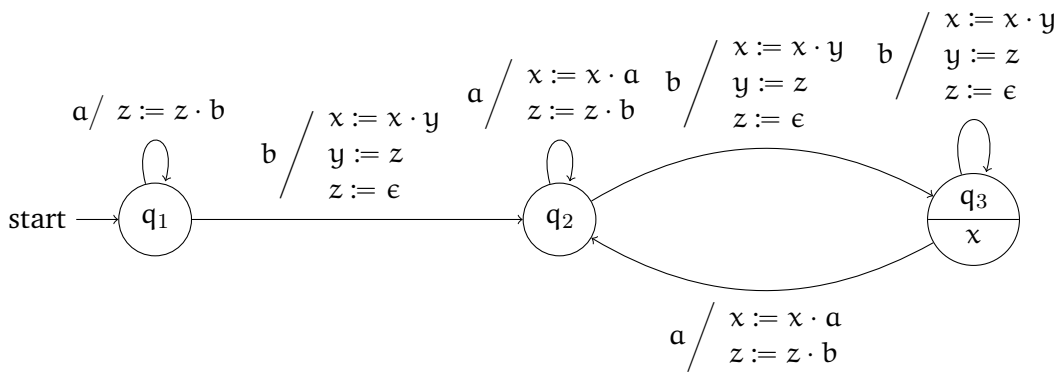


Figure 3.2: An example of a streaming string transducer (SST). We name this machine M_2. The details of the function being computed are presently unimportant. Observe that the machine has a finite set of control states, $Q = \{q_1, q_2, q_3\}$, and a finite set of registers $V = \{x, y, z\}$. Each register holds an intermediate result, and registers are combined and updated during transitions. The state q_3 is the only accepting state, and when the machine finishes execution in this state, the value of register x is emitted as output. We will formally define SSTs in chapter 3. (Repeated from page 47.)

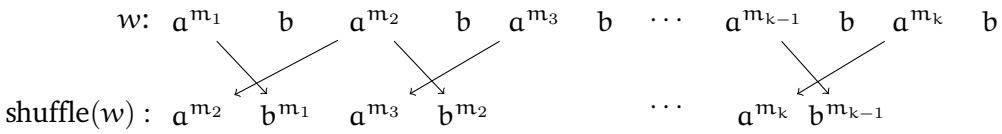


Figure 2.3: Defining shuffle(w). (Repeated from page 23.)

The closure theorem, which converts DReX expressions into equivalent SSTs is relatively straightforward, and mostly follows the traditional regular-expression-to-NFA translation algorithm. The completeness theorem is the more challenging claim to prove.

So far, we have described four operators in the DReX calculus: else, split, iter, and combine. We conjecture that the function "shuffle" defined as follows:

$$\text{shuffle}(a^{m_1} b a^{m_2} b \cdots a^{m_k} b) = a^{m_2} b^{m_1} a^{m_3} b^{m_2} \cdots a^{m_k} b^{m_{k-1}}$$

is inexpressible using only these four combinators. See figure 2.3. The rough intuition is that each of these operators first divides the input into disjoint patches, and then maps each of these patches into disjoint substrings of the output, in the same order as in the input. The function shuffle does not have this property, and is therefore unlikely to be expressible using these operators alone.

On the other hand, shuffle *is* implementable by an SST: in fact, this is the function implemented by the machine M_2 we saw in figure 3.2. One important technical insight in

this thesis is in the identification of an additional operator "chain" which allows the proof of theorem 5.1 to go through.

chain(e, r)**.** We will now briefly describe the chained sum combinator, chain. Let e be a DReX expression, and r be a regular expression. Then chain(e, r) is a DReX expression, and it operates as follows. Given an input string w, the chained sum expression first divides it into patches, $w = w_1 w_2 \cdots w_n$, where each patch $w_i \in \Sigma^*$ matches the regular expression r. The expression chain(e, r) then applies the sub-expression e to each pair of adjacent patches $w_i w_{i+1}$ and glues the results together to obtain the result:

$$[\![\text{chain}(e, r)]\!] = [\![e]\!](w_1 w_2)[\![e]\!](w_2 w_3) \cdots [\![e]\!](w_{n-1} w_n)$$

In the case of shuffle, $r = a^*b$, and e maps strings of the form $a^{m_1}ba^{m_2}b$ to $a^{m_2}b^{m_1}$. See figure 2.3. We will formally define chain in chapter 2. The motivation behind the combinator is two-fold: first, we believe that shuffle is inexpressible using the remaining combinators, and second, the operation emerges naturally as an idiom during the proof of theorem 5.1. The final construction of a DReX expression from an SST in chapter 5 will involve a detailed analysis of the pattern of data flows between the registers of the given SST.

A theory of regularity for numerical functions. To prove properties about the expressive power of QREs, we would like a model of functions from streams to numerical quantities which naturally generalizes the previous notion of regular string transformations. Weighted automata [49] are the best known of the various automaton models which map streams to numerical values.

Unfortunately, weighted automata are limited to two operations which form a semiring. This is insufficient for our purposes, because we would like to express a richer class of functions, simultaneously involving multiple arithmetic operations such as $+$, min and max, and multiset operations such as avg and median.

We therefore use the more recent concept of *regular cost functions* [13]. This model is parameterized by the set of operations Ops: there can be any number of operations, and each operation op $\in$ Ops can be of arbitrary arity. Regular cost functions coincide with MSO-definable *string-to-term* transformations, and are also captured by the one-pass operational model of streaming string-to-term transducers (SSTTs). SSTTs are very similar to SSTs: they have a finite set of control states Q, and a finite set of registers V to hold intermediate results of computation, except that the registers now hold terms rather than concrete values. We will define SSTTs in chapter 3, and just as the case of SSTs for regular string transformations, SSTTs will be our characterization of choice for regular cost functions. Paralleling the claim that DReX and SSTs are equi-expressive, in chapter 5, we will show that QREs and SSTTs can express the same class of functions.

1.2.2 The Expression Evaluation Problem

The most important computational problem associated with QREs and DReX expressions is that of evaluation: given an expression e and an input stream w, compute $[\![e]\!](w)$.

A straightforward approach is to "operationalize" the semantics, by using dynamic programming and evaluating each sub-expression of e on each sub-stream of w. Unfortunately, this algorithm requires $O(|w|^3)$ time, and does not scale to streams longer than a few thousand elements. Ideally, we would like an evaluation algorithm which makes a single left-to-right pass over the input stream, and processes each symbol of w in $\text{poly}(|e|)$ time.

Because of the analogy between DReX expressions (split, else, iter, etc.) and regular expressions (concatenation, union, Kleene-*, etc.), another approach is to construct an transducer for evaluation similar to the automaton built to match strings against regular expressions. Unfortunately, in this approach, the expression $\text{combine}(e, f)$ involves the product construction, and repeated invocations of the product construction results in a transducer whose size is exponential in the expression size $|e|$.

Bounding the memory usage. Consider a pair of states, $q_e \in Q_e$ and $q_f \in Q_f$, of each of the component machines when constructing the product. While the product machine has significantly more states in total than each of the component machines, $|Q| = |Q_e||Q_f|$, each product state can itself be represented in a much smaller amount of space, $|(q_e, q_f)| \approx |q_e| + |q_f|$, where the notation $|q|$ indicates the amount of space required to encode the representation of a state. Therefore, instead of explicitly constructing this transducer with an exponentially large state space, we build function "evaluators", which are stateful machines which lazily (and implicitly) traverse the states of the underlying transducer.

However, the underlying automaton is still a non-deterministic machine. To match a string against an NFA in one-pass, we have to simultaneously maintain all eventualities, i.e. the set

$$Q_w = \{q \in Q \mid q_0 \to^w q\}$$

of all states q which are reachable from the initial state q_0 along some path labelled w. Each element $q \in Q_w$ corresponds to a different *potential* parse tree of the entire string ww', where w' is the (still unseen) suffix. Memory usage, and hence processing time, is determined by the size of Q_w, where $|Q_w| \leqslant |Q|$. Therefore, even if we do not explicitly construct the product machine for $\text{combine}(e, f)$, executing an implicit non-deterministic product machine would require memory exponential in the expression size, $|\text{combine}(e, f)|$.

Expression consistency. Therefore, the main challenge is to find a subset of DReX and QREs which does not sacrifice expressiveness, but still permits fast and memory efficient evaluation algorithms, in particular by restricting the ability to express complicated anti-patterns, such as with combine, which has the flavour of language intersection:

$$[\![\text{combine}(e, f)]\!](w) = [\![e]\!](w) \cdot [\![f]\!](w)$$
$$w \in [\![r_1 \cap r_2]\!] \iff w \in [\![r_1]\!] \wedge w \in [\![r_2]\!]$$

We call this restricted set the *consistent* fragment, obtained by imposing additional constraints on the domains of the sub-expressions of each combinator. For example, for $\text{combine}(e, f)$ to be consistent, we require that $\text{Dom}(e) = \text{Dom}(f)$. The domain of the entire expression,

$\mathrm{Dom}(\mathrm{combine}(e, f))$ is then equal to the domain of each sub-expression and the operator cannot express language intersection. For $\mathrm{split}(e, f)$, we require that $\mathrm{Dom}(e)$ and $\mathrm{Dom}(f)$ are unambiguously concatenable. In the case of the choice combinator, e else f, we require $\mathrm{Dom}(e)$ and $\mathrm{Dom}(f)$ to be disjoint. Consistency is closely linked to the notion of unambiguous regular expressions, and also ensures that QREs and DReX expressions produce at most a single output string for each input, i.e. that the expression encodes a *partial function* rather than an unrestricted relation.

Consistency can be checked in $\mathrm{poly}(|e||\Sigma|)$ time if the input stream is drawn from a finite alphabet Σ. The consistency rules are simple to state and do not sacrifice expressive power. Most natural QREs and DReX expressions are already consistent, indicating that all streams must be unambiguously parsed, and do not pose a significant burden on the programmer. Notably, the few times we wrote non-consistent expressions, the problem turned out to be a mistake in the expression, rather than a limitation of the expressions we could write.

For the consistent fragment, we are able to provide a fast, memory-efficient one-pass evaluation algorithm: given an expression e and an input stream $w = w_1 w_2 \cdots w_n$, each symbol w_i of the stream is processed in $O(|e|^2)$ time, and during this processing, the algorithm produces the value of $[\![e]\!](w_1 w_2 \cdots w_i)$.

Informally, each expression evaluator keeps track of potential parse trees of w as multiple *threads*, and updates the threads on reading each input symbol. The key technical challenge is to limit the number of threads to some function of the expression size, but *independent* of the length of the input stream $|w|$. We exploit the consistency requirements to achieve this by establishing a correspondence between the active threads of each evaluator and the states of the underlying NFA.

Size of the output. The output of a DReX expression, $[\![e]\!](w)$ may be comparable in size to the input: consider for example the identity function, $\mathrm{id} = \mathrm{iter}(\mathrm{true} \mapsto x)$, or the string reversal function, $\mathrm{rev} = \text{left-iter}(\mathrm{true} \mapsto x)$. At the very least, the evaluator needs to keep this output in memory, and its memory usage therefore depends on the size of the output. In particular, we define the value

$$s_l(e, w) = \max\{|[\![e']\!](w')| \mid e' \text{ sub-expression of } e, \text{ and } w' \text{ substring of } w\}, \qquad (1.2.1)$$

as a bound on the size of the largest intermediate result produced during evaluation. We will similarly define a quantity $s_t(e, w)$ for QRE evaluation. The main technical contributions of chapter 6 are QRE / DReX evaluation algorithms which satisfy the following properties.

Theorem 6.1. *1. Let e be a consistent DReX expression, and $w = w_1 w_2 \cdots w_n$ be a sequence of symbols. There is an algorithm which makes one pass over w, and processes each symbol w_i in $O(|e|^2)$ time. During this processing, it will also output the value of $[\![e]\!](w_1 w_2 \cdots w_i)$, if it is defined. While reading the first i symbols, the algorithm will consume less than $O(|e| s_l(e, w_1 w_2 \cdots w_i))$ memory.*

2. Let e be a consistent, well-typed, single-use QRE, and $w = w_1 w_2 \cdots w_n$ be a sequence of input symbols. There is an algorithm which makes one pass over w, and processes

each symbol w_i in $O(|e|^2)$ time. During this processing, it will also output the value of $[\![e]\!](w_1 w_2 \cdots w_i)$, if it is defined. While reading the first i symbols, the algorithm will consume less than $O(|e| s_t(e, w_1 w_2 \cdots w_i))$ memory.

Term compression and approximation algorithms. Observe that the QRE evaluation algorithm in theorem 6.1 requires $O(|e| s_t(e, w))$ memory, where $s_t(e, w) = O(|e||w|)$ is the size of the largest intermediate term produced while evaluating e on w. For some output domains, such as strings, it is impossible to do significantly better than this bound.

However, when the expression only involves arithmetic operations, $+$, min, max, and avg, we are able to "compress" intermediate terms so that $s_t(e, w) \leqslant \text{poly}(|e|)$, independent of the length of the stream w.

Another class of queries for which $s_t(e, w)$ cannot be improved is when the expression involves quantile queries, such as median, or select_k. Here, $\text{select}_k(A)$, for $0 \leqslant k \leqslant 1$, is the $k|A|$-th smallest element of A. It is a well-known lower bound that computing $\text{median}(A)$ or $\text{select}_k(A)$ in one pass over the elements of A requires $|A|$ space, and similar lower bounds on exact computation have motivated the study of approximation algorithms for data streams.

In the case of QREs, when the expression involves a bounded number of quantile queries, and the programmer provides an error tolerance $\epsilon > 0$, we are able to reduce $s_t(e, w)$ to $\text{poly}(\epsilon^{-1}|e| \log(|w|))$. In this case, we guarantee that the result we produce, v, and the true result $\hat{v}$ are related as $(1 - \epsilon)\hat{v} \leqslant v \leqslant (1 + \epsilon)\hat{v}$. This result is particularly interesting because it suggests that we can incorporate continuing developments from the streaming algorithms literature, which is focused on finding time- and memory-efficient algorithms for specific problems, into our framework, which addresses the orthogonal problem of conveniently expressing a large class of simple queries.

Experimental performance. We implemented our evaluation and consistency-checking algorithms and evaluated them on several text transformations: deletion of comments from a program, insertion of quotes around words, tag extraction from XML documents, reversing dictionaries, and the reordering and aligning of misplaced fields in BibTeX files. The evaluation algorithm for consistent DReX scales to large inputs (less than for 8 seconds for 100,000 characters), while the dynamic programming algorithm, due to the cubic complexity in the size of the input, does not scale in practice (more than 60 seconds for 5,000 characters) and therefore has limited applicability. Finally, the consistency-checking algorithm is very fast in practice (less than 0.6 seconds for programs of size $\approx$ 3,600 sub-expressions), and it is also very helpful in identifying sources of ambiguity in the implemented programs.

1.3 Thesis Outline

We begin by defining DReX and QREs in chapter 2. While this chapter is necessarily formal, we will also include some pedagogical examples to improve readability.

Part I is devoted to our expressiveness claims. We define regular cost functions in chapter 3. Our definitions are operational: regular string transformations are those that can be

implemented by an SST, and regular cost functions are those that can be implemented using an SSTT. In chapter 4, we describe the translation from consistent function expressions to transducer models, and in chapter 5, we describe the translation from transducer models to function expressions.

Part II presents the fast evaluation algorithm for consistent QREs. The algorithm itself is described in chapter 6. We consider issues related to term compression and approximate query evaluation in chapter 7, and present an experimental evaluation of the framework in chapter 8.

We conclude by summarizing the related work in chapter 9, and discuss potential areas of future research in chapter 10.

Chapter 2

Syntax and Semantics

In this chapter, we will define the basic expressions and combinators that make up the QRE / DReX framework. We will begin by defining DReX expressions in section 2.1, including the important idea of expression consistency. In section 2.2, we will motivate the definition of QREs by writing some representative expressions, and outline our primitive notions—types, operations, and terms—in section 2.3. In section 2.4, we will define QREs: the development of ideas will closely follow the pattern used for DReX, but will involve some additional considerations beyond expression consistency, particularly well-typedness and single-usage. In section 2.5, we will construct a few example QREs, and finally introduce the streaming composition combinator in section 2.6.

2.1 DReX

DReX is a formalism to express mappings between input strings $w \in \Sigma^*$, and an output monoid $(\mathbb{D}, \cdot, 1_{\mathbb{D}})$ [19]. The most important instantiation is where $\mathbb{D} = \Gamma^*$, the set of strings over an output alphabet Γ. In this case, DReX can be used to express string transformations [14], similar to tools such as sed, AWK and Perl. Some simple numerical queries can also be expressed using the monoid $(\mathbb{R}, +, 0)$. In this section, we will formally define the syntax and semantics of DReX expressions. Each expression e encodes a relation, $[\![e]\!] \subseteq \Sigma^* \times \mathbb{D}$, and we will write "$e : \Sigma^* \rightsquigarrow \mathbb{D}$" to succinctly indicate the input and output domains.

Basic expressions. Pick a character $a \in \Sigma$, and designate an output $d \in \mathbb{D}$. There are three basic expression constructors: (*a*) $a \mapsto d$, which maps the input string a to the constant output d, (*b*) $\epsilon \mapsto d$, which maps the empty string ϵ to the output d, and (*c*) bot, which is undefined everywhere.

$$[\![a \mapsto d]\!] = \{(a, d)\}, \tag{2.1.1}$$

$$[\![\epsilon \mapsto d]\!] = \{(\epsilon, d)\}, \text{ and} \tag{2.1.2}$$

$$[\![\text{bot}]\!] = \varnothing. \tag{2.1.3}$$

Choice. For each pair of expressions e, f, the expression e else f non-deterministically applies either e or f. This is the straightforward analogue of the union operator in regular expressions.

$$\begin{aligned}[\![e \text{ else } f]\!] &= \{(w, d) \mid (w, d) \in [\![e]\!] \text{ or } (w, d) \in [\![f]\!]\} \\ &= [\![e]\!] \cup [\![f]\!]. \end{aligned} \tag{2.1.4}$$

Concatenation. The expression split(e, f) splits the input stream w into two parts, $w = w_e w_f$, and applies e to the first part, and f to the second.

$$[\![\text{split}(e, f)]\!] = \{(w_e w_f, d_e \cdot d_f) \mid (w_e, d_e) \in [\![e]\!], (w_f, d_f) \in [\![f]\!]\}. \tag{2.1.5}$$

The sister-expression left-split(e, f) is similar, except for the order in which d_e and d_f are combined:

$$[\![\text{left-split}(e, f)]\!] = \{(w_e w_f, d_f \cdot d_e) \mid (w_e, d_e) \in [\![e]\!], (w_f, d_f) \in [\![f]\!]\}. \tag{2.1.6}$$

Clearly, split and left-split overlap for the special case of commutative monoids. In more general cases, such as the the cost domain of strings under concatenation, the two expressions are different.

Iteration. Given an input stream w, iter(e) breaks it up into pieces $w = w_1 w_2 \cdots w_n$ such that e is defined for each w_i. The results d_1, d_2, …, d_n are then glued back together to form the output, $d_1 \cdot d_2 \cdots d_n$. Formally, $[\![\text{iter}(e)]\!]$ is the smallest set defined by the following rules:

1. $(\epsilon, 1_{\mathbb{D}}) \in [\![\text{iter}(e)]\!]$, where $1_{\mathbb{D}}$ is the identity element of the monoid $\mathbb{D}$.
2. Whenever $(w, d) \in [\![\text{iter}(e)]\!]$, and $(w_e, d_e) \in [\![e]\!]$, $(ww_e, d \cdot d_e) \in [\![\text{iter}(e)]\!]$.

Similar to left-split, left-iter(e) differs only in the direction in which concatenation is performed. Formally, $[\![\text{left-iter}(e)]\!]$ is the smallest set defined by the following rules:

1. $(\epsilon, 1_{\mathbb{D}}) \in [\![\text{left-iter}(e)]\!]$.
2. Whenever $(w, d) \in [\![\text{left-iter}(e)]\!]$, and $(w_e, d_e) \in [\![e]\!]$, $(ww_e, d_e \cdot d) \in [\![\text{left-iter}(e)]\!]$.

Example 2.1. If the expression echo_Σ maps each individual character to itself, then the expression id $=$ iter(echo_Σ) encodes the identity function over all strings. The expression rev $=$ left-iter(echo_Σ) represents the symmetric string reversal function. △

Function combination. The expression combine(e, f) concatenates the results of e and f, each applied to the entire string w.

$$[\![\text{combine}(e, f)]\!] = \{(w, d_e \cdot d_f) \mid (w, d_e) \in [\![e]\!] \text{ and } (w, d_f) \in [\![f]\!]\}. \tag{2.1.7}$$

In example 2.1, we expressed the identity function, id, by applying the iter combinator to the single-character echo echo_Σ. The string-copy expression copy $=$ combine(id, id) maps each input string w to the output ww.

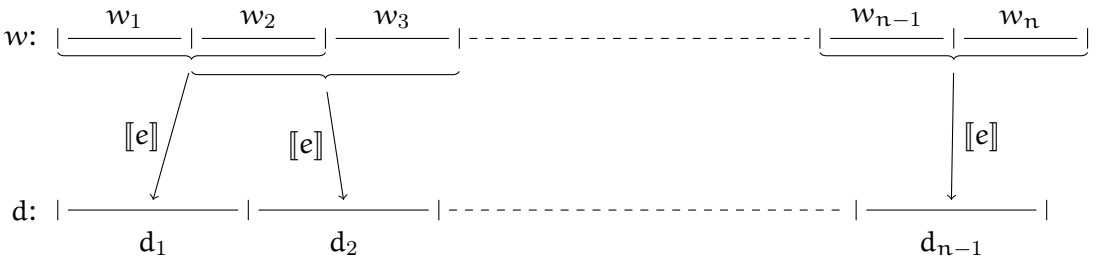


Figure 2.1: The semantics of chain(e, r). The input string w is broken into pieces $w = w_1 w_2 \cdots w_n$, where each substring $w_i \in [\![r]\!]$. The sub-expression e is applied to each pair of subsequent chunks $w_i w_{i+1}$, and the results are concatenated to produce the final output.

Chained sum. The final operator is the chained sum, which allows us to "mix" outputs produced by different parts of the input string. Given e, and a regular expression r, chain(e, r) breaks the input string w into pieces, $w = w_1 w_2 \cdots w_n$, such that: (*a*) $n \geqslant 2$, (*b*) for each i, $w_i \in [\![r]\!]$, and (*c*) for each i, there exists $d_i \in \mathbb{D}$ such that $(w_i w_{i+1}, d_i) \in [\![e]\!]$. Then, w is mapped to the output $d_1 \cdot d_2 \cdots d_{n-1}$. See figure 2.1. This is a new operator, without a traditional regular expression counterpart. The motivation is two-fold: first, we believe that shuffle (which we will see in example 2.6) is inexpressible using the remaining operators, and second, the operation naturally emerges as an idiom during the proof of theorem 5.1.

Formally, $[\![\text{chain}(e, r)]\!]$ is the closure of the following rules:

1. For each pair of strings, $w_1 \in [\![r]\!]$ and $w_2 \in [\![r]\!]$, and each $d \in \mathbb{D}$, if $(w_1 w_2, d) \in [\![e]\!]$, then:

   $$(w_1 w_2, d) \in [\![\text{chain}(e, r)]\!].$$

2. For all strings w_1, w_2, w_3, and for each pair of values $d, d_e \in \mathbb{D}$, if $(w_1 w_2, d) \in [\![\text{chain}(e, r)]\!]$, $(w_2 w_3, d_e) \in [\![e]\!]$, $w_1 \in [\![r^*]\!]$, $w_2 \in [\![r]\!]$, and $w_3 \in [\![r]\!]$, then:

   $$(w_1 w_2 w_3, d \cdot d_e) \in [\![\text{chain}(e, r)]\!].$$

The symmetric expression left-chain(e, r) is defined by the following rules:

1. For each pair of strings, $w_1 \in [\![r]\!]$ and $w_2 \in [\![r]\!]$, and each $d \in \mathbb{D}$, if $(w_1 w_2, d) \in [\![e]\!]$, then:

   $$(w_1 w_2, d) \in [\![\text{left-chain}(e, r)]\!].$$

2. For all strings w_1, w_2, w_3, and for each pair of values $d, d_e \in \mathbb{D}$, if $(w_1 w_2, d) \in [\![\text{left-chain}(e, r)]\!]$, $(w_2 w_3, d_e) \in [\![e]\!]$, $w_1 \in [\![r^*]\!]$, $w_2 \in [\![r]\!]$, and $w_3 \in [\![r]\!]$, then:

   $$(w_1 w_2 w_3, d_e \cdot d) \in [\![\text{left-chain}(e, r)]\!].$$

Symbolic predicates and character transforms. Consider echo_Σ, the single-character echo. Over a finite alphabet, $\Sigma = \{a_1, a_2, \ldots, a_k\}$, echo_Σ can be expressed as:

$$\text{echo}_\Sigma = a_1 \mapsto a_1 \text{ else } a_2 \mapsto a_2 \text{ else } \cdots \text{ else } a_k \mapsto a_k.$$

We will regularly consider large alphabets, such as Unicode ($\approx$ 128,000 characters and growing), and even potentially infinite alphabets, such as the set of all measurements produced by a sensor. In these settings, this approach of explicitly handling each character does not scale. We therefore extend the basic expressions to allow properties over values to be described symbolically using predicates. Our formalization here is derived from the recently proposed model of symbolic transducers [94].

Over each alphabet Σ, pick a non-empty (and potentially even infinite) collection of predicates Φ_Σ. Each predicate is a computable function $\varphi : \Sigma \to \text{Bool}$. We also choose a collection of basic character transformations, Λ. Each character transform is a computable function, $\lambda : \Sigma \to \Gamma$. For each predicate φ and character transform λ, $\varphi \mapsto \lambda$ applies λ to those characters $a \in \Sigma$ such that $\varphi(a) = \text{true}$:

$$[\![\varphi \mapsto \lambda]\!] = \{(a, \lambda(a)) \mid a \in \Sigma, \text{ and } \varphi(a) = \text{true}\}. \tag{2.1.8}$$

We require:

1. that the satisfiability of predicates be decidable, and
2. that Φ_Σ be effectively closed under boolean operations: $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \neg\varphi_1 \in \Phi_\Sigma$, for each $\varphi_1, \varphi_2 \in \Phi_\Sigma$.

We will assume that predicate evaluation, boolean combination, and satisfiability checking can all be performed in $O(1)$ time. The chosen predicates Φ_Σ divide Σ into equivalence classes called MINTERMS: two values $a, b \in \Sigma$ are equivalent if $\varphi(a) = \varphi(b)$, for all $\varphi \in \Phi_\Sigma$. We refer to the set of minterms of Φ_Σ as $\text{Minterms}(\Phi_\Sigma)$.

We summarize the syntax of DReX combinators in figure 2.2. The interpretation of regular expressions is standard: each regular expression r is associated with a language $[\![r]\!] \subseteq \Sigma^*$, defined inductively as follows: (*a*) $[\![\varphi]\!] = \{a \in \Sigma \mid \varphi(a) = \text{true}\}$, (*b*) $[\![\epsilon]\!] = \{\epsilon\}$, (*c*) $[\![\bot]\!] = \varnothing$, (*d*) $[\![r_1 + r_2]\!] = [\![r_1]\!] \cup [\![r_2]\!]$, (*e*) $[\![r_1 \cdot r_2]\!] = [\![r_1]\!] \cdot [\![r_2]\!]$, and (*f*) $[\![r^*]\!] = [\![r]\!]^*$.

Example 2.2. Common examples of character predicates and transformations include the classification of letters into upper- and lower-case variants, $\text{isUpperCase}(x)$ and $\text{isLowerCase}(x)$, and the conversion functions $\text{toUpperCase}(x)$ and $\text{toLowerCase}(x)$.

We previously expressed the identity function as $\text{id} = \text{iter}(\text{echo}_\Sigma)$, where echo_Σ is the single-character echo. Over an arbitrary alphabet, this function can be expressed as $\text{echo}_\Sigma = \text{true} \mapsto x$. Several variations of this expression are also useful: $\text{iter}(\text{isLowerCase}(x) \mapsto \text{toUpperCase}(x))$ maps sequences of lower-case letters to their upper-case counterparts. The identity function restricted to strings not containing a space is given by $\text{id}_{\neg\,\text{isSpace}} = \text{iter}(\neg\,\text{isSpace}(x) \mapsto x)$. More interesting functions can be constructed using the conditional

$$
\begin{array}{rcll}
e, f, \ldots & ::= & \varphi \mapsto \lambda \mid \epsilon \mapsto d \mid \text{bot} & \text{Transformations} \\
& \mid & e \text{ else } f & \\
& \mid & \text{split}(e, f) \mid \text{left-split}(e, f) & \\
& \mid & \text{iter}(e) \mid \text{left-iter}(e) & \\
& \mid & \text{chain}(e, r) \mid \text{left-chain}(e, r) & \\
& \mid & \text{combine}(e, f) & \\
r & ::= & \varphi \mid \epsilon \mid \bot & \text{Regular expressions} \\
& \mid & r_1 + r_2 \mid r_1 \cdot r_2 \mid r^* &
\end{array}
$$

Figure 2.2: The syntax of DReX.

operator: Define:

$$
\begin{aligned}
\text{swCase} &= \text{swCase}_1 \text{ else } \text{swCase}_2, \text{ where} \\
\text{swCase}_1 &= \text{isUpperCase}(x) \mapsto \text{toLowerCase}(x), \text{ and} \\
\text{swCase}_2 &= \text{isLowerCase}(x) \mapsto \text{toUpperCase}(x).
\end{aligned}
$$

The expression swCase flips the case of a single input character, and so iter(swCase) switches the case of each character in the input string. △

Example 2.3 (Minterms)**.** Consider the set of integers $\mathbb{Z}$, and the basic predicates:

$$
\begin{aligned}
p_0(x) &= \text{“}x \equiv 0 \pmod 3\text{”}, \\
p_1(x) &= \text{“}x \equiv 1 \pmod 3\text{”}, \text{ and} \\
p_2(x) &= \text{“}x \equiv 2 \pmod 3\text{”}.
\end{aligned}
$$

The boolean closure of these 3 basic predicates, $\Phi_{\mathbb{Z}} = \{\text{false}, p_0, p_1, p_2, p_0 \vee p_1, p_1 \vee p_2, p_2 \vee p_0, \text{true}\}$ contains 8 elements, while there are just 3 minterms, corresponding to the predicates p_0, p_1 and p_2. The number of minterms, $|\text{Minterms}(\Phi_\Sigma)|$, is therefore often much smaller than the number of predicates, $|\Phi_\Sigma|$, and is useful in measuring the computational complexity of our algorithms. △

Example 2.4. Mangling filenames is a common operation performed on the command line. Consider the function strip, which maps full filenames, such as “`/home/user/file.txt`”, to just the location, “`/home/user`”. We wish to drop all characters starting from the last occurrence of “/”. The following expression performs this operation:

$$
\begin{aligned}
\text{strip} &= \text{split}(\text{id}, \texttt{"/"} \mapsto \epsilon, \text{drop}_{\neg\texttt{"/"}}), \text{ where} \\
\text{drop}_{\neg\texttt{"/"}} &= \text{iter}(x \neq \texttt{"/"} \mapsto \epsilon)
\end{aligned}
$$

maps all strings not containing an occurrence of “/” to ϵ. △

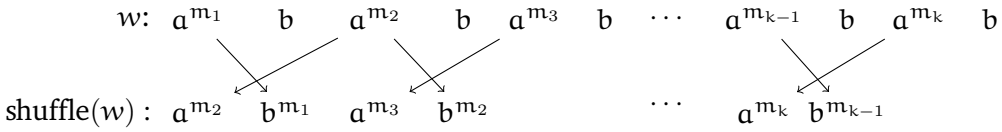


Figure 2.3: Defining shuffle(w).

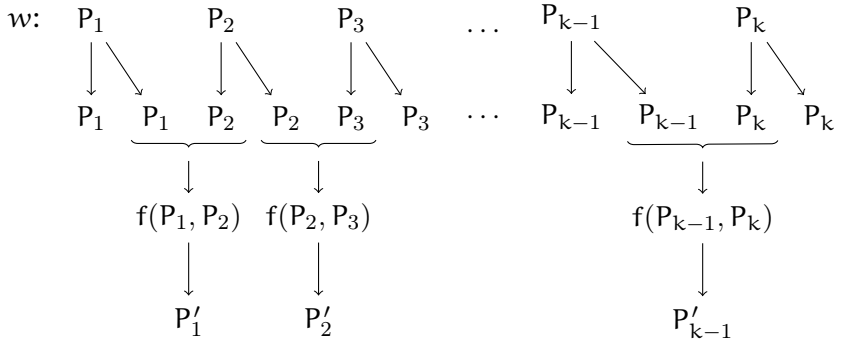


Figure 2.4: Expressing shuffle using function combinators. Each patch P_i is a a string of the form a^*b.

Example 2.5. Given a string of the form "*First-Name Last-Name*", the expressions

$$\text{echoFirst} = \text{split}(\text{id}_{\neg\,\text{isSpace}}, \text{isSpace}(x) \mapsto \epsilon, \text{iter}(\text{true} \mapsto \epsilon)), \text{ and}$$
$$\text{echoLast} = \text{split}(\text{iter}(\text{true} \mapsto \epsilon), \text{isSpace}(x) \mapsto \epsilon, \text{id}_{\neg\,\text{isSpace}})$$

output "*First-Name*" and "*Last-Name*" respectively. The two expressions can be combined into combine(echoLast, echoFirst), which outputs "*Last-NameFirst-Name*". Note that the space in between is omitted—the expression

$$\text{swap} = \text{combine}(\text{split}(\text{echoLast}, \epsilon \mapsto \text{" "}), \text{echoFirst})$$

preserves this space. △

Example 2.6 (shuffle)**.** Consider the function shuffle : $\Sigma^* \to \Sigma^*$ from figure 2.3. If $\Sigma = \{a, b\}$ and $w = a^{m_1}ba^{m_2}b \cdots a^{m_k}b$, with $k \geqslant 2$, then $\text{shuffle}(w) = a^{m_2}b^{m_1}a^{m_3}b^{m_2} \cdots a^{m_k}b^{m_{k-1}}$.

We will now express shuffle using the chained sum combinator: we illustrate the main idea in figure 2.4. We first divide the input w into patches P_i, each of the form a^*b, and the output into patches P'_i, each of the form a^*b^*. Each input patch P_i should be scanned twice, first to produce the a-s in P'_{i-1}, and then again to produce the b-s in P'_i. The pattern of the input patches is given by $r = a^*b$. Consider the expression:

$$f = \text{left-split}(\text{split}(\text{iter}(a \mapsto b), b \mapsto \epsilon), \text{split}(\text{iter}(a \mapsto a), b \mapsto \epsilon)).$$

Given two consecutive input patches $P_i P_{i+1}$, f produces the output patch P'_i. It follows that shuffle = chain(f, r). △

Example 2.7. DReX can also be used to write simple queries in output domains other than the set of strings. Consider the situation of a customer who frequents a coffee shop. Each cup of coffee he purchases costs \$2, but if he fills out a survey, then all cups of coffee purchased that month cost only \$1 (including cups already purchased). Here $\Sigma = \{C, S, M\}$ denoting respectively the purchase of a cup of coffee, completion of the survey, and the passage of a calendar month. The expression:

$$\begin{aligned} m &= m_{\neg S} \text{ else } m_S, \text{ where} \\ m_{\neg S} &= \text{iter}(C \mapsto 2), \text{ and} \\ m_S &= \text{split}(\text{iter}(C \mapsto 1), S \mapsto 0, \text{iter}(C \mapsto 1 \text{ else } S \mapsto 0)) \end{aligned}$$

maps the events of a single month to the customer's debt. The expression:

$$\text{coffee} = \text{split}(\text{iter}(\text{split}(m, M \mapsto 0)), m)$$

maps the entire purchase history of the customer to the amount he needs to pay the store. △

2.1.1 Expression consistency

We have defined DReX expressions as denoting relations, rather than (partial) functions. A single input string *w* could thus be mapped to multiple (possibly even infinitely many) outputs. This is troublesome for the following reasons: (*a*) All our examples naturally denote functions, such as id, swap, and shuffle. An expression with multiple outputs is a code smell. (*b*) We would ideally like to solve the "function evaluation" problem: find *the* value of $[\![e]\!](w)$, given an expression *e* and an input string *w*. Therefore, we will now define the class of "consistent" DReX expressions: the central objects of our study. The main idea is to make the definitions of the choice, split sum and iteration combinators unambiguous.

Unambiguity. Two languages L_1, L_2 are UNAMBIGUOUSLY CONCATENABLE if for all strings $u, u' \in L_1$, $v, v' \in L_2$, if $uv = u'v'$, then $u = u'$ and $v = v'$ (see figure 2.5). A language L is UNAMBIGUOUSLY ITERABLE if for all strings $u_1, u_2, \ldots, u_m \in L$, $v_1, v_2, \ldots, v_n \in L$, if $u_1 u_2 \cdots u_m = v_1 v_2 \cdots v_n$, then $m = n$, and $u_i = v_i$ for all i. UNAMBIGUOUS regular expressions are inductively defined as follows:

1. φ, ϵ, and $\perp$ are all unambiguous,
2. $r_1 + r_2$ is unambiguous if:
   (a) r_1 and r_2 are both unambiguous, and
   (b) $[\![r_1]\!] \cap [\![r_2]\!] = \varnothing$,
3. $r_1 \cdot r_2$ is unambiguous if:

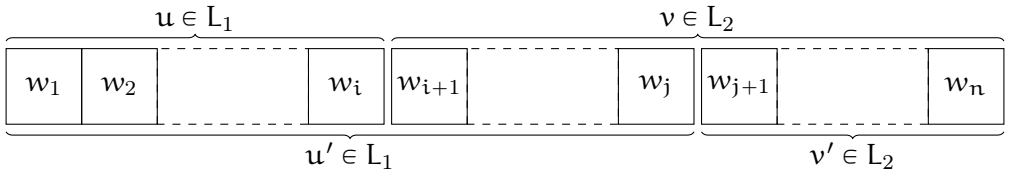


Figure 2.5: L_1 and L_2 are unambiguously concatenable if for all $u, u' \in L_1$ and $v, v' \in L_2$, if $uv = u'v'$, then $u = u'$ and $v = v'$. Equivalently, L_1 and L_2 are unambiguously concatenable iff there is no string $w \in L_1 L_2$ with multiple parse trees, as shown in this figure.

   (a) r_1 and r_2 are both unambiguous, and
   (b) $[\![r_1]\!]$ and $[\![r_2]\!]$ are unambiguously concatenable.

4. r^* is unambiguous if r is unambiguous and $[\![r]\!]$ is unambiguously iterable.

Example 2.8. The regular expression $a \cdot a^*$ is unambiguous: the only way to parse a matched string is to split after the first symbol. The regular expression $a^* \cdot a^*$ is a simple example of a regular expression which is not unambiguous. Next observe that the languages $[\![a \cdot a^*]\!]$ and $[\![b \cdot b^*]\!]$ are disjoint. Therefore $a \cdot a^* + b \cdot b^*$ is unambiguous. On the other hand, both a^* and b^* match the empty string, and so $a^* + b^*$ is not unambiguous. △

Remark 2.9. Unambiguity is a property of the regular expression rather than that of the underlying language. For example, the regular expressions a^*, $a^* \cdot a^*$, and $(a^*)^*$ all denote the same language,

$$[\![a^*]\!] = [\![a^* \cdot a^*]\!] = [\![(a^*)^*]\!] = \{a^n \mid n \in \mathbb{N}\},$$

but only a^* is unambiguous. In fact, every regular language can be represented by an unambiguous regular expression—as we will point out in section 5.1—and this observation is central to our construction in chapter 5.

Consistency rules. Given a DReX expression e, let $\mathrm{Dom}(e) = \{w \mid \exists d, (w, d) \in [\![e]\!]\}$ be its domain. CONSISTENT expressions are inductively defined as follows.

1. The base expressions $\varphi \mapsto \lambda$, $\epsilon \mapsto d$, and bot are always consistent.
2. e else f is consistent if:
   (a) both sub-expressions e and f are consistent, and
   (b) $\mathrm{Dom}(e) \cap \mathrm{Dom}(f) = \varnothing$.
3. $\mathrm{split}(e, f)$ and $\text{left-split}(e, f)$ are consistent if:
   (a) both sub-expressions e and f are consistent,

(b) $\mathrm{Dom}(\mathrm{f}) \neq \varnothing$, and

(c) $\mathrm{Dom}(e)$ and $\mathrm{Dom}(\mathrm{f})$ are unambiguously concatenable.

4. iter(e) and left-iter(e) are consistent if:

   (a) e is consistent, and

   (b) $\mathrm{Dom}(e)$ is unambiguously iterable.

5. chain(e, r) and left-chain(e, r) are consistent if:

   (a) e is consistent and r is unambiguous,

   (b) $\mathrm{Dom}(e) = [\![\mathrm{r} \cdot \mathrm{r}]\!]$, and

   (c) $[\![\mathrm{r}]\!]$ is unambiguously iterable.

6. combine(e, f) is consistent if:

   (a) both sub-expressions e and f are consistent, and

   (b) $\mathrm{Dom}(e) = \mathrm{Dom}(\mathrm{f})$.

For split(e, f) and left-split(e, f), observe that one of the consistency requirements is that $\mathrm{Dom}(\mathrm{f}) \neq \varnothing$: This requirement will later turn out to be convenient (but not crucial) for the proof of lemma 6.10, which claims that the evaluation algorithm is correct for these combinators. In any case, this is not a severe restriction because for all expressions $\mathrm{f}_\perp$ with $\mathrm{Dom}(\mathrm{f}_\perp) = \varnothing$, split$(e, \mathrm{f}_\perp)$ and left-split$(e, \mathrm{f}_\perp)$ are both equivalent to bot.

Theorem 2.10, proved by structural induction on e, is a straightforward consequence of the semantics and the above consistency rules.

Theorem 2.10. *If e is a consistent DReX expression, and $w \in \Sigma^*$ such that $(w, \mathrm{d}) \in [\![e]\!]$ and $(w, \mathrm{d}') \in [\![e]\!]$, then $\mathrm{d} = \mathrm{d}'$. In other words, $[\![e]\!] : \Sigma^* \to \mathbb{D}$ is a partial function.*

Complexity of consistency checking. The following theorem establishes the complexity of consistency checking, and its proof will follow from proposition 2.12.

Theorem 2.11. *The consistency of a DReX expression e can be determined in time poly$(|e|$ $|\mathit{Minterms}(\Phi_\Sigma)|)$.*

The main subroutines required to establish consistency are checking unambiguity of regular expressions and determining their equivalence. The complexities of these sub-tasks are established by the following results.

Proposition 2.12. *1. Whether a given regular expression r is unambiguous is decidable in time poly$(|\mathrm{r}||\mathit{Minterms}(\Phi_\mathrm{r})|)$.*

2. Given two unambiguous regular expressions r_1 and r_2, whether $[\![\mathrm{r}_1]\!] = [\![\mathrm{r}_2]\!]$ is decidable in time poly$(|\mathrm{r}_1||\mathrm{r}_2||\mathit{Minterms}(\Phi_\mathrm{r})|)$ [86].

Proof sketch. Consider the standard regular-expression-to-NFA translation algorithm [85]. To determine whether a given regular expression is unambiguous, we perform additional checks at each step of the construction. Given two NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$, their disjointedness can be determined in polynomial time by the product construction. Unambiguous concatenability and unambiguous iterability can also be determined in polynomial time by the product construction of the NFA with itself.

To prove that two regular expressions r_1 and r_2 are equivalent, it is sufficient to prove that: (*a*) $[\![\mathrm{r}_1]\!] \cap [\![\mathrm{r}_2]\!] = [\![\mathrm{r}_1]\!]$, and (*b*) $[\![\mathrm{r}_1]\!] \cap [\![\mathrm{r}_2]\!] = [\![\mathrm{r}_2]\!]$. For a given language L, let $\#_{\mathrm{L}}(\mathrm{l})$ be the number of strings of length l:

$$\#_{\mathrm{L}}(\mathrm{l}) = |\{w \in \mathrm{L} \mid |w| = \mathrm{l}\}|.$$

Stearns and Hunt [86] present a polynomial time procedure based on difference equations to determine $\#_{\mathrm{L}}(\mathrm{l})$ where L is specified by an unambiguous NFA. They then bound the length l_0 of the smallest string that can potentially exist in $[\![\mathrm{r}_1]\!]$ but not in $[\![\mathrm{r}_1]\!] \cap [\![\mathrm{r}_2]\!]$ to a polynomial in the NFA size. The equivalence check then reduces to checking for each potential length $\mathrm{l} \in \{0, 1, \ldots, \mathrm{l}_0\}$, whether $\#_{[\![\mathrm{r}_1]\!]}(\mathrm{l}) = \#_{[\![\mathrm{r}_2]\!]}(\mathrm{l}) = \#_{[\![\mathrm{r}_1]\!] \cap [\![\mathrm{r}_2]\!]}(\mathrm{l})$. □

2.2 Towards More General Cost Domains

DReX can already be used to express simple numerical queries over strings. For instance, in example 2.7, we wrote a query to obtain the total amount of coffee purchased by the customer. However, there are many natural queries which cannot be expressed using DReX. An example is the query which computes the *average* amount of coffee the customer drinks each month. This is because we have assumed that the cost domain is a monoid with a single operation—in this case addition—and split, iter and combine automatically apply this operation to compute their outputs. Our first step is therefore to allow more general types and operations in the framework.

Example 2.13. Recall that the coffee-shop data stream was a sequence of tuples $w \in (\mathrm{C} + \mathrm{S} + \mathrm{M})^*$. We now wish to compute the the average number of cups of coffee that the customer purchases each month.

Given a sequence of events $w \in (\mathrm{C} + \mathrm{S})^*$, the expression $\#_{\mathrm{C}} = \mathrm{iter}(\mathrm{C} \mapsto 1 \text{ else } \mathrm{S} \mapsto 0, +)$ splits the stream into individual events, maps each C to the constant 1 and each survey S to the constant 0. It then combines these results using the operator $+$ to obtain the amount of coffee purchased. The generalization is in allowing combinators such as $\mathrm{iter}(e, \mathrm{op})$ and $\mathrm{split}(e, \mathrm{f}, \mathrm{op})$ to accept an additional argument "op", which indicates how results from the sub-expressions are to be combined.

With this additional parameter, the rest of the average-coffee query is straightforward: First, we use $\#_{\mathrm{C}}$ as a sub-expression to map the events of a single month (streams of the form $(\mathrm{C} + \mathrm{S})^* \cdot \mathrm{M}$) to the total amount of coffee purchased:

$$\#^{\mathrm{M}}_{\mathrm{C}} = \mathrm{split}(\#_{\mathrm{C}}, \mathrm{M} \mapsto 0, +).$$

We finally write:

$$\#_C^{\text{avg}} = \text{iter}(\#_C^M, \text{avg})$$

to compute the average quantity of coffee that the customer drinks each month. △

Another common idiom is the fold operation from functional programming. Given an initial value c, a QRE e, and an operation op, each of appropriate type, $\text{fold}(c, e, \text{op})$ divides the input stream into sub-streams $w = w_1 w_2 \cdots w_n$, applies e to each part to obtain results $d_1, d_2, \ldots, d_n$, and outputs $\text{op}(\cdots(\text{op}(\text{op}(c, d_1), d_2), \ldots), d_n)$.

Example 2.14. Consider the transactions of a customer with a bank. The event stream w is of the form $(\mathbb{R} \cup \{M\})^* \cdot M$, where each number $d \in \mathbb{R}$ indicates the deposit / withdrawal of money from the account, and M indicates the end of a calendar month. The bank pays some interest, say 1%, at the end of each month. Our goal is to compute the final balance.

The first sub-problem is to summarize the transactions of each month: the expression

$$\delta = \text{split}(\text{iter}(d \in \mathbb{R} \mapsto d, +), M \mapsto 0, +)$$

computes the total amount deposited / withdrawn from the account given a stream w of the form $\mathbb{R}^*M$. Given a starting balance b, and a total deposit d, the function $(b, d) \mapsto 1.01b + d$ computes the closing balance after interest was applied. Now, the expression:

$$\text{bal} = \text{fold}(0, \delta, (b, d) \mapsto 1.01b + d)$$

computes the final balance, assuming that the account was initially empty. △

2.3 Preliminaries

The central idea behind QREs is that $\text{split}(e, f, \text{op})$, $\text{iter}(e, \text{op})$, and $\text{fold}(c, e, \text{op})$ are all special cases of expressions which map input streams to output *terms*, rather than concrete values. This is also a convenient way to handle non-associative operators, operators without an identity element, and even non-binary operations. A particularly appealing benefit will be the eventual separation, in figure 2.9, of the parsing combinators, split, iter, and else, from the cost operations.

Types and operations. Fix a universe $\mathcal{T}$ of TYPES, and a collection $\text{Ops} = \{\text{op}_1, \text{op}_2, \ldots\}$ of OPERATIONS. Each operation is a computable function of the form $\text{op} : T_1 \times T_2 \times \cdots \times T_k \to T$, for some $k \geqslant 1$, and types $T_1, T_2, \ldots, T_k, T$. We call k the ARITY of op.

Some examples of types are: (*a*) the sets $\mathbb{Z}$, $\mathbb{R}$, and $\text{Bool} = \{\text{true}, \text{false}\}$ of integers, real numbers, and booleans respectively, (*b*) the set $\text{MSet}(\mathbb{R})$ of finite multisets of real numbers, (*c*) finite alphabets such as $\Sigma_{ab} = \{a, b\}$, and (*d*) products $T \times U$, for $T, U \in \mathcal{T}$, and (*e*) sets of finite strings T^*, for $T \in \mathcal{T}$.

Examples of operations include: (*a*) addition, $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, (*b*) multiset operations such as union, $\cup : \text{MSet}(\mathbb{R}) \times \text{MSet}(\mathbb{R}) \to \text{MSet}(\mathbb{R})$, average, $\text{avg} : \text{MSet}(\mathbb{R}) \to \mathbb{R}$, and order

statistics such as $\min : \mathrm{MSet}(\mathbb{R}) \to \mathbb{R}$, $\max : \mathrm{MSet}(\mathbb{R}) \to \mathbb{R}$, and $\mathrm{select}_k : \mathrm{MSet}(\mathbb{R}) \to \mathbb{R}$, for $0 \leqslant k \leqslant 1$,[1] and (*c*) string operations such as concatenation, $\cdot : T^* \times T^* \to T^*$.

We will occasionally need to pick an arbitrary element from a type, such as when initializing or resetting the registers of an automaton. We therefore assume that each type is non-empty, and freely write statements such as, "choose an element d of type T".

Parameters and terms. Let $\mathcal{P} = \{p, q, \ldots\}$ be a set of PARAMETERS. We will associate each parameter p with a type T, and express this succinctly as $p : T$. For convenience, we assume that there is an infinite supply of parameters of each type. From constants c, parameters p, and operations op, we can naturally build TERMS, which are well-typed productions of the following grammar:

$$t \quad ::= \quad c \mid p \mid \mathrm{op}(t_1, t_2, \ldots, t_k) \tag{2.3.1}$$

For each pair of terms t_1 and t_2, and parameter p, such that both t_2 and p are of the same type, we write $t_1[p := t_2]$ for the term where every occurrence of p in t_1 is replaced with t_2. $\mathrm{Terms}(T)$ is the set of all terms of type T. We refer to the set of parameters occurring in t as $\mathrm{Param}(t)$.

Example 2.15. Consider the terms:

$$t_1 = p + \min(q + 2, 3), \text{ and}$$
$$t_2 = \min(p, r).$$

Then:

$$t_3 = t_1[p := t_2] = \min(p, r) + \min(q + 2, 3).$$

$\mathrm{Param}(t_1) = \{p, q\}$, $\mathrm{Param}(t_2) = \{p, r\}$, and $\mathrm{Param}(t_3) = \{p, q, r\}$. △

Single-use terms. Let p be an integer-valued parameter. Consider the term $t_1 = p + p$, and for each $i \in \mathbb{N}$, let $t_{i+1} = t_i[p := t_1]$.

$$\begin{aligned}
t_1 &= p + p = 2p \\
t_2 &= t_1[p := t_1] = (p + p) + (p + p) \\
t_3 &= t_2[p := t_1] = p + p + \cdots + p \qquad (8 \text{ times}) \\
&\cdots \\
t_n &= t_{n-1}[p := t_1] = p + p + \cdots + p \qquad (2^n \text{ times})
\end{aligned}$$

The term t_n was constructed in n steps, but encodes a term of size $O(2^n)$. If the terms used in a QRE are unrestricted, they may generate exponentially large outputs in the size of the input stream. For computational feasibility, we therefore restrict our attention to single-use

[1] Here $\mathrm{select}_k(A)$ is $(k|A|)$-th smallest element of A, so that $\mathrm{median}(A) = \mathrm{select}_{0.5}(A)$.

expressions. A term t is SINGLE-USE if every parameter occurs at most once in t. If t and t' are single-use terms, then $|t[p := t']| \leqslant |t| + |t'|$.

In particular, $t = p + p$ is not a single-use term. On the other hand, all terms t_1, t_2, and t_3 from example 2.15 are single-use.

Remark 2.16. We will assume that:

1. all numerical constants can be stored in $O(1)$ space,
2. for each operator op, and appropriately typed terms t_1, t_2, ..., t_k, $\mathrm{op}(t_1, t_2, \ldots, t_k)$ can be constructed in $O(1)$ time,
3. given terms t_1, t_2 and a parameter p, $t_1[p := t_2]$ can be constructed in $O(1)$ time, and
4. that parameter-free terms of size n can be evaluated in $O(n)$ time.

2.4 Quantitative Regular Expressions

QREs are a generalization of DReX, first to rich cost domains such as numbers with min, max, and avg, and second, to produce terms over the cost domain, rather than concrete values [17].

To illustrate the difference between producing concrete values and terms as output, consider the expressions:

$$e_c = (a^n \cdot b) \mapsto n, \text{ and}$$
$$e_t = (a^n \cdot b) \mapsto \min(n, p).$$

Both expressions basically count the number of a-s in the input string. However, given the input string $a^{34} \cdot b$, the first expression e_c outputs the integer 34, while e_t produces the *syntactic object* $t =$ "$\min(34, p)$", where $p : \mathbb{Z}$ is an integer-valued parameter. When provided with a value to be substituted in place of p, say $p := 7$, t can be further evaluated to a specific integer: $t[p := 7] = \min(34, 7) = 7$.

Next, consider the query which counts the minimum number of a-s between subsequent occurrences of b. This query can be expressed as:

$$e_{mc} = \mathrm{iter}(e_c, \min).$$

Alternatively, it can be written as:

$$e_{mt} = \mathrm{iter}(e_t \to p).$$

Given an input string w, the expression e_{mt} first splits it into pieces $w = w_1 w_2 \cdots w_k$, where each substring w_i is of the form $a^{n_i} \cdot b$, and applies e_t to each piece, producing the sequence of intermediate terms $t_i = \min(n_i, p)$. The expression e_{mt} then threads these intermediate results together along the parameter p:

$$\begin{aligned}[\![e_{mt}]\!](w) &= t_k[p := t_{k-1}[p := \cdots]] \\ &= \min(n_k, p)[p := \min(n_{k-1}, p)[p := \cdots]] \\ &= \min(n_k, n_{k-1}, \ldots, n_1, p).\end{aligned}$$

The idea is that by pushing the operator "min" down from the iteration construct, $\text{iter}(\cdots, \min)$, to the underlying expression, $\text{iter}(\cdots \to \mathrm{p})$, the parsing combinators are made agnostic both of the cost domain and the peculiarities of individual cost operators, such as their arities, their identity elements, and whether they are commutative and / or associative.

The QRE evaluation problem—say of evaluating e_{t} on the string $a^3 4 \cdot \mathrm{b}$—is then that of producing this syntactic term object "$\min(34, \mathrm{t})$", or something identically equal (the terms $\min(\mathrm{t}, 34)$ and $\min(34, 97, \mathrm{t})$ are both acceptable). When the output term is parameter-free, it can be further simplified into a concrete numerical value.

Just as with DReX, we will initially introduce QREs as defining relations between input streams and output terms. The sub-class of *consistent* expressions will define partial functions from input streams to terms. In addition, we will define *well-typed* expressions, which always output well-typed terms. The typing rules statically eliminate nonsensical outputs such as "$\min(\mathrm{p}, \text{true}) + \text{false}$". Finally, we will define the class of *single-use* expressions, which will always output single-use terms, and have the pleasant property that $|[\![e]\!](w)| = O(|e||w|)$.

Basic expressions. The basic expressions are mostly identical to those in DReX: (*a*) $\varphi \mapsto \lambda$ applies λ to elements which satisfy φ, (*b*) $\epsilon \mapsto \mathrm{t}$ maps the empty input stream to the term t, and (*c*) bot identifies the empty relation.

Choice, cost operations, and term substitution. The expression e else f is also a straightforward generalization: the expression non-deterministically applies either e or f. The only difference is that the output produced is a term, rather than a concrete value.

In the DReX expression $\text{combine}(e, \mathrm{f})$, the cost operation "$\cdot$" used to combine the results of e and f was implicit. With QREs, we have to be more specific: for each cost operator op, the expression $\text{op}(e, \mathrm{f})$ produces $\text{op}([\![e]\!](w), [\![\mathrm{f}]\!](w))$. See figure 2.6. For example, $\max(e, \mathrm{f})$ produces the maximum of the results of e and f. The expression can naturally be generalized to operators op of arbitrary arity: $\text{op}(e_1, e_2, \ldots, e_k)$. This is the only class of operations which has knowledge of the output domain: all the other operators are concerned with parsing the input stream, and are agnostic of the cost operations permitted.

The expression $e\{\mathrm{p} := \mathrm{f}\}$ applies both e and f to the entire input stream and then produces the result $\mathrm{t}_e[\mathrm{p} := \mathrm{t}_{\mathrm{f}}]$. It is similar to $\text{op}(e, \mathrm{f})$, except that substitution is a meaningful operation for terms over *any* cost domain.

Remark 2.17. Substitution, $\mathrm{t}[\mathrm{p} := \mathrm{t}']$, is an operation over terms, and produces a new term where all occurrences of p in t are replaced with t'. In particular, it does not appear in the basic grammar of terms in equation 2.3.1. The operation should not be confused with the QRE substitution combinator, $e\{\mathrm{p} := \mathrm{f}\}$, which indicates that term substitution must be performed with the results of e and f at runtime. To highlight this difference, we use braces for the QRE combinator instead of square brackets.

Concatenation. We illustrate the concatenation operation in figure 2.7. Instead of providing an operation op with which to combine the intermediate results, we supply a parameter p: if e produces t_e and f produces t_{f}, then $\text{split}(e \to^{\mathrm{p}} \mathrm{f})$ produces $\mathrm{t}_{\mathrm{f}}[\mathrm{p} := \mathrm{t}_e]$. The symmetric

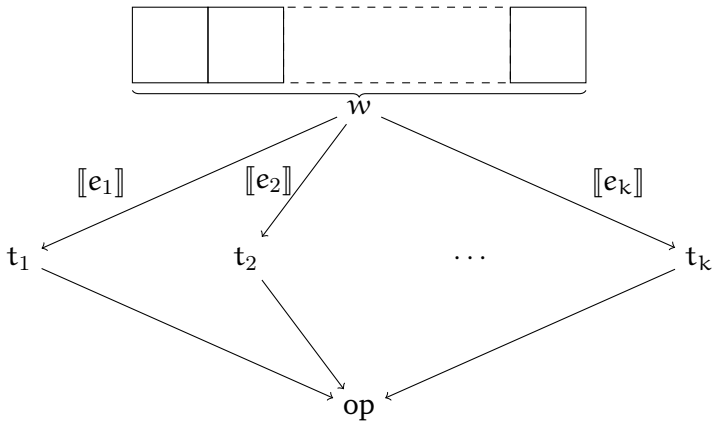


Figure 2.6: Defining $\mathrm{op}(e_1, e_2, \ldots, e_k)$. If for any i, $[\![e_i]\!](w)$ is undefined, then $[\![\mathrm{op}(e_1, e_2, \ldots, e_k)]\!](w)$ is also undefined.

expression $\mathrm{split}(e \leftarrow^q f)$ differs in the direction in which substitution is performed: the result is the term $t_e[q := t_f]$.

Example 2.18 ($\mathrm{split}(e, f, \mathrm{op})$)**.** Let T_e and T_f be the output types of e and f respectively. Choose a parameter $p : T_e$ which does not occur in f. Consider the expression:

$$f' = \mathrm{op}(\mathrm{Dom}(f) \mapsto p, f).$$

Given an input stream w, f' outputs $\mathrm{op}(p, t_f)$, where t_f is the output of f. We can use f' to desugar $\mathrm{split}(e, f, \mathrm{op})$ as follows:

$$\mathrm{split}(e, f, \mathrm{op}) = \mathrm{split}(e \rightarrow^p f').$$

Given the results t_e and t_f of e and f, the expression above outputs $t'_f[p := t_e] = \mathrm{op}(p, t_f)[p := t_e] = \mathrm{op}(t_e, t_f)$. △

Iteration. In contrast to DReX, which had two "iteration-like" constructs, iter and chain, we now only have a single generalized iteration operator. The first idea is to thread a parameter through the iterated expression, rather than have a top-level operator combine the results. In this sense, our grand-unified iteration combinator is very similar to the fold operation. The second key idea is to evaluate *multiple* expressions during each iteration, rather than just a single expression. We illustrate the behavior of $\mathrm{iter}(e_1 \rightarrow p_1, e_2 \rightarrow p_2)$ in figure 2.8.

Example 2.19 ($\mathrm{fold}(c, e, \mathrm{op})$)**.** Say that $c : T$, and let $p : T$ be an arbitrary parameter which does not occur in e. If $(w, t) \in [\![e]\!]$, then the expression:

$$e' = \mathrm{op}(\mathrm{Dom}(e) \mapsto p, e)$$

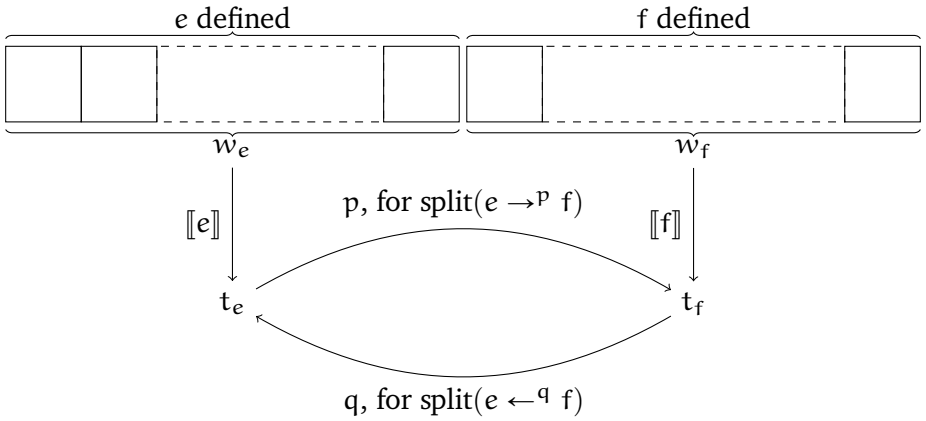


Figure 2.7: Defining split($e \to^p f$) and split($e \leftarrow^q f$). A pair of words w_e, w_f is found such that $w = w_e w_f$ and $[\![e]\!](w_e)$ and $[\![f]\!](w_f)$ are both defined. The direction in which the results are substituted then depends on the direction of the combinator used.

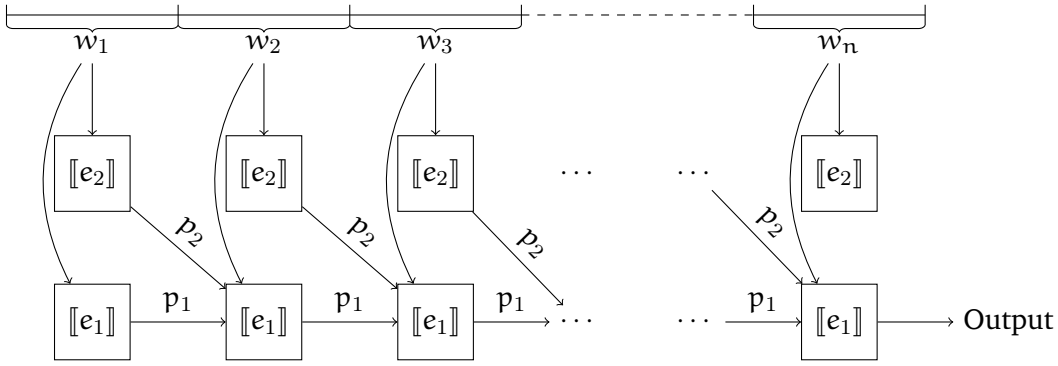


Figure 2.8: Semantics of the iter combinator. This figure describes the "right"-iteration operation, $r = \text{iter}(e_1 \to p_1, e_2 \to p_2)$. The input string w is split into chunks, w_1, w_2, ..., w_n, and $[\![e_1]\!]$ and $[\![e_2]\!]$ are applied to each chunk.

$e, f, \ldots$	$::=$	$\varphi \mapsto \lambda \mid \epsilon \mapsto t \mid \text{bot}$	Basic expressions
	$\mid$	e else f	Regular combinators
	$\mid$	$\text{split}(e \to^{p} f) \mid \text{split}(e \leftarrow^{q} f)$	
	$\mid$	$\text{iter}(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k)$	
	$\mid$	$\text{iter}(p_1 \leftarrow e_1, p_2 \leftarrow e_2, \ldots, p_k \leftarrow e_k)$	
	$\mid$	$\text{op}(e_1, e_2, \ldots, e_k)$	Cost operations
	$\mid$	$e\{p \coloneqq f\}$	Substitution

Figure 2.9: The syntax of QREs.

maps w to $\text{op}(p, t)$. Consider the expression:

$$f = \text{iter}(e' \to p).$$

If $w = w_1 w_2 \cdots w_n$, where $[\![e]\!](w_i) = d_i$, for each i, then f produces the result

$$\text{op}(\cdots(\text{op}(\text{op}(p, d_1), d_2), \ldots), d_n).$$

All that remains is to substitute the initial value of the parameter p:

$$\text{fold}(c, e, \text{op}) = f\{p \coloneqq (\text{Dom}(e)^* \mapsto c)\}. \qquad \triangle$$

Example 2.20 ($\text{iter}(e, \text{op})$)**.** In the expression $\text{iter}(e, \text{op})$, op is an arbitrary-arity operator over values of type T_e, or equivalently, a function from multisets $\text{MSet}(T_e)$ to values of type T, for some types T_e and T. The first expression:

$$e' = \text{fold}(\varnothing, \{e\}, \cup)$$

maps streams w to the multiset of results produced by iterating e along the stream. Then we can write:

$$\text{iter}(e, \text{op}) = \text{op}(e'). \qquad \triangle$$

QREs, formally defined. We summarize the syntax and semantics of quantitative expressions in figures 2.9 and 2.10 respectively.

Consistency rules. The consistency rules for QREs are identical to those we have already seen for DReX: see figure 2.1. We therefore also have similar consequences:

Theorem 2.21. *1. If e is a consistent QRE, and $w \in \Sigma^*$ such that $(w, t) \in [\![e]\!]$ and $(w, t') \in [\![e]\!]$, then $t = t'$.*

2. Whether a quantitative expression e is consistent can be determined in time poly$(|e|, |\mathit{Minterms}(\Phi_\Sigma)|)$.

$$\frac{\varphi(a) = \text{true}}{(a, \lambda(a)) \in [\![\varphi \mapsto \lambda]\!]}\ (\textsc{symb}) \qquad \frac{}{(\epsilon, t) \in [\![\epsilon \mapsto t]\!]}\ (\textsc{eps})$$

$$\frac{(w, t) \in [\![e]\!]}{(w, t) \in [\![e \text{ else } f]\!]}\ (\textsc{else1}) \qquad \frac{(w, t) \in [\![f]\!]}{(w, t) \in [\![e \text{ else } f]\!]}\ (\textsc{else2})$$

$$\frac{w = w_e w_f \qquad (w_e, t_e) \in [\![e]\!] \qquad (w_f, t_f) \in [\![f]\!]}{(w, t_f[p \coloneqq t_e]) \in [\![\text{split}(e \to^p f)]\!]}\ (\textsc{rsplit})$$

$$\frac{w = w_e w_f \qquad (w_e, t_e) \in [\![e]\!] \qquad (w_f, t_f) \in [\![f]\!]}{(w, t_e[q \coloneqq t_f]) \in [\![\text{split}(e \leftarrow^q f)]\!]}\ (\textsc{lsplit})$$

$$\frac{}{(\epsilon, p_1) \in [\![\text{iter}(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k)]\!]}\ (\textsc{rbase})$$

$$\frac{\begin{array}{c} w = w_{\text{pre}} w_{\text{post}} \\ (w_{\text{pre}}, t_1) \in [\![e_1]\!] \qquad (w_{\text{pre}}, t_2) \in [\![e_2]\!] \qquad \cdots \qquad (w_{\text{pre}}, t_k) \in [\![e_k]\!] \\ (w_{\text{post}}, t_{\text{post}}) \in [\![\text{iter}(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k)]\!] \\ t = t_{\text{post}}[p_1 \coloneqq t_1, p_2 \coloneqq t_2, \ldots, p_k \coloneqq t_k] \end{array}}{(w, t) \in [\![\text{iter}(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k)]\!]}\ (\textsc{rstep})$$

$$\frac{}{(\epsilon, p_1) \in [\![\text{iter}(p_1 \leftarrow e_1, p_2 \leftarrow e_2, \ldots, p_k \leftarrow e_k)]\!]}\ (\textsc{lbase})$$

$$\frac{\begin{array}{c} w = w_{\text{pre}} w_{\text{post}} \\ (w_{\text{pre}}, t_{\text{pre}}) \in [\![\text{iter}(p_1 \leftarrow e_1, p_2 \leftarrow e_2, \ldots, p_k \leftarrow e_k)]\!] \\ (w_{\text{post}}, t_1) \in [\![e_1]\!] \qquad (w_{\text{post}}, t_2) \in [\![e_2]\!] \qquad \cdots \qquad (w_{\text{post}}, t_k) \in [\![e_k]\!] \\ t = t_{\text{pre}}[p_1 \coloneqq t_1, p_2 \coloneqq t_2, \ldots, p_k \coloneqq t_k] \end{array}}{(w, t) \in [\![\text{iter}(p_1 \leftarrow e_1, p_2 \leftarrow e_2, \ldots, p_k \leftarrow e_k)]\!]}\ (\textsc{lstep})$$

$$\frac{(w, t_1) \in [\![e_1]\!] \qquad (w, t_2) \in [\![e_2]\!] \qquad \cdots \qquad (w, t_k) \in [\![e_k]\!]}{(w, \text{op}(t_1, t_2, \ldots, t_k)) \in [\![\text{op}(e_1, e_2, \ldots, e_k)]\!]}\ (\textsc{combine})$$

$$\frac{(w, t_e) \in [\![e]\!] \qquad (w, t_f) \in [\![f]\!]}{(w, t_e[p \coloneqq t_f]) \in [\![e\{p \coloneqq f\}]\!]}\ (\textsc{subst})$$

Figure 2.10: The semantics of QREs.

Table 2.1: Consistency rules for QREs.

e	... is CONSISTENT:
$\varphi \mapsto \lambda$ $\epsilon \mapsto t$ bot	always
e else f	if e and f are consistent and $\mathrm{Dom}(e) \cap \mathrm{Dom}(f) = \varnothing$.
split$(e \to^p f)$ split$(e \leftarrow^q f)$	if e and f are consistent, $\mathrm{Dom}(f) \neq \varnothing$ and $\mathrm{Dom}(e)$ and $\mathrm{Dom}(f)$ are unambiguously concatenable.
iter$(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k)$ iter$(p_1 \leftarrow e_1, p_2 \leftarrow e_2, \ldots, p_k \leftarrow e_k)$	if $\mathrm{Dom}(e_1) = \mathrm{Dom}(e_2) = \cdots = \mathrm{Dom}(e_k)$ is unambiguously iterable.
$\mathrm{op}(e_1, e_2, \ldots, e_k)$	if $\mathrm{Dom}(e_1) = \mathrm{Dom}(e_2) = \cdots = \mathrm{Dom}(e_k)$.
$e\{p \coloneqq f\}$	if $\mathrm{Dom}(e) = \mathrm{Dom}\, f$.

Well-typed expressions. Despite the presence of multiple data types in our formalism, we have so far ignored the problem of ill-typed expressions. For example, the expression $\max(a \mapsto 2, a \mapsto \text{true})$ is syntactically well-formed, and on the input stream a, it produces the output $\max(2, \text{true})$. To avoid the problem of such nonsensical outputs, we introduce a simple type system on QREs. Expressions e are associated with typing judgments of the form $e : \Sigma^* \rightsquigarrow T_{\text{out}}$, indicating that e maps input streams from Σ^* to well-typed output terms of type T_{out}. Consequently, an expression is WELL-TYPED if for some input alphabet Σ and output data type T_{out}, $e : \Sigma^* \rightsquigarrow T_{\text{out}}$. We summarize the typing rules in figure 2.11. The following theorem is a simple consequence of the typing rules and semantics.

Theorem 2.22. *If $e : \Sigma^* \rightsquigarrow T_{out}$ is a well-typed expression, and $(w, t) \in [\![e]\!]$, then $w \in \Sigma^*$ and $t \in$ Terms(T_{out}).*

Parameter support and single-use expressions. Let $e = a \mapsto p + p$. Consider the expression iter$(e \to p)$ and streams of the form a^n. If $v_n = [\![\text{iter}(a \mapsto p + p \to p)]\!](a^n)$, then the sequence of outputs formed is as follows:

$$
\begin{aligned}
v_0 &= p, \\
v_1 &= v_0[p \coloneqq [\![e]\!](a)] = p[p \coloneqq p + p] = p + p, \\
v_2 &= v_1[p \coloneqq [\![e]\!](a)] = (p + p)[p \coloneqq p + p] = 4p, \\
&\ldots \\
v_n &= v_{n-1}[p \coloneqq [\![e]\!](a)] = 2^n p.
\end{aligned}
$$

$$\frac{\varphi : \Sigma \to \mathsf{Bool} \quad \lambda : \Sigma \to \mathsf{T}_{\mathrm{out}}}{\varphi \mapsto \lambda : \Sigma^* \rightsquigarrow \mathsf{T}_{\mathrm{out}}}\ (\textsc{t-symb}) \qquad \frac{t : \mathsf{T}_{\mathrm{out}}}{\epsilon \mapsto t : \Sigma^* \rightsquigarrow \mathsf{T}_{\mathrm{out}}}\ (\textsc{t-eps})$$

$$\frac{}{\mathsf{bot} : \Sigma^* \rightsquigarrow \mathsf{T}_{\mathrm{out}}}\ (\textsc{t-bot}) \qquad \frac{e : \Sigma^* \rightsquigarrow \mathsf{T}_{\mathrm{out}} \quad f : \Sigma^* \rightsquigarrow \mathsf{T}_{\mathrm{out}}}{e\ \mathsf{else}\ f : \Sigma^* \rightsquigarrow \mathsf{T}_{\mathrm{out}}}\ (\textsc{t-else})$$

$$\frac{e : \Sigma^* \rightsquigarrow \mathsf{T}_e \quad p : \mathsf{T}_e \quad f : \Sigma^* \rightsquigarrow \mathsf{T}_f}{\mathsf{split}(e \to^{p} f) : \Sigma^* \rightsquigarrow \mathsf{T}_f}\ (\textsc{t-rsplit}) \qquad \frac{e : \Sigma^* \rightsquigarrow \mathsf{T}_e \quad q : \mathsf{T}_f \quad f : \Sigma^* \rightsquigarrow \mathsf{T}_f}{\mathsf{split}(e \leftarrow^{q} f) : \Sigma^* \rightsquigarrow \mathsf{T}_e}\ (\textsc{t-lsplit})$$

$$\frac{\begin{array}{cc} e_1 : \Sigma^* \rightsquigarrow \mathsf{T}_1 & p_1 : \mathsf{T}_1 \\ e_2 : \Sigma^* \rightsquigarrow \mathsf{T}_2 & p_2 : \mathsf{T}_2 \\ \multicolumn{2}{c}{\cdots} \\ e_k : \Sigma^* \rightsquigarrow \mathsf{T}_k & p_k : \mathsf{T}_k \end{array}}{\begin{array}{l} \mathsf{iter}(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k) : \Sigma^* \rightsquigarrow \mathsf{T}_1 \\ \mathsf{iter}(p_1 \leftarrow e_1, p_2 \leftarrow e_2, \ldots, p_k \leftarrow e_k) : \Sigma^* \rightsquigarrow \mathsf{T}_1 \end{array}}\ (\textsc{t-iter})$$

$$\frac{e_1 : \Sigma^* \rightsquigarrow \mathsf{T}_1 \quad e_2 : \Sigma^* \rightsquigarrow \mathsf{T}_2 \quad \cdots \quad e_k : \Sigma^* \rightsquigarrow \mathsf{T}_k \quad \mathsf{op} : \mathsf{T}_1 \times \mathsf{T}_2 \times \cdots \times \mathsf{T}_k \to \mathsf{T}}{\mathsf{op}(e_1, e_2, \ldots, e_k) : \Sigma^* \rightsquigarrow \mathsf{T}}\ (\textsc{t-combine})$$

$$\frac{e : \Sigma^* \rightsquigarrow \mathsf{T}_e \quad q : \mathsf{T}_f \quad f : \Sigma^* \rightsquigarrow \mathsf{T}_f}{e\{p \coloneqq f\} : \Sigma^* \rightsquigarrow \mathsf{T}_e}\ (\textsc{t-subst})$$

Figure 2.11: Typing rules for QREs.

Table 2.2: Defining $\mathrm{Param}(e)$ and single-use expressions.

e	$\mathrm{Param}(e)$	... is single-use if:
$\varphi \mapsto \lambda$	$\varnothing$	always.
$\epsilon \mapsto t$	$\mathrm{Param}(t)$	t is single-use.
bot	$\varnothing$	always.
e else f	$\mathrm{Param}(e) \cup \mathrm{Param}(f)$	e and f are both single-use.
$\mathrm{split}(e \to^p f)$	$\mathrm{Param}(e) \cup (\mathrm{Param}(f) \backslash \{p\})$	e and f are both single-use, and $\mathrm{Param}(e) \cap (\mathrm{Param}(f) \backslash \{p\}) = \varnothing$
$\mathrm{split}(e \leftarrow^q f)$	$(\mathrm{Param}(e) \backslash \{q\}) \cup \mathrm{Param}(f)$	e and f are both single-use, and $(\mathrm{Param}(e) \backslash \{q\}) \cap \mathrm{Param}(f) = \varnothing$
$\mathrm{iter}(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k)$ $\mathrm{iter}(p_1 \leftarrow e_1, p_2 \leftarrow e_2, \ldots, p_k \leftarrow e_k)$	$\{p_1, p_2, \ldots, p_k\}$	$\forall i$, e_i is single-use, $\mathrm{Param}(e_i) \subseteq \{p_1, p_2, \ldots, p_k\}$, and $\mathrm{Param}(e_i)$ is pairwise disjoint.
$\mathrm{op}(e_1, e_2, \ldots, e_k)$	$\bigcup_i \mathrm{Param}(e_i)$	$\forall i$, e_i is single-use, and $\mathrm{Param}(e_i)$ is pairwise-disjoint
$e\{p := f\}$	$(\mathrm{Param}(e) \backslash \{p\}) \cup \mathrm{Param}(f)$	e and f are both single-use, and $(\mathrm{Param}(e) \backslash \{p\}) \cap \mathrm{Param}(f) = \varnothing$

Even consistent, well-typed expressions, left otherwise unrestricted, may map input streams of length n to output terms of size $O(2^n)$. This is the same concern that in section 2.3 led us to define single-use terms. We now lift the idea to single-use *expressions*.

We associate each expression e with a parameter support $\mathrm{Param}(e)$. The idea is that $\mathrm{Param}(\llbracket e \rrbracket(w)) \subseteq \mathrm{Param}(e)$, and it therefore over-approximates the set of parameters on which $\llbracket e \rrbracket(w)$ may *ever* depend. Using $\mathrm{Param}(e)$, we then define the set of single-use expressions so that sub-expressions always operate on distinct parameter spaces. We formally define $\mathrm{Param}(e)$ and the single-use restrictions in table 2.2. The following theorem is a straightforward consequence:

Theorem 2.23. *1. If $(w, t) \in \llbracket e \rrbracket$, then $\mathrm{Param}(t) \subseteq \mathrm{Param}(e)$.*

2. If e is single-use and $(w, t) \in \llbracket e \rrbracket$, then t is single-use and $|t| = O(|e|, |w|)$.

QREs as a generalization of DReX. We introduced QREs as a generalization of DReX: this connection between the two is formalized by the following theorem.

Theorem 2.24. *1. Let $e_d : \Sigma^* \rightsquigarrow \mathbb{D}$ be a consistent DReX expression, where $(\mathbb{D}, \cdot, 1_{\mathbb{D}})$ is the output monoid. Let the cost domain appear in the space of types, $\mathbb{D} \in \mathcal{T}$, and the space*

of operations Ops include the monoid operation $\cdot$. Then there is a consistent, well-typed, single-use QRE e_q which is equivalent to e_d.

2. *Let $e_q : \Sigma^* \rightsquigarrow \mathbb{D}$ be a consistent, well-typed, single-use QRE, where $(\mathbb{D}, \cdot, 1_{\mathbb{D}})$ is a monoid. If all sub-expressions e' of e output values of type $\mathbb{D}$, $e' : \Sigma^* \rightsquigarrow \mathbb{D}$, and all terms t appearing in e are of type $\mathbb{D}$ with $\cdot$ being the only operator used for combination, then there is a consistent DReX expression e_d which is equivalent to e_q.*

The proof will follow from our results in part I: from every DReX expression, we can construct an equivalent SST, SST and SSTTs are equi-expressive when the cost domain is a monoid (theorem 3.9), and from the SSTT, we can extract an equivalent QRE. The process can be run in reverse to extract a DReX expression from the QRE.

2.5 Examples

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

We will first write the expression t_{sc} which calculates the time needed to travel from the suburbs to the city. We define the predicates $\varphi_{ss}(d) = (d.\texttt{line} = \texttt{sl} \wedge d.\texttt{station} = \texttt{suburb})$ and $\varphi_{sc}(d) = (d.\texttt{line} = \texttt{sl} \wedge d.\texttt{station} = \texttt{city})$ which indicate that the suburban line train is at the suburban station and at the city station respectively. The first two expressions,

$$t_{\text{sc-pickup}} = \text{split}((\neg\varphi_{ss})^* \mapsto 0, \varphi_{ss} \mapsto \texttt{time}, (\neg\varphi_{sc})^* \cdot \varphi_{sc} \mapsto 0, +), \text{ and}$$
$$t_{\text{sc-drop}} = \text{split}((\neg\varphi_{ss})^* \cdot \varphi_{ss} \cdot (\neg\varphi_{sc})^* \mapsto 0, \varphi_{sc} \mapsto \texttt{time}, +)$$

are both defined for streams of the form $(\neg\varphi_{ss})^* \cdot \varphi_{ss} \cdot (\neg\varphi_{sc})^* \cdot \varphi_{sc}$. The expressions return the time at which the traveler was picked up from the suburbs and dropped off at the city centre respectively. Our goal was to express the commute time from the suburbs to the city:

$$t_{sc} = t_{\text{sc-drop}} - t_{\text{sc-pickup}}.$$

We can similarly express the query t_{ca} for the commute time from the city to the airport. The total travel time from the suburbs to the airport is given by

$$t_{sa} = \text{split}(t_{sc}, t_{ca}, +).$$

Finally, the expression:

$$t_{avg} = \text{iter}(t_{sa}, \text{avg})$$

computes the average travel time from the suburbs to the airport.

2.5.2 Rolling data telephone plans

Now consider a simple telephone plan, where the customer has a 5 GB monthly download limit. She can potentially use more than this amount, for an additional fee at the end of the month. Otherwise, the unused quota is added to her next month's limit, up to a maximum of 20 GB.

The data domain $\mathbb{D}_{\text{tel}} = \mathbb{R} \cup \{\mathrm{M}\}$, where $\mathrm{d} \in \mathbb{R}$ indicates a download of d gigabytes and M indicates the end of the billing cycle and payment of the telephone bill. Given the entire browsing history of the customer, $w \in [\![(\mathrm{R}^* \cdot \mathrm{M})^*]\!]$, we wish to compute the download limit for the current month.

The first expression, $\text{totalDown} = \text{iter}(\mathrm{d} \in \mathbb{R} \mapsto \mathrm{d}, +)$ maps a sequence of downloads $w \in \mathbb{R}^*$ to the total data downloaded. The browsing history of a single month is given by the regular expression $\mathrm{r_m} = \mathbb{R}^* \cdot \mathrm{M}$, and the expression

$$\text{monthDown} = \text{split}(\text{totalDown}, \mathrm{M} \mapsto 0, +)$$

maps data streams $w \in [\![\mathrm{r_m}]\!]$ to the total quantity of data downloaded.

The following expression gives the unused allowance available at the end of the month, in terms of l_0, the download limit at the beginning of the month:

$$\text{unused}(\mathrm{l}_0) = (\mathrm{r_m} \mapsto \mathrm{l}_0) - \text{monthDown}\,.$$

The value returned by unused may be negative, or larger than 20 GB, so the expression:

$$\text{rollover}(\mathrm{l}_0) = \min(\max(\text{unused}(\mathrm{l}_0), \mathrm{r_m} \mapsto 0), \mathrm{r_m} \mapsto 20)$$

caps it to between 0 GB and 20 GB, and gives the download quota to be added to the next month's limit.

The expression $\text{nextQuota}(\mathrm{l}_0) = \text{rollover}(\mathrm{l}_0) + 5$ provides the download limit for the next month in terms of l_0 and the current month's browsing history. Finally, the expression

$$\text{quota} = (\text{iter}(\text{nextQuota} \to \mathrm{l}_0))[\mathrm{l}_0 := (\mathrm{r_m^*} \mapsto 5)]$$

maps the entire browsing history of the customer to the download limit of the current month.

2.5.3 Aggregating weather reports

Our final example deals with a stream of weather reports. Let the data domain $\mathbb{D}_{\text{wth}} = \mathbb{R} \cup \{\texttt{autEqx}, \texttt{sprEqx}, \texttt{newYear}, \ldots\}$, where the symbols `autEqx` and `sprEqx` represent the autumn and spring equinoxes. A measurement $\mathrm{d} \in \mathbb{R}$ indicates a temperature of d°C. We wish to compute the average winter temperature. For this query, let winter be defined as the time between an autumn equinox and the subsequent spring equinox.

Observe that the regular expressions $\mathrm{r}_{\text{summ}} = \mathbb{R}^* \cdot \texttt{autEqx}$ and $\mathrm{r}_{\text{winter}} = \mathbb{R}^* \cdot \texttt{sprEqx}$ respectively capture a sequence of temperature readings made before the start of winter and

the start of summer. Given a sequence of temperature readings for a year, $w \in [\![\mathbb{R}^* \cdot \texttt{autEqx} \cdot \mathbb{R}^* \cdot \texttt{sprEqx}]\!]$, the expression

$$\begin{aligned} y &= \text{split}(s, w, \cup), \text{ where} \\ s &= r_{\text{summ}} \mapsto \varnothing, \text{ and} \\ w &= \text{split}(\text{iter}(d \in \mathbb{R} \mapsto \{d\}, \cup), \texttt{sprEqx} \mapsto \varnothing, \cup) \end{aligned}$$

returns the set of temperature measurements made in winter. The expression

$$\text{avgWinter} = \text{avg}(\text{iter}(y, \cup))$$

encodes the ultimate query of interest: the average winter-time temperature reading.

2.6 Streaming Composition

Recall example 2.14: the data stream was a sequence of customer transactions with a bank. The original query bal mapped input streams w of the form $(\mathbb{R} \cup \{M\})^* \cdot M$ to the closing balance. We now wish to express the query hiBal, which returns the maximum account balance ever achieved at the end of a month.

Given a sequence of numbers $w' \in \mathbb{R}^*$, the query $\text{hi} = \text{iter}(\text{true} \mapsto x, \max)$ returns the largest element of the stream. It is natural to compute hiBal by first evaluating the query bal on the original input stream w, and feeding the sequence of results produced by this evaluator as input to the subsequent query hi. See figure 2.12. To express this idiom, we introduce the "streaming composition" operator $\gg$. With this operator, hiBal can be defined as:

$$\text{hiBal} = \text{bal} \gg \text{hi}\,.$$

Streaming composition has been very useful in our later work [20]. We will now formally define the operator and its semantics.

$e \gg f$. Let e be a consistent QRE with $\text{Param}(e) = \varnothing$, so that it maps input streams $w \in \Sigma^*$ to concrete values $v \in T_e$. Let f be a consistent expression which maps streams $w' \in T_e^*$ to terms $t \in \text{Terms}(T_f)$. Then the expression $e \gg f$ maps input streams $w \in \Sigma^*$ to output terms $t \in \text{Terms}(T_f)$, and is defined as follows. If $w = w_1 w_2 \cdots w_n$, then for each $i \in \{0, 1, 2, \ldots, n\}$, let:

$$v_i = \begin{cases} [\![e]\!](w_1 w_2 \cdots w_i) & \text{if } [\![e]\!](w_1 w_2 \cdots w_i) \text{ is defined, and} \\ \epsilon & \text{otherwise,} \end{cases}$$

$$\text{so that } [\![e \gg f]\!](w) = \begin{cases} [\![f]\!](v_0 v_1 \cdots v_n) & \text{if } [\![e]\!](w) \text{ is defined, and} \\ \perp & \text{otherwise.} \end{cases} \tag{2.6.1}$$

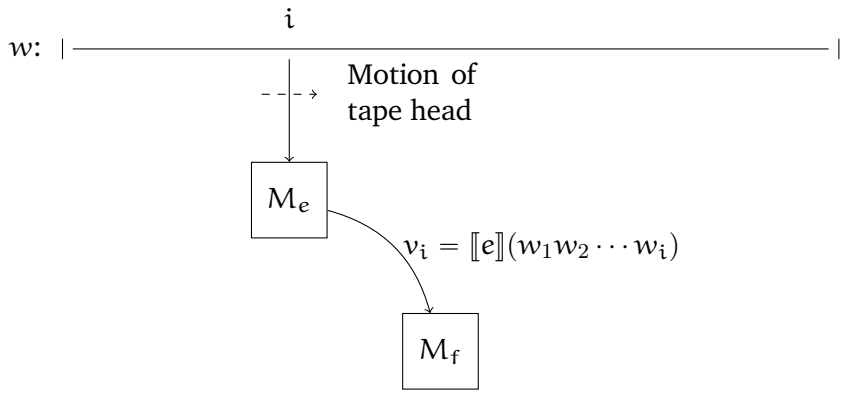


Figure 2.12: Streaming composition $e \gg f$ is best visualized operationally, as shown here. The evaluator M_e for e reads the original input stream w in one left-to-right pass. The sequence of results produced by M_e, on progressively longer prefixes of w, is fed into M_f.

Param$(e \gg f)$**.** The parameter support is defined as $\text{Param}(e \gg f) = \text{Param}(f)$. It does not make sense for elements v_i of the intermediate stream $v_0 v_1 \cdots v_n$ to be incomplete terms. We have therefored mandated in the definition of $e \gg f$ that $\text{Param}(e) = \varnothing$, so that $v_0 v_1 \cdots v_n$ is a sequence of concrete values.

QRE$_\gg$ and expression consistency. We write $\text{QRE}_\gg$ for the space of QREs extended with the streaming composition operator. For the remaining operators, we naturally lift the notion of consistency from plain QREs (table 2.1) to the extended class. Observe that we only define $e \gg f$ in the case when both sub-expressions e and f are consistent. We declare the expression $e \gg f$ to be consistent whenever $\text{Dom}(e \gg f)$ is a regular language.

Remark 2.25. While it is always the case that $\text{Dom}(e \gg f) \subseteq \text{Dom}(e)$, $\text{Dom}(e \gg f)$ need not always be regular. We will not present an algorithm to mechanically determine the consistency of expressions in the extended class, $\text{QRE}_\gg$. Depending on the choice of cost domain and its operators, the problem of mechanical consistency checking may even be undecidable.

Example 2.26. Streaming composition is useful in constructing multi-stage pipelines. Consider an online retailer who manages a set of customer ids, CustId. The event stream is a sequence of requests from customers, $w \in \text{CustId}^*$. Each customer $c \in \text{CustId}$ has a home city $\text{city}(c)$: this mapping is maintained in a separate table. To organize its inventory, the company would like to count the number of requests from Philadelphia.

This query can be naturally expressed as a two-step lookup-compute pipeline as follows. The first expression

$$e_1 = \text{split}(\Sigma^* \mapsto 0, \text{city}, \pi_2)$$

maps streams of the form CustId^+ to the city of the last customer. Here, $\pi_2(x, y) = y$ is the

standard second-projection function. The second expression:

$$e_2 = \text{iter}(\text{x} = \text{Philadelphia} \mapsto 1 \text{ else } \text{x} \neq \text{Philadelphia} \mapsto 0, +)$$

maps a sequence of city names to the number of occurrences of Philadelphia. The two expressions can then be glued together to write the final query: count $= e_1 \gg e_2$. $\triangle$

Part I

Expressiveness

Chapter 3

Regular Cost Functions

In this chapter, we will define regular cost functions and regular string transformations. Regular string transformations were first characterized as the class of functions which can be implemented by two-way deterministic finite state transducers (2-FSTs). Following Courcelle's suggestion of describing graph transformations as formulas in monadic second-order logic [38], Engelfriet and Hoogeboom showed that the string transformations thus expressible coincide with those implementable using 2-FSTs [51], and introduced the adjective "regular" to describe this class. Finally, Alur and Černý [9] introduced the deterministic one-pass model of streaming string transducers (SSTs), and showed that SSTs implement the class of regular string transformations.

In a recent paper [13], we generalized the idea of regular string transformations to the class of regular cost functions. Regular cost functions share many of the same appealing properties as regular string transformations, including closure under regular look-ahead and input reversal, and can be equivalently represented either logically as string-to-term transformations definable in MSO, or operationally as functions implemented by streaming string-to-term transducers (SSTT).

In this chapter, we will present the operational definitions of these classes as the functions implementable using SSTs and SSTTs. Our principal expressiveness results are that DReX expresses the same class of functions as SSTs, and that QREs express the same class of functions as SSTTs.

3.1 Streaming String Transducers

An SST is a one-way machine which can compute functions such as:

$$f(w) = \begin{cases} w & \text{if } w \text{ ends with an } a, \text{ and} \\ \text{reverse}(w) & \text{otherwise.} \end{cases} \tag{3.1.1}$$

To be able to compute f in one left-to-right pass, the machine maintains multiple registers, whose values are updated on each transition. In figure 3.1, we draw the SST M_1 which computes f. Observe that its operation is guided by a finite space of control states, $\{q_0, q_1\}$,

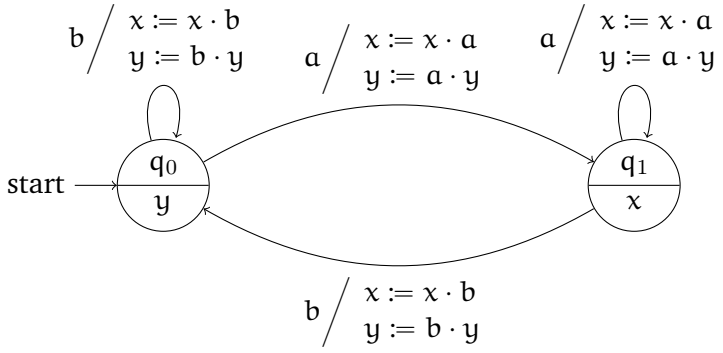


Figure 3.1: Example SST M_1 which computes f from equation 3.1.1. The machine ends in state q_1 for all strings ending with an a, and ends in state q_0 otherwise. The register x always contains the entire string w read so far, and y always contains reverse(w). The machine outputs the value of register y in state q_0 and outputs the value of x in state q_1.

and that the registers are themselves test-free. Now consider the SST M_2 from figure 3.2. Observe that registers can also be combined during transition updates, such as the assignment $x := x \cdot y$. These updates happen in parallel. To prevent non-linear output growth, we require the updates to be syntactically "copyless", i.e. that on each transition, each register may appear at most once on the right-hand side of an assignment statement. If x and y are registers and a and b are constants, then the following assignment is copyless:

$$x := x \cdot y$$
$$y := aba$$

but the following assignment is not:

$$x := x \cdot y$$
$$y := y \cdot a$$

because the register y appears on the right-hand side of the new values of both x and y. Copylessness is similar in intent and operation to the single-use restriction that concerned us in chapter 2.

Definition 3.1 (Copylessness)**.** If $V = \{v_1, v_2, \ldots, v_k\}$ is a finite set of registers and $\mathbb{D}$ is a cost domain, then a string $w \in (V \cup \mathbb{D})^*$ is COPYLESS if each register $v_i \in V$ occurs at most once in w. A function $f : V \to (V \cup \mathbb{D})^*$ is copyless if the concatenated output string "$f(v_1)f(v_2)\cdots f(v_k)$" is copyless.

Definition 3.2 (SST)**.** An SST is a tuple $M = (Q, V, \Sigma, \mathbb{D}, \delta, \mu, q_0, F, \nu)$, where:

1. Q is a finite set of states,
2. V is a finite set of registers,

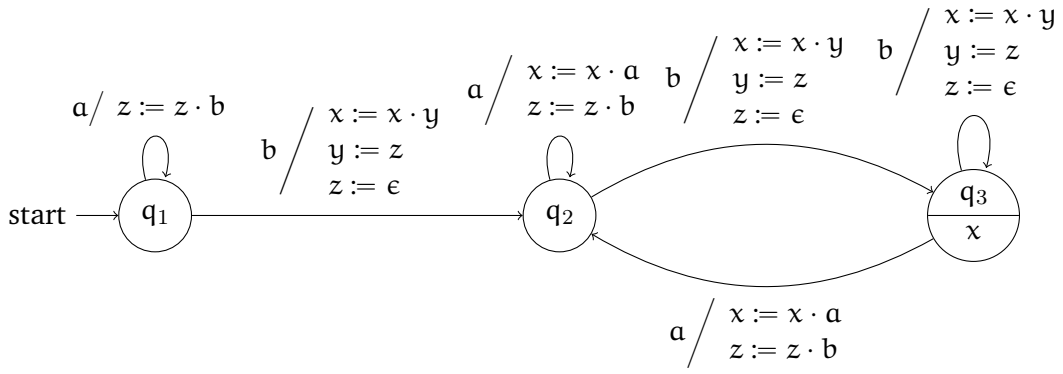


Figure 3.2: SST M_2 which computes the function shuffle from figure 2.3.The machine has a finite set of control states $Q = \{q_1, q_2, q_3\}$, and a finite set of string-valued registers, $V = \{x, y, z\}$. There are two principal restrictions: (*a*) registers are *test-free*, i.e. transitions cannot depend on the contents of the registers, and (*b*) register updates are copyless.

3. Σ is a finite input alphabet,
4. $\mathbb{D}$ is the output monoid (with operation $\cdot$ and identity element $1_{\mathbb{D}}$),
5. $\delta : Q \times \Sigma \to Q$ is the state transition function,
6. $\mu : Q \times \Sigma \times V \to (V \cup \mathbb{D})^*$ is the register update function such that for each $q \in Q$ and $a \in \Sigma$, the partial application $\mu(q, a) : V \to (V \cup \mathbb{D})^*$ is copyless,
7. $q_0 \in Q$ is the initial state,
8. $F \subseteq Q$ is the set of accepting states, and
9. $\nu : F \to (V \cup \mathbb{D})^*$ is the output function, such that for each $q \in F$, $\nu(q)$ is copyless.

Semantics. We define the semantics of an SST M using CONFIGURATIONS. A configuration indicates the current state and register valuation: formally, it is a pair $\gamma = (q, \mathrm{val})$, where $q \in Q$ and $\mathrm{val} : V \to \mathbb{D}$. The initial configuration $\gamma_0 = (q_0, \mathrm{val}_0)$, where $\mathrm{val}_0(v) = 1_{\mathbb{D}}$, for each register $v \in V$.

Next, the register valuation function $\mathrm{val} : V \to \mathbb{D}$ can be naturally lifted to strings $\hat{\mathrm{val}} = (V \cup \mathbb{D})^* \to \mathbb{D}$ as follows:

$$\hat{\mathrm{val}}(w) = \begin{cases} 1_{\mathbb{D}} & \text{if } w = \epsilon, \\ \hat{\mathrm{val}}(w') \cdot \mathrm{val}(v) & \text{if } w = w'v, \text{ for some register } v, \text{ and} \\ \hat{\mathrm{val}}(w') \cdot d & \text{if } w = w'd, \text{ for some constant } d \in \mathbb{D}. \end{cases}$$

For example, if $\mathrm{val} = \{x \mapsto ab, y \mapsto ba\}$, then $\hat{\mathrm{val}}(yabx) = baabab$.

Given the present configuration $\gamma = (q, \mathrm{val})$ of the machine, and the next input symbol a, we define the subsequent configuration $\gamma' = (q', \mathrm{val}')$ as follows: (*a*) $q' = \delta(q, a)$, and (*b*) for each $v \in V$, $\mathrm{val}'(v) = \hat{\mathrm{val}}(\mu(q, a, v))$. We succinctly represent this relation as $\gamma \to^a \gamma'$, and naturally lift this to entire input strings $w \in \Sigma^*$: $\gamma \to^w \gamma'$.

Finally, M defines a partial function $[\![M]\!] : \Sigma^* \to \mathbb{D}$ defined as follows. Given an input string w, say $\gamma \to^w \gamma_f$, where $\gamma_f = (q_f, \mathrm{val}_f)$. If $q_f \in F$, then:

$$[\![M]\!](w) = \hat{\mathrm{val}}_f(\nu(q_f)),$$

and otherwise, if $q_f \notin F$, then $[\![M]\!](w)$ is undefined. A string transformation $f : \Sigma^* \to \Gamma^*$ is REGULAR if there is an SST M such that for all w, $f(w) = [\![M]\!](w)$.

Function composition. The most important closure property of SSTs is under function composition. Other properties, such as closure under input reversal or regular look-ahead are corollaries of the following theorem.

Theorem 3.3 ([10])**.** *Let Σ and Γ be finite alphabets, and $\mathbb{D}$ be a monoid. If M_1 is an SST which computes a partial function $[\![M_1]\!] : \Sigma^* \to \Gamma^*$, and M_2 is an SST which computes a partial function $[\![M_2]\!] : \Gamma^* \to \mathbb{D}$, then there is an effectively constructible SST M which computes $[\![M]\!] = [\![M_2]\!] \circ [\![M_1]\!]$.*

Non-determinism. We can extend SSTs with non-determinism in the obvious way: the state transition function $\delta : Q \times \Sigma \to Q$ is replaced with a state transition relation $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$, and the register update function $\mu : Q \times \Sigma \times V \to (V \cup \mathbb{D})^*$ is replaced with $\mu : \Delta \times V \to (V \cup \mathbb{D})^*$. As expected, each non-deterministic SST M defines a relation $[\![M]\!] \subseteq \Sigma^* \times \mathbb{D}$.

A non-deterministic machine is UNAMBIGUOUS if for each input string $w = w_1 w_2 \cdots w_n$, there is at most one path $\pi = q_0 \to^{w_1} q_1 \to^{w_2} q_2 \to^{w_3} \cdots \to^{w_n} q_n$ from the initial state q_0 such that q_n is an accepting state. If M is unambiguous, then $[\![M]\!] : \Sigma^* \to \mathbb{D}$ is once again a *partial function* (each input mapped to at most one output). Unambiguous SSTs are no more expressive than plain deterministic SSTs:

Theorem 3.4 ([15])**.** *If M is an unambiguous SST which maps input strings $w \in \Sigma^*$ to outputs $d \in \mathbb{D}$, then there is an effectively constructible (*deterministic*) SST M_d such that $[\![M]\!] = [\![M_d]\!]$.*

Proof sketch. We can use the fact that SSTs are closed under composition to provide a simple proof of the present claim. First, every unambiguous SST can be turned into an ϵ-transition-free unambiguous SST by simply taking the closure of all ϵ-transitions. Then, we express M as the composition of two SSTs M_{d1} and M_{d2}:

1. M_{d1} maps each input string w to the accepting path π through the unambiguous machine M, if it exists. Formally, if $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation of M, then the partial function $[\![M_{d1}]\!] : \Sigma^* \to \Delta^*$ maps input strings w to the (necessarily unique) sequence of transitions through M:

$$d_1 = (q_0, w_1, q_1), d_2 = (q_1, w_2, q_2), d_3 = (q_2, w_3, q_3), \ldots, d_n = (q_{n-1}, w_n, q_n) \in \Delta^*,$$

where q_0 is the initial state, q_n is an accepting state of M. Let the entire input string be the sequence of symbols $w = w_1 w_2 \cdots w_n$. In place of the original symbol w_i, M_{d1} instead outputs the specific transition that must now be taken so that the machine will eventually reach an accepting state. This decision is clearly dependent on the (still unseen) suffix $w_{i+1} w_{i+2} \cdots w_n$. It is therefore natural to construct M_{d1} itself as the composition of several simpler SSTs:

(a) M_{d11} reverses the input string.

(b) M_{d12} maps a sequence of characters $w_n w_{n-1} \cdots w_1$ to the sequence of functions $f_n f_{n-1} \cdots f_1$ defined as follows. Each function $f_i : \Delta \to \text{Bool}$ maps transitions $d = (q, a, q') \in \Delta$ to the truth value of the following proposition: "$a = w_i$ and there exists a path $\pi = q' \to^{w_{i+1} w_{i+2} \cdots w_n} q_f$, such that q_f is an accepting state of M". Informally, $f_i(d)$ predicts the destiny of the execution of the original machine M in case transition d is taken after reading the symbol w_i.

Observe that the space of these "destiny functions" $f : \Delta \to \text{Bool}$ is finite. The state space of M_{d12} is the set of all destiny functions. After reading the symbol a, M_{d12} transitions from the state f to the state f' defined as follows:

$$f'((q, b, q')) = (a = b) \wedge \bigvee \{f((q', c, q_f)) \mid (q', c, q_f) \in \Delta\}$$

(c) M_{d13} reverses the input string (a sequence of destiny functions).

(d) The final machine M_{d14} parses this knowledge of the future to the specific action that M must take to reach an accepting state. Formally, given a sequence of destiny functions, $f_1 f_2 \cdots f_n$, M_{d14} produces a sequence of transitions $d_1 d_2 \cdots d_n \in \Delta^*$, such that:

i. $d_1 = (q_0, a, q_1)$, where q_0 is the initial state, and

ii. for all i, (*a*) $f_i(d_i) = \text{true}$, and (*b*) d_i and d_{i+1} are adjacent transitions: if $d_i = (q, a, q')$ and $d_{i+1} = (q'', b, q''')$, then $q' = q''$.

2. M_{d2} takes an accepting path π through the original machine M as input, and produces the final value as output.

Observe that $[\![M]\!](w) = [\![M_{d2}]\!] \circ [\![M_{d14}]\!] \circ [\![M_{d13}]\!] \circ [\![M_{d12}]\!] \circ [\![M_{d11}]\!](w)$, for all input strings w. The result follows by direct application of theorem 3.3. □

Remark 3.5. We will only use unambiguous SSTs for the proof of theorem 4.1. In all other cases, when we use the unqualified term "SST", we refer to the deterministic class of machines.

3.2 Streaming String-to-Term Transducers

QREs generalized DReX by allowing expressions to map input strings to output terms. Similarly, SSTTs generalize SSTs by producing and maintaining intermediate terms, rather than concrete values. Most simply, an SSTT is obtained by permitting the registers of the machine to carry terms (as syntactic objects), instead of being restricted to carrying concrete values.

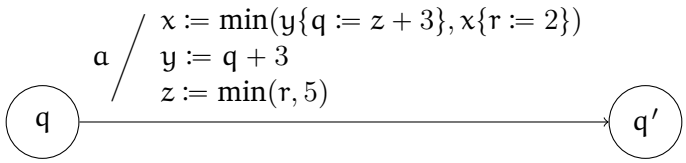


Figure 3.3: Example transition in an SSTT. There are three registers, $\{x, y, z\}$, all integer-valued. The parameters of interest are p, q, and r, all also integer-valued. We declare that $P_x = \{p, r\}$, $P_y = \{q\}$, and $P_z = \{r\}$, indicating what parameters the runtime values of these registers may depend on. There are two parts to ensuring copylessness: first, the registers must appear at most once on the right-hand side of the update expressions, and second, each register must always hold a single-use term.

There are two issues in this generalization that require a little care: (*a*) ensuring that each register always holds a term of the correct type, and (*b*) ensuring copylessness in the presence of parameter substitutions. Each register ν holds a term t: the idea is to associate ν with a set of parameters P_ν which over-approximates $\mathrm{Param}(t)$, and ensure that the register updates are consistent with P_ν.

Recall from section 2.3 that $\mathcal{P}$ is the set of all parameters in the universe. For simplicity, we assume that the set of registers V is itself a finite subset of $\mathcal{P}$, $V \subseteq \mathcal{P}$. Each register ν is associated with a type T, expressed succinctly as $\nu : T$, and each register be associated with a set $P_\nu \subseteq \mathcal{P}$ of parameters such that $P_\nu \cap V = \varnothing$.

Example 3.6. We present an example transition of an SSTT in figure 3.3. There are three integer-valued registers, x, y, and z, and three parameters, p, q, and r, all also integer-valued. We declare that $P_x = \{p, r\}$, $P_y = \{q\}$, and $P_z = \{r\}$, i.e. that at runtime, if the register valuations are $\{x \mapsto t_x, y \mapsto t_y, z \mapsto t_z\}$, then

$$\begin{aligned}&\mathrm{Param}(t_x) \subseteq \{p, r\},\\ &\mathrm{Param}(t_y) \subseteq \{q\}, \text{ and}\\ &\mathrm{Param}(t_z) \subseteq \{r\}.\end{aligned}$$

Thus, the following is a well-formed register assignment:

$$\mathrm{val}_1 = \{x \mapsto \min(p, r + 5), y \mapsto 4, z \mapsto r + 2\},$$

but the following is not well-formed:

$$\mathrm{val}_2 = \{x \mapsto \min(p, r + 5), y \mapsto r + 4, z \mapsto r + 2\},$$

because here y depends on the forbidden parameter r.

Now focus on the register updates during the $q \to q'$ transition. As with QREs, the update expression $\dot{t_1}\{p := \dot{t_2}\}$ indicates that the actual term substitution needs to be performed while

executing the machine. Observe that the expresions guarantee that if the original register values depend only on permitted parameters ($\mathrm{Param}(\mathrm{val}(\nu)) \subseteq P_\nu$, for all registers ν), then their final values also depend only on permitted parameters.

Lastly, let the initial register valuation be $\{x \mapsto \min(p, r), y \mapsto q + 2, z \mapsto r\}$. Then the term held by x after the transition is

$$\begin{aligned} t'_x &= \min((q+2)[q := r+3], \min(p, r)[r := 2]) \\ &= \min((r+3)+2, \min(p, 2)) \\ &= \min(r+5, p, 2), \end{aligned}$$

which is also a single-use term. On the other hand, consider the following alternative update expression for the register x:

$$x := \min(y\{q := z+3\}, x).$$

If we consider the same initial valuation of the registers as before, then the term held by x after the transition is:

$$\begin{aligned} t''_x &= \min(q+2[q := r+3], \min(p, r)) \\ &= \min((r+3)+2, \min(p, r)) \\ &= \min(r+5, p, r), \end{aligned}$$

which is not single-use. Observe that the original update expression of figure 3.3 statically guarantees that if the original register valuation is single-use, then so is the final register valuation.

We will set up the definitions of this section so that SSTTs automatically guarantee these well-typedness and single-use properties. △

Register valuations, update terms, and register assignments. A REGISTER VALUATION is a function $\mathrm{val} : V \to \text{Terms}$. It is WELL-FORMED if for each ν, (*a*) $\mathrm{val}(\nu) : T_\nu$, (*b*) $\mathrm{val}(\nu)$ is single-use, and (*c*) $\mathrm{Param}(\mathrm{val}(\nu)) \subseteq P_\nu$.

From constants c, parameters p, registers ν, and operations op, we can naturally construct UPDATE TERMS:

$$\begin{aligned} \dot{t} \quad ::= \quad & c \mid p \mid \mathrm{op}(\dot{t}_1, \dot{t}_2, \ldots, \dot{t}_k) \\ \mid \quad & \dot{t}_1\{p := \dot{t}_2\} \end{aligned} \tag{3.2.1}$$

Compare this definition with the definition of terms in equation 2.3.1: the only difference is the addition of the operator for substitution $\dot{t}_1\{p := \dot{t}_2\}$. As with QREs, this operation is distinct from term substitution $t_1[p := t_2]$, and indicates that substitution is to be performed at runtime.

Let $\mathrm{UpdTerms}(T)$ be the set of well-typed update terms of type T. The parameter support $\mathrm{Param}(\dot{t})$ of an update term $\dot{t}$ is inductively defined as follows: (*a*) $\mathrm{Param}(c) = \varnothing$, (*b*) $\mathrm{Param}(p) = \{p\}$ if $p \notin V$ and $\mathrm{Param}(\nu) = P_\nu$ for all $\nu \in V$, (*c*) $\mathrm{Param}(\mathrm{op}(\dot{t}_1, \dot{t}_2, \ldots, \dot{t}_k)) =$

$\mathrm{Param}(\dot{t}_1) \cup \mathrm{Param}(\dot{t}_2) \cup \cdots \cup \mathrm{Param}(\dot{t}_k)$, and (*d*) $\mathrm{Param}(\dot{t}_1[p := \dot{t}_2]) = (\mathrm{Param}(\dot{t}_1) \backslash \{p\}) \cup \mathrm{Param}(\dot{t}_2)$. The register support $\mathrm{Regs}(\dot{t})$ is the set of registers appearing in $\dot{t}$.

COPYLESS update terms $\dot{t}$ are inductively defined as follows: (*a*) the leaf terms c and p are always copyless, (*b*) $\mathrm{op}(\dot{t}_1, \dot{t}_2, \ldots, \dot{t}_k)$ is copyless if each sub-term $\dot{t}_i$ is copyless and $\mathrm{Param}(\dot{t}_i)$ is pairwise-disjoint, and (*c*) $\dot{t}_1[p := \dot{t}_2]$ is copyless if $(\mathrm{Param}(\dot{t}_1) \backslash \{p\}) \cap \mathrm{Param}(\dot{t}_2) = \varnothing$.

A REGISTER ASSIGNMENT is a function $f : V \to \mathrm{UpdTerms}$. It is WELL-FORMED if $\mathrm{Regs}(f(v))$ is pair-wise disjoint and for each register v, (*a*) $f(v) : T_v$, (*b*) $f(v)$ is copyless, and (*c*) $\mathrm{Param}(f(v)) \subseteq P_v$.

Given the set of register $V = \{v_1, v_2, \ldots, v_k\}$, a register valuation $\mathrm{val} : V \to \mathrm{Terms}$ and a register assignment $f : V \to \mathrm{UpdTerms}$, we define the subsequent register valuation $\mathrm{val}' = f(\mathrm{val})$ as follows:

$$\mathrm{val}'(v) = f(v)[v_1 := \mathrm{val}\, v_1, v_2 := \mathrm{val}(v_2), \ldots, v_k := \mathrm{val}(v_k)].$$

The following result is a sanity check that the above definitions are meaningful:

Proposition 3.7. *If val is a well-formed register valuation, and $f : V \to$ UpdTerms is a well-formed register assignment, then the subsequent register valuation $\mathrm{val}' = f(\mathrm{val})$ is also a well-formed register valuation.*

Definition 3.8 (SSTT)**.** An SSTT is a tuple $M = (Q, V, \Sigma, T_{\mathrm{out}}, \delta, \mu, q_0, \mathrm{val}_0, F, \nu)$, where:

1. Q is a finite set of states,
2. V is a finite set of registers,
3. Σ is a finite input alphabet,
4. T_{out} is the type of the output term,
5. $\delta : Q \times \Sigma \to Q$ is the state transition function,
6. $\mu : Q \times \Sigma \times V \to \mathrm{UpdTerms}$ is the register update function such that for each q, a, the register assignment $\mu(q, a) : V \to \mathrm{UpdTerms}$ is well-formed,
7. $q_0 \in Q$ is the initial state,
8. val_0 is a well-formed initial register valuation,
9. $F \subseteq Q$ is the set of accepting states, and
10. $\nu : F \to \mathrm{Terms}(T_{\mathrm{out}})$ is the output function.

Semantics. We will define the semantics of SSTTs using configurations, just as we did with SSTs. A configuration of M is a pair $\gamma = (q, \text{val})$, where $q \in Q$ is the current state, and val is a well-formed register valuation. The initial configuration is $\gamma_0 = (q_0, \text{val}_0)$.

Given a configuration $\gamma = (q, \text{val})$ and an input symbol a, the update function $\mu(q, a) : V \to \text{UpdTerms}$ defines a register assignment. We then define the subsequent configuration $\gamma' = (q', \text{val}')$ as: (*a*) $q' = \delta(q, a)$, and (*b*) $\text{val}' = \mu(q, a)(\text{val})$. We use the notation $\gamma \to^a \gamma'$ to express the relation that γ' is the successor configuration of γ, and lift this to entire strings $\gamma \to^w \gamma'$ in the usual way.

Finally, we define the function $[\![M]\!] : \Sigma^* \to \text{Terms}(T_{\text{out}})$ implemented by the SSTT M as follows. Given an input string w, say that $\gamma_0 \to^w \gamma_f$, where $\gamma_f = (q_f, \text{val}_f)$. If $q_f \in F$, then:

$$[\![M]\!](w) = \nu(q)[\nu_1 := \text{val}_f(\nu_1)][\nu_2 := \text{val}_f(\nu_2)][\cdots][\nu_k := \text{val}_f(\nu_k)]. \tag{3.2.2}$$

Otherwise, $[\![M]\!](w)$ is undefined. A function $f : \Sigma^* \to \text{Terms}(T_{\text{out}})$ is a REGULAR COST FUNCTION if there is an SSTT M such that for all input streams w, $f(w) = [\![M]\!](w)$.

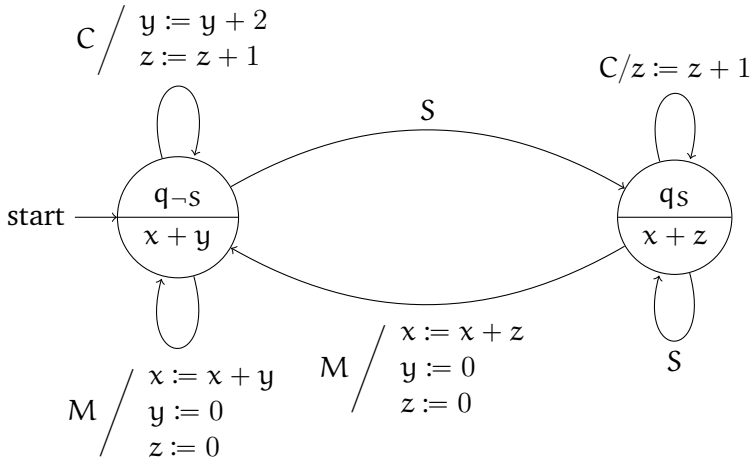
We present some example machines in figure 3.4. While parameters are required for full expressive power, in most cases of practical interest, parameter-free SSTTs suffice, as shown in the figure. Furthermore, the definition of SSTTs we give here, where each register holds an arbitrary term, is slightly more permissive than the original definition of [13], where each register contained at most one hole named "**?**". By theorem 3 of [11], it turns out that the two definitions are equivalent.

SSTTs generalize the class of functions expressible by SSTs. If the cost domain is restricted to a monoid $(\mathbb{D}, \cdot, 1_{\mathbb{D}})$, then the following theorem states that the classes of functions expressible SSTs and SSTTs coincide. This theorem originally appears as theorem 3 of [13], and states that "$\mathbb{F}^c(\mathbb{D}, \otimes) = \mathbb{R}(\mathbb{D}, \otimes)$". When rephrased in the terminology of this thesis, it states:

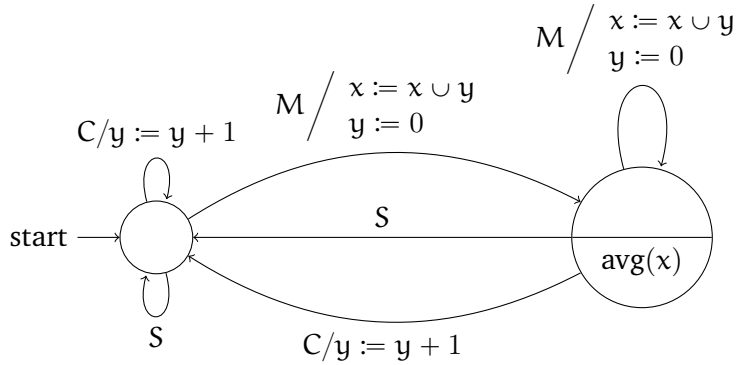
Theorem 3.9 (Theorem 3 of [13])**.** *Let $(\mathbb{D}, \cdot, 1_{\mathbb{D}})$ be a monoid, and Σ be a finite input alphabet. Let $\mathcal{F}$ be the class of functions $\Sigma^* \to \mathbb{D}$ expressible by SSTs. Let $\mathcal{R}$ be the class of functions expressible by SSTTs where all registers ν and all terms t appearing in the description are of type $\mathbb{D}$, and the only operation appearing in t is the monoid operation $\cdot$. Then $\mathcal{F}$ and $\mathcal{R}$ are equal.*

Non-determinism. Once again, we can replace the state transition function $\delta : Q \times \Sigma \to Q$ with a state transition relation $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ and the register update function $\mu : Q \times \Sigma \times V \to \text{UpdTerms}$ with $\mu : \Delta \times V \to \text{UpdTerms}$ to obtain non-deterministic SSTTs. A non-deterministic machine M defines a relation, $[\![M]\!] \subseteq \Sigma^* \times \text{Terms}(T_{\text{out}})$. The machine is unambiguous if for each string w, there is at most one accepting path $q_0 \to^w q_f$. For unambiguous machines, the denoted object is once again a partial function $[\![M]\!] : \Sigma^* \to \text{Terms}(T_{\text{out}})$. Along similar lines as theorem 3.4, we have the following result:

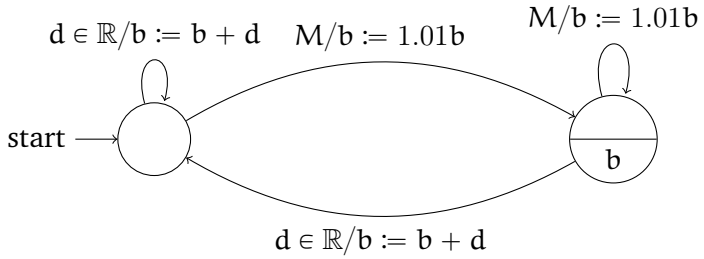
Theorem 3.10. *If M is an unambiguous SSTT which maps input strings $w \in \Sigma^*$ to outputs $t \in \text{Terms}(T_{out})$, then there is an effectively constructible (*deterministic*) SSTT M_d such that $[\![M]\!] = [\![M_d]\!]$.*



(a) M_3 computes the function coffee from example 2.7.



(b) M_4 computes the function $\#_C^{avg}$ from example 2.13.



(c) M_5 computes the function bal from example 2.14.

Figure 3.4: Example SSTTs.

Proof. We first construct an SST M_{d1} which maps an input string w to the (necessarily unique) accepting path through M. M_{d1} is identical to the construction presented in theorem 3.4. The second SSTT M_{d2} maps each each accepting π through the original machine M to the final value computed. By theorem 7 of [11], a deterministic machine M_d can be constructed which computes the same function as the composition $[\![M_{d2}]\!] \circ [\![M_{d1}]\!]$. □

Remark 3.11. As with SSTs, we will only use unambiguous SSTTs for the proof of theorem 4.1. The adjective "deterministic" will always be implicit in our usage of the unqualified term "SSTT".

Chapter 4

Converting Function Expressions into Transducers

The QRE / DReX can express exactly the class of regular functions. In this chapter, we will prove the first half of this claim, by presenting a translation algorithm from DReX expressions to SSTs and from QREs to SSTTs. The algorithms are mostly straightforward generalizations of the classical algorithm which converts regular expressions into equivalent DFAs, assuming the determinization procedure of theorems 3.4 and 3.10. Formally, the main result of this chapter is the following:

Theorem 4.1 (Closure)**.** *If Σ is a finite input alphabet, then:*

1. *If e is a consistent DReX expression, then we can construct an SST M which computes $[\![e]\!]$.*
2. *If e is a consistent QRE, then we can construct an SSTT M which computes $[\![e]\!]$.*

We made a stylistic decision in chapter 3 to present SSTs and SSTTs as operating over a finite input alphabet Σ. This is consistent with their original descriptions [10, 13], but narrows the immediate applicability of the results of this chapter to finite input alphabets. Theorem 4.1 will readily generalize if a determined reader rephrases the definitions of regular machines to operate on symbolic predicates.

4.1 From DReX to SSTs

We will now prove the first part of theorem 4.1, that every DReX expression can be translated into an equivalent SST. We will divide the proof into two parts: In the first part, lemma 4.2, we will show that every DReX expression that does not contain an occurrence of the chain or left-chain combinators can be translated into an equivalent SST. The construction will proceed by induction on the expression e, and is very similar to the familiar conversion from regular expressions to NFAs. While we could, in principle, carry out a similar construction even for the case of the chained sum, the details would be messy and unenlightening. Instead, in lemma 4.3, we construct a desugaring from the chained sum to function composition and

the remaining DReX operators, and we already know that SSTs are closed under function composition (theorem 3.3). The first part of theorem 4.1 will follow from lemmas 4.2 and 4.3.

Lemma 4.2. *If e is a consistent DReX expression without any occurrences of the chain or left-chain combinators, then we can effectively construct an SST M which computes e.*

Proof. The proof is by induction on the expression e. For each case, we will construct an unambiguous SST M_u, which can then immediately be determinized by applying theorem 3.4. For the induction hypotheses, we will assume *deterministic* SSTs for each sub-expression. We present several of the cases graphically in figure 4.1. For each machine constructed, we need to establish: (*a*) copylessness, (*b*) unambiguity, and (*c*) that it computes the relevant expression: well-parsed strings correspond to accepting paths and vice-versa.

Base cases: These cases are straightforward, as presented in figures 4.1a, 4.1b, and 4.1c. The machines are trivially copyless, unambiguous, and compute the appropriate base functions.

combine(e, f)**:** We perform the traditional product construction. Given SSTs M_e and M_f for e and f respectively, we construct a machine M with state space $Q = Q_e \times Q_f$ and register set $V = V_e \uplus V_f$. Here $A \uplus B = (\{0\} \times A) \cup (\{1\} \times B)$ is the disjoint union operator. The initial state of M is (q_{0e}, q_{0f}), and the set of accepting states $F = \{(q_{fe}, q_{ff}) \mid q_{fe} \in F_e, q_{ff} \in F_f\}$. The output function in state (q_{fe}, q_{ff}) is given by $\nu_e(q_{fe}) \cdot \nu_f(q_f)$, where the component output functions ν_e and ν_f are implicitly renamed to the new registers $\{0\} \times V_e$ and $\{1\} \times V_f$ respectively. For each symbol $a \in \Sigma$, and all synchronized transitions $q_e \to^a q'_e$ of M_e and $q_f \to^a q'_f$ of M_f, we create a transition $(q_e, q_f) \to^a (q'_e, q'_f)$ in M. Finally, the registers are updated separately: $\mu((q_e, q_f), a, v_e) = \mu_e(q_e, a, v_e)$ for each register v_e of M_e, and $\mu((q_e, q_f), a, v_f) = \mu_f(q_f, a, v_f)$. Once again, the registers in these equations are implicitly renamed to the appropriate registers of the product machine.

Because the registers are maintained separately, and have no interaction except in the output function ν, copylessness is trivial.

Consider a potentially ambiguous input string w, with two different accepting paths π_1 and π_2 through the product machine. Project each of these paths on to the component machines, to obtain paths π_{1e}, π_{1f}, π_{2e}, π_{2f}. Since $\pi_1 \neq \pi_2$, it follows that either $\pi_{1e} \neq \pi_{2e}$ or that $\pi_{1f} \neq \pi_{2f}$. Each of these possibilities leads to the conclusion that we have discovered an ambiguous string for one of the component machines, and this violates the induction hypothesis that the machines are deterministic. We therefore conclude that M is unambiguous.

Finally, by the definition of combine, every string w can be parsed by combine(e, f) iff it can also be parsed by both e and f. By the induction hypothesis, each of these cases coincide with the component machines returning correct results on w. By the construction, this coincides with the product machine M returning the expected result. Correctness follows.

e **else** f**:** See figure 4.1d. The register set is the disjoint union of those of the component machines: $V = V_e \uplus V_f$. Within each sub-structure, the registers are independently updated $\mu(q_e, a, \nu_e) = \mu_e(q_e, a, \nu_e)$ and $\mu(q_f, a, \nu_f) = \mu_f(q_f, a, \nu_f)$ for each state-register pair (q_e, ν_e) of M_e and (q_f, ν_f) of M_f. The "irrelevant" registers are simply reset on each transition: $\mu(q_e, a, \nu_f) = \mu(q_f, a, \nu_e) = \epsilon$, for each state-register pair (q_e, ν_e) of M_e and (q_f, ν_f) of M_f. The output functions are also directly lifted from those of the component machines: $\nu(q_e) = \nu_e(q_e)$ and $\nu(q_f) = \nu_f(q_f)$ for each state $q_e \in Q_e$ and $q_f \in Q_f$. It is straightforward to show copylessness, unambiguity, and correctness of the resulting machine M.

split(e, f) **and left-split**(e, f)**:** The machine maintains states $Q = Q_e \uplus Q_f$, and registers $V = (V_e \uplus V_f) \cup \{\text{acc}\}$. See figure 4.1e. The additional register acc is an accumulator, used to remember the result produced by M_e while in the M_f-sub-structure. The accepting states are $F = \{(1, q_{ff}) \mid q_{ff} \in F_f\}$, i.e. the accepting states of M_f. The output function appends the value of acc at the beginning of the output string for split(e, f): $\nu((1, q_{ff})) = \text{acc} \cdot \nu_f(q_{ff})$ for each accepting state q_{ff} of M_f, and at the end of the output string for left-split(e, f): $\nu((1, q_{ff})) = \nu_f(q_{ff}) \cdot \text{acc}$.

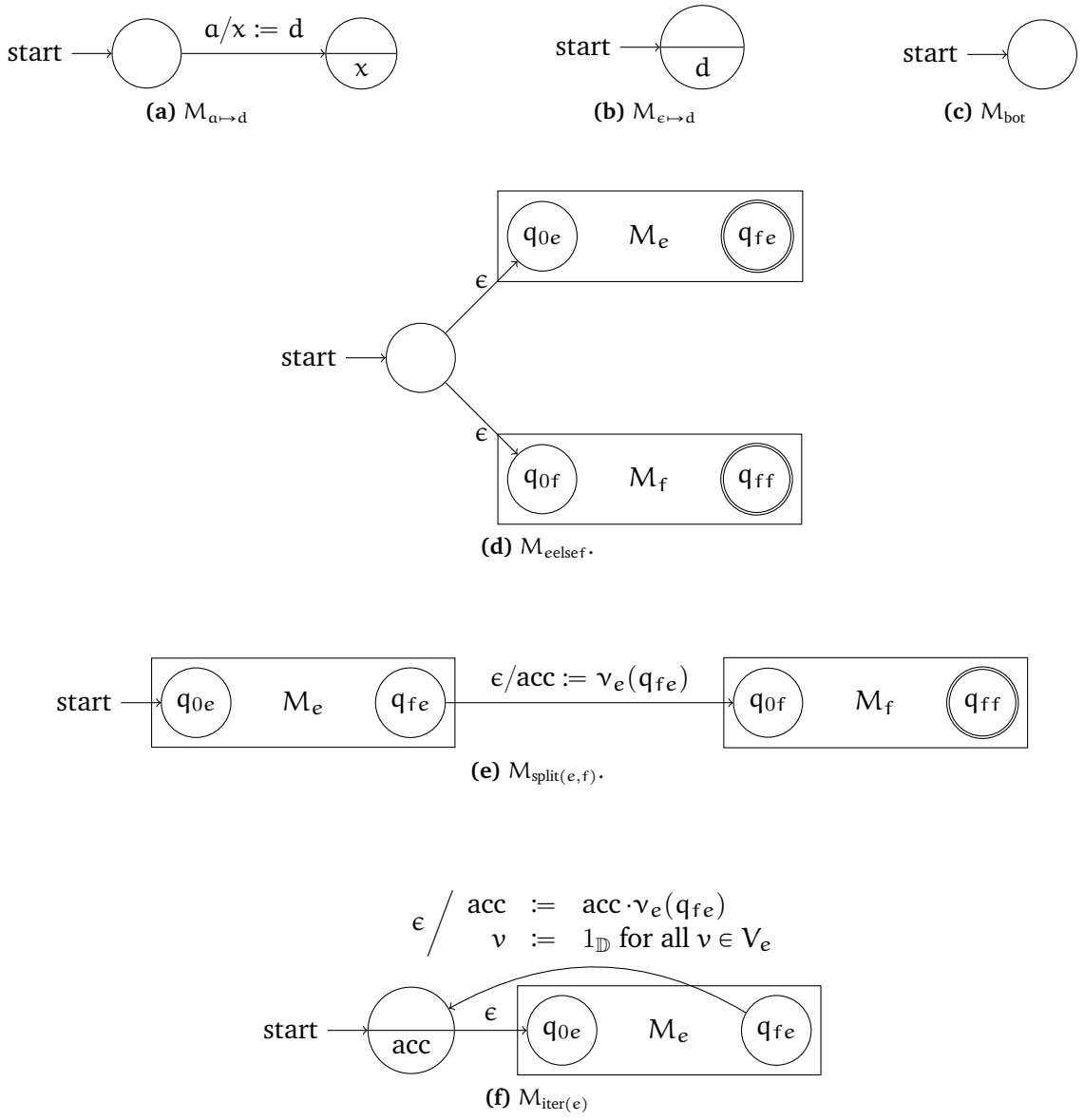
The constructed machine is copyless. Because the original expression split(e, f) is consistent, M is also unambiguous. Finally, by the induction hypothesis, every well-parsed string contains an accepting path through this machine, and vice-versa. Correctness follows.

iter(e) **and left-iter**(e)**:** The states of the machine are $Q = \{q_0\} \cup Q_e$, where q_0 is the initial and only accepting state of the machine. The combined result of iter(e) on the prefix are maintained in a register named acc. The machine is trivially copyless, and unambiguity follows from the consistency of the expression iter(e).

We will now argue that $M_{\text{iter}(e)}$ computes $[\![\text{iter}(e)]\!]$. First, notice that the underlying control structure of $M_{\text{iter}(e)}$ is an NFA which accepts $\text{Dom}(e)^*$. Next, consider an arbitrary input string $w \in \text{Dom}(e)^+$: by unambiguity, there is a unique split $w = w_{\text{pre}} w_{\text{post}}$, where $w_{\text{pre}} \in \text{Dom}(e)^*$ and $w_{\text{post}} \in \text{Dom}(e)$, and therefore a unique run leading to an accepting state q_{fe} of the component SST M_e. The induction hypothesis states that M_e computes $[\![e]\!]$: therefore the value of the output function $\nu_e(q_{fe})$ at the end of this run is equal to $[\![e]\!](w_{\text{post}})$. From this, the following inductive invariant can be proved: for every input string w and split $w = w_{\text{pre}} w_{\text{post}}$ such that $w_{\text{pre}} \in \text{Dom}(e)^*$ and $w_{\text{post}} \notin \text{Dom}(e)^*$, on the (necessarily unique) run corresponding to this split, the value of the register acc is equal to $[\![\text{iter}(e)]\!](w_{\text{pre}})$. It follows that the machine $M_{\text{iter}(e)}$ computes $[\![\text{iter}(e)]\!]$. □

Lemma 4.3. *If chain*(e, r) *(resp. left-chain*(e, r)*) is a consistent DReX expression, then there is an effectively constructible SST* M *which computes chain*(e, r) *(resp. left-chain*(e, r)*).*

We will not give a direct construction to prove lemma 4.3, but rather present a desugaring of the chained sum into the remaining operators *and* function composition. The proof follows by applying closure under function composition (theorem 3.3) to the remaining lemmas of



(a) $M_{a \mapsto d}$

(b) $M_{e \mapsto d}$

(c) M_{bot}

(d) M_{eelsef}.

(e) $M_{split(e,f)}$.

(f) $M_{iter(e)}$

Figure 4.1: Constructing unambiguous SSTs from DReX expressions.

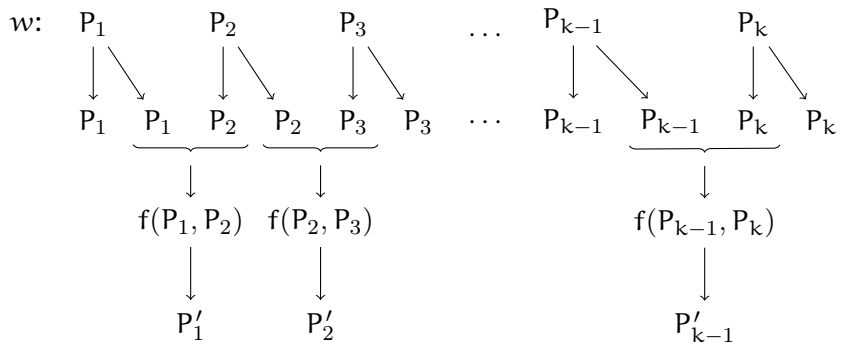


Figure 2.4: Expressing shuffle using function combinators. Each patch P_i is a a string of the form a^*b. (Repeated from page 23.)

this section. Besides its simplicity, this approach also highlights the related fact that function composition is an alternative to chained sum for the completeness result of chapter 5.

We present an example of the desugaring transformation with the function shuffle from example 2.6. We had finally expressed shuffle as:

$$\begin{aligned}\text{shuffle} &= \text{chain}(f, r), \text{ where}\\ &\quad f = \text{left-split}(\text{split}(\text{iter}(a \mapsto b), b \mapsto \epsilon), \text{split}(\text{iter}(a \mapsto a), b \mapsto \epsilon)), \text{ and}\\ &\quad r = a^*b.\end{aligned}$$

The idea is to map each stage of figure 2.4 to an expression without chained sum.

1. Let $\text{copy}_r = \text{combine}(\text{id}_r, \text{id}_r)$ where $\text{id}_r = \text{split}(\text{iter}(a \mapsto a), b \mapsto b)$. This expression maps strings $w \in [\![r]\!]$ to $w{\cdot}w$. The first stage of the transformation is given by $\text{iter}(\text{copy}_r)$.

2. The second stage of the transformation drops the first and last patches which match r in the intermediate result: $\text{drop}_r = \text{split}(r \mapsto \epsilon, \text{id}_r, r \mapsto \epsilon)$.

3. The last stage of the transformation $\text{iter}^+(f)$ iteratively applies f to substrings of the intermediate result. The combinator $\text{iter}^+(e)$ is syntactic sugar for $\text{split}(e, \text{iter}(e))$, similar to the "one-or-more" Kleene-+ operator.

Notice that all newly constructed expressions, $\text{iter}^+(e)$, drop_r and $\text{iter}(\text{copy}_r)$ are consistent. We can now write $[\![\text{shuffle}]\!] = [\![\text{iter}^+(e)]\!] \circ [\![\text{drop}_r]\!] \circ [\![\text{iter}(\text{copy}_r)]\!]$. More generally, given an arbitrary regular expression r, define:

$$\begin{aligned}\text{id}_r &= w \in [\![r]\!] \mapsto w,\\ \text{copy}_r &= \text{combine}(\text{id}_r, \text{id}_r), \text{ and}\\ \text{drop}_r &= \text{split}(r \mapsto \epsilon, \text{id}_r, r \mapsto \epsilon).\end{aligned}$$

The expression "$w \in [\![r]\!] \mapsto w$" is syntactic sugar for the identity function restricted to strings which match r and can be easily desugared into the (*non-chained sum*) combinators. If r is unambiguously iterable, then all three expressions just constructed are consistent, and the following claim can be easily proved:

Claim 4.4. For every pair of consistent expressions $\text{chain}(e, r)$ and $\text{left-chain}(e, r)$,

1. $[\![\text{chain}(e, r)]\!] = [\![\text{iter}^+(e)]\!] \circ [\![\text{drop}_r]\!] \circ [\![\text{iter}(\text{copy}_r)]\!]$, and
2. $[\![\text{left-chain}(e, r)]\!] = [\![\text{left-iter}^+(e)]\!] \circ [\![\text{drop}_r]\!] \circ [\![\text{iter}(\text{copy}_r)]\!]$.

Furthermore, all subexpressions in the above identities can be expressed without the chained sum combinators.

Proof of lemma 4.3. Follows from claim 4.4, lemma 4.2, and theorem 3.3. □

4.2 From QREs to SSTTs

We will now prove the second part of theorem 4.1, by showing that every QRE can be translated into an equivalent SSTT. Once again, the proof will proceed by induction on the QRE e. Most cases are similar to the constructions presented for lemma 4.2. However, unlike the case of DReX with the chained sum, in the present case, *all* constructions are similar to the regular-expression-to-NFA conversion procedure.

We will be presenting an unambiguous SSTT for each combinator in the framework: in particular, for each case, we will need to show that: (*a*) the constructed machine is unambiguous, (*b*) the register update and output functions are well-formed, i.e. that they are well-typed, single-use, and respect the parameter constraints, and (*c*) that the machine computes the intended function. We will split the construction into three lemmas: lemmas 4.5 and 4.6 cover the case of all combinators other than iteration, and lemma 4.7 will prove the proposition for the case of the iteration combinator.

Lemma 4.5. *For each input symbol $a \in \Sigma$, cost value $d \in \mathbb{D}$, and term t, there exist SSTTs which are equivalent to $a \mapsto d$, $\epsilon \mapsto t$, and bot.*

The constructions are identical to those used for the DReX-to-SST conversion in figures 4.1a, 4.1b, and 4.1c. The constructed the machines are clearly unambiguous, well-formed, and correctly compute the appropriate functions.

Lemma 4.6. *If e and f are consistent, well-typed, single-use QREs, and M_e and M_f are SSTTs which compute the functions $[\![e]\!]$ and $[\![f]\!]$ respectively, and if p and q are arbitrary parameters, then we can effectively construct SSTTs which are equivalent t the following QREs if they are also consistent, well-typed, and single-use: (a) e else f, (b) split$(e \to^p f)$ and split$(e \leftarrow^q f)$, (c) op(e, f), and (d) $e\{p := f\}$.*

The case for $\text{op}(e, f)$ naturally generalizes to operators with different arities.

Proof. Let Q_e and Q_f be the state spaces of M_e and M_f respectively. Recall that we define the disjoint union $A \uplus B = (\{0\} \times A) \cup (\{1\} \times B)$, i.e. the union the elements tagged with their origin. Despite its pedantic inaccuracy, for convenience, we will not distinguish between elements of $A \uplus B$ and the original elements of A and B, as the only difference is the addition of the origin tag.

e **else** f**:** The construction is identical to that in figure 4.1d for the corresponding DReX operator. The machine initially makes a non-deterministic decision to process the string either using machine M_e or using M_f.

We define $Q = \{q_0\} \cup (Q_e \uplus Q_f)$, and $V = V_e \uplus V_f$. The state transition relation is given by:

$$\Delta = \{(q_0, \epsilon, q_{0e}), (q_0, \epsilon, q_{0f})\} \cup \delta_e \cup \delta_f$$

Along the ϵ-transition out of q_0, all registers are unchanged:

$$\mu((q_0, \epsilon, q), \nu) = \nu.$$

Within the transitions of the component machine M_e (resp. M_f), registers $\nu \in V_e$ (resp. $\nu \in V_f$) are updated according to their original update expressions, and the remaining registers are left unchanged:

$$\mu((q, a, q'), \nu) = \begin{cases} \mu_e(q, a, \nu) & \text{if } (q, a, q') \in \delta_e, \text{ and } \nu \in V_e, \\ \mu_f(q, a, \nu) & \text{if } (q, a, q') \in \delta_f, \text{ and } \nu \in V_f, \text{ and} \\ \nu & \text{otherwise.} \end{cases}$$

The initial state is q_0, and all registers are initialized to their respective initial values in M_e and M_f:

$$\text{val}_0(\nu) = \begin{cases} \text{val}_{0e}(\nu) & \text{if } \nu \in V_e, \text{ and} \\ \text{val}_{0f}(\nu) & \text{otherwise.} \end{cases}$$

The accepting states $F = F_e \uplus F_f$, and each accepting state borrows its output expression from the machine of its provenance:

$$\nu(q) = \begin{cases} \nu_e(q) & \text{if } q \in F_e, \text{ and} \\ \nu_f(q) & \text{otherwise.} \end{cases}$$

By QRE consistency, the domains $\text{Dom}(e)$ and $\text{Dom}(f)$ are disjoint, and so the machine constructed is unambiguous. The register assigments are well-formed because the assignments of the component machines are assumed to be well-formed. Finally, every string w which is accepted by M is also accepted either by M_e or by M_f, and conversely, any string which is accepted by M_e or by M_f is also accepted by M. It can therefore be shown that the constructed machine M computes the desired function $[\![e \text{ else } f]\!]$.

split$(e \to^{p} f)$**:** See figure 4.1e for split(e, f). The present construction is similar. We define the state space $Q = Q_e \uplus Q_f$, and the register set $V = \{\text{acc}\} \cup (V_e \uplus V_f)$. The accumulator register acc is of type T_e, the output type of M_e, and $P_{\text{acc}} = \text{Param}(e)$.

For every transition $q \to^{a} q'$ of M_e and M_f, we include a corresponding transition in M, and additionally include ϵ-transitions from every final state of M_e to the initial state of M_f:

$$\Delta = \delta_e \cup \delta_f \cup \{(q_{fe}, \epsilon, q_{0f}) \mid q_{fe} \in F_e\}.$$

On transitions borrowed from M_e, the subset of registers V_e is updated according to original updates, the subset of registers $V_f \cup \{\text{acc}\}$ is carried along unchanged. On transitions borrowed from M_f, the subset of registers $V_e \cup \{\text{acc}\}$ is carried along unchanged, while registers from V_f are updated as per the original updates in M_f. On the transitions which "jump" from Q_e to Q_f, the accumulator is updated to the output of M_e: i.e. if $(q, a, q') \notin \delta_e \cup \delta_f$, then:

$$\mu((q, a, q'), \text{acc}) = \nu_e(q).$$

All other registers are reset to their original values during the Q_e–Q_f jump.

The initial state of M is q_{0e}, and the registers $v \in V_e \uplus V_f$ are initialized to their initial values from M_e and M_f. The accumulator acc is initialized to an arbitrary value of type T_e. The final states are $F = F_f$. The output function is given by substituting the contents of the accumulator for the parameter p in the original output expression for M_f:

$$\nu(q_f) = \nu_f(q_f)\{p := \text{acc}\}.$$

We know that the QRE split$(e \to^{p} f)$ is consistent, so that Dom(e) and Dom(f) are unambiguously concatenable: it follows that the constructed machine M is unambiguous. It can be verified that all reigster updates are well-formed, i.e. that they are well-typed, and obey the single-use constraints, and only invoke the appropriate parameters ($P_{\text{acc}} = \text{Param}(e)$). Finally, we claim the following invariants: (*a*) as long as the execution is within the subset Q_e of states, the values of registers $v \in V_e$ coincide with the values of M_e, (*b*) if the machine jumped from Q_e to Q_f after processing w_{pre}, and after that processed the suffix w_{post}, then the value of the accumulator acc is equal to $[\![e]\!](w_{\text{pre}})$, and the values of registers $v \in V_f$ coincide with the values of M_f on processing w_{post}. It can then be proved that the machine correctly computes the desired function, $[\![\text{split}(e \to^{p} f)]\!]$. A similar construction can be followed for split$(e \leftarrow^{q} f)$.

op(e, f)**:** We use the product construction. The state space of $M_{\text{op}(e,f)}$, $Q = Q_e \times Q_f$, and the registers are $V = V_e \uplus V_f$.

The transition relation $\Delta \subseteq Q \times \Sigma \times Q$ is given by:

$$\Delta = \{((q_e, q_f), a, (\delta_e(q_e, a), \delta_f(q_f, a))) \mid q_e \in Q_e, q_f \in Q_f, a \in \Sigma\}.$$

For each transition $((q_e, q_f), a, (q'_e, q'_f)) \in \Delta$, the register update expression is defined as follows:

$$\mu(((q_e, q_f), a, (q'_e, q'_f)), v) = \begin{cases} \mu_e(q_e, a, v) & \text{if } v \in V_e, \text{ and} \\ \mu_f(q_f, a, v) & \text{otherwise, if } v \in V_f. \end{cases}$$

The initial state $q_0 = (q_{0e}, q_{0f})$, and the initial register valuation is given by:

$$\text{val}_0(v) = \begin{cases} \text{val}_{0e}(v) & \text{if } v \in V_e, \text{ and} \\ \text{val}_{0f}(v) & \text{otherwise, if } v \in V_f. \end{cases}$$

The set of accepting states $F = F_e \times F_f$, and the output function is defined as:

$$\nu((q_e, q_f)) = \text{op}(\nu_e(q_e), \nu_f(q_f)).$$

The machine is unambiguous because the component machines are deterministic. The register update function lifts the register update expressions from the component machines, which are well-typed by assumption, and so the updates of M are also well-typed. The register updates for the subsets V_e and V_f are computed independently, and so the register updates are single-use and respect the parameter constraints.

Finally, every accepting run of M on a string w corresponds to simultaneous accepting runs of M_e and M_f. Therefore, the term $[\![M]\!](w)$ in this case coincides with the expected output term $\text{op}([\![M_e]\!](w), [\![M_f]\!](w))$. Conversely, every string w which is simultaneously accepted by M_e and M_f is also accepted by M: again, in this case, M produces the term $[\![M]\!](w)$ which is equal to the expected value $\text{op}([\![M_e]\!](w), [\![M_f]\!](w))$. M therefore correctly computes the intended QRE $\text{op}(e, f)$.

A similar construction can be followed for $e\{p \coloneqq f\}$. □

Lemma 4.7. *If $e = iter(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k)$ is a consistent, well-typed, single-use QRE and the machines M_1, M_2, …, M_k compute e_1, e_2, …, e_k respectively, then we can also construct an SSTT M which computes e.*

Proof. For simplicity, we focus on the two-expression case, $e = \text{iter}(e_1 \to p_1, e_2 \to p_2)$. At its heart, it is a combination of the product construction for $\text{combine}(e_1, e_2)$ and the simple iteration construct $\text{iter}(e_1)$. See figure 4.3.

The state space $Q = \{q_0\} \cup (Q_1 \times Q_2)$, and the register set $V = \{\text{acc}_1, \text{acc}_2\} \cup (V_1 \uplus V_2)$: the disjoint union of the registers of M_1 and M_2, and two new accumulators, acc_1 and acc_2. Within the product substructure, $Q_1 \times Q_2$, the state transition relation is defined as usual: $(q_1, q_2) \to^a (q'_1, q'_2)$ iff the transitions $q_1 \to^a q'_1$ and $q_2 \to^a q'_2$ occur in M_1 and M_2.

On the $q_0 \to^\epsilon (q_{01}, q_{02})$ transition, every register v_1 of M_1 is updated according to the initial register valuation $\text{val}_{01}(v_1)$, and every register v_2 of M_2 is updated according to its initial register valuation $\text{val}_{02}(v_2)$. During the return transition, $(q_{f1}, q_{f2}) \to^\epsilon q_0$, the results of M_1 and M_2 are loaded into the accumulators as shown in the figure. The remaining

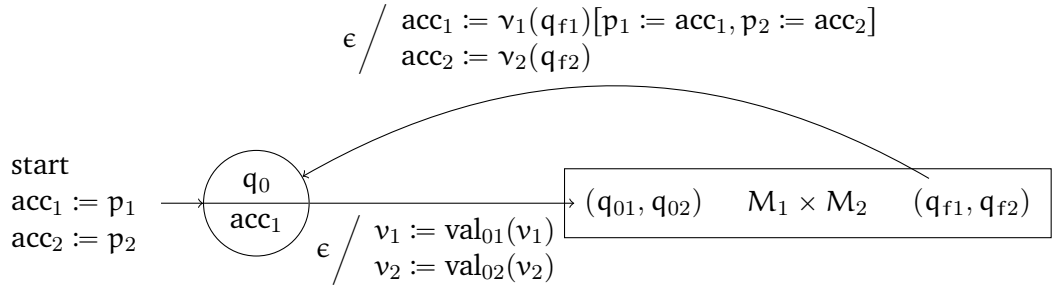


Figure 4.3: Unambiguous SSTT for $e = \mathrm{iter}(e_1 \to p_1, e_2 \to p_2)$. The register set $V = \{\mathrm{acc}_1, \mathrm{acc}_2\} \cup (V_1 \uplus V_2)$, where V_1 and V_2 are the register sets of the component machines. On the return transition $(q_{f1}, q_{f2}) \to q_0$, all other registers are arbitrarily reset.

registers are arbitrarily reset. The register acc_1 depends on the parameters $P_{\mathrm{acc}_1} = \{p_1, p_2\}$, and acc_2 depends on the parameters $P_{\mathrm{acc}_2} = \{p_2\}$.

Because $\mathrm{Dom}(e_1) = \mathrm{Dom}(e_2)$ and it is unambiguously iterable, the machine thus constructed is also unambiguous. The initial register valuation is clearly well-formed: each initial value is well-typed, each value is a single-use term, and the initial value of each register ν respects the parameter dependencies P_ν. Because the register sets are disjoint, the register updates within the product construction are also well-formed. Finally, it can also be verified that the return transition $(q_{f1}, q_{f2}) \to^{\epsilon} q_0$ is also a well-formed update—it is well-typed, single-use, and respects parameter sets. □

Chapter 5

Converting Transducers into Function Expressions

In the previous chapter, we presented algorithms to convert QREs and DReX expressions into equivalent transducers. In this chapter, we will prove the converse:

Theorem 5.1 (Completeness)**.** *1. For every SST* M*, we can construct an equivalent consistent DReX expression* e*.*

2. For every SSTT M*, we can construct an equivalent consistent, well-typed, and single-use QRE* e*.*

We will build on the classical algorithm to convert DFAs into equivalent regular expressions. In the traditional construction, the task is to build a representation for the set of strings,

$$L_{q,q'} = \{w \in \Sigma^* \mid q \to^w q'\},$$

which lead the automaton from the initial state q to the final state q'. In case of transducers, we are additionally interested in the way in which the register values are modified during execution: strings may be appended before and after the initial values, and register contents may be concatenated in complicated ways. We will perform a double induction, first by constraining the set of intermediate states the execution may visit, and then by constraining the patterns of register updates along the execution. This proof is therefore the most technically challenging argument of this thesis.

Except for the lowest levels, the construction for both QREs and DReX expressions are very similar. For ease of presentation, we will first present the proof in full detail for the case of SSTs, and describe the differences in the case of term transducers in section 5.11. For the rest of this chapter, we fix an SST $M = (Q, V, \Sigma, \mathbb{D}, \delta, \mu, q_1, F, \nu)$, where $Q = \{q_1, q_2, \ldots, q_n\}$ and $V = \{v_1, v_2, \ldots, v_k\}$, with $k \geqslant 1$.[1]

[1]The $k \geqslant 1$ assumption is needed because we will freely speak of the "domain of an expression vector", an intermediate object we will introduce in the construction, and this does not make sense if $k = 0$. Notice that theorem 5.1 is simple for SSTs with $V = \varnothing$, because they always output one of only a finite set of values.

5.1 Motivation

From DFAs to regular expressions. From the control structure of M, we can naturally extract a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_1, F)$. We first recap the classical algorithm [85] that transforms the DFA A into an equivalent unambiguous regular expression. Our algorithm to convert transducers into function expressions will piggyback on this translation; hence this review.

Recall that $Q = \{q_1, q_2, \ldots, q_n\}$. The idea is to iteratively compute, for each $i \in \{0, 1, 2, \ldots, n\}$, and for each pair of states $q, q' \in Q$, a regular expression $r^{(i)}(q, q')$, which matches exactly those non-empty strings w such that:

1. $q \to^w q'$, and
2. the path $\sigma = q \to^w q'$ only visits *intermediate*[2] states in the set $\{q_1, q_2, \ldots, q_i\}$.

These regular expressions are defined as follows:

$$r^{(0)}(q, q') = \{a \in \Sigma \mid q \to^a q'\}, \text{ and} \tag{5.1.1}$$

$$r^{(i+1)}(q, q') = r^{(i)}(q, q') + r^{(i)}(q, q_{i+1}) \cdot r^{(i)}(q_{i+1}, q_{i+1})^* \cdot r^{(i)}(q_{i+1}, q'). \tag{5.1.2}$$

The language accepted by A can now be represented by the regular expression:

$$r_A = \begin{cases} \epsilon + \sum_{q_f \in F} r^{(n)}(q_1, q_f) & \text{if } q_1 \in F, \text{ and} \\ \sum_{q_f \in F} r^{(n)}(q_1, q_f) & \text{otherwise.} \end{cases}$$

Each string w in $[\![r^{(i+1)}(q, q')]\!]$ either passes through q_{i+1} along the way, or it does not. If it does, then it can unambiguously be divided into substrings which match $r^{(i)}(q_{i+1}, q_{i+1})$ between subsequent visits to q_{i+1}. The constructed expression $r^{(i)}(q, q')$ is therefore unambiguous, for all i, q, q'. Furthermore, it only matches non-empty strings, and therefore, the final regular expression r_A is also unambiguous.

The "effect" of a string. The effect of processing a string is to change the state of the machine. In a DFA, this effect can be described by a table of the form:

$$\text{effect}(w) = \begin{cases} q_1 \mapsto q_1' \\ q_2 \mapsto q_2' \\ \cdots \\ q_n \mapsto q_n', \end{cases}$$

indicating the state q_i' in which the DFA would terminate, in case it processed w starting from q_i. The string w is entirely described by its effect: if w and w' have the same effect, then for all contexts w_{pre} and w_{post}, $w_{\text{pre}} \cdot w \cdot w_{\text{post}}$ is accepted by the DFA iff $w_{\text{pre}} \cdot w' \cdot w_{\text{post}}$ is also accepted by the machine.

[2] i.e. excluding the initial state q and terminal state q'.

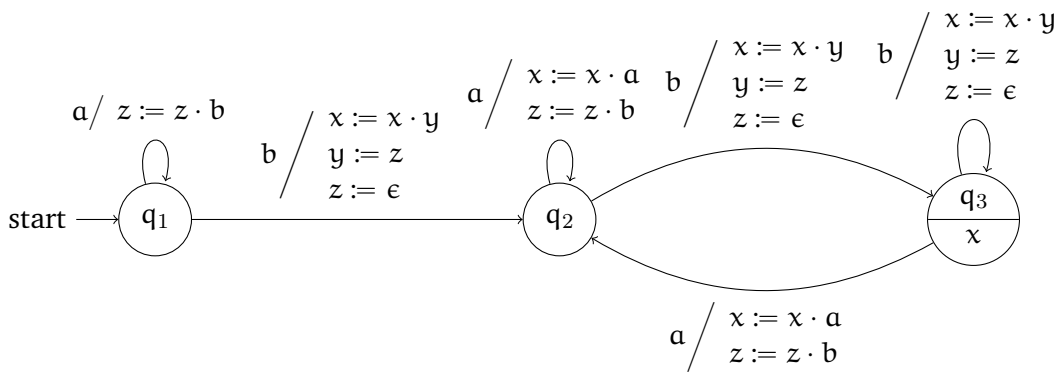


Figure 3.2: SST M_2 which computes the function shuffle from figure 2.3. (Repeated from page 47.)

The state space Q is finite, and so the space of effects $Q \to Q$ is also finite. The DFA-to-regular-expression construction, equations 5.1.1 and 5.1.2, classifies strings by their effect on the automaton.

In the case of SSTs, in addition to changing the control state, processing a string also results in the register values being updated. Recall the SST M_2 from figure 3.2. Consider the input string $baab$ processed from state q_2. The string may be summarized by the pair $(q_3 = \delta(q_2, baab), \{x \mapsto xyaaz, y \mapsto bb, z \mapsto \epsilon\})$, where $q_3 = \delta(q_2, baab)$ is the final state, and the register update $x \mapsto xyaaz$ indicates the final value of x in terms of the initial register valuation. See figure 5.2. Similarly, the input string aab, when processed starting from the state q_2, may be summarized as $(q_3, \{x \mapsto xaay, y \mapsto zbb, z \mapsto \epsilon\})$. Unlike in the case of DFAs, the space of effects is no longer finite, because of the unbounded nature of register updates.

String summaries. In step i of the transducer-to-function expression translation, we consider all strings $w \in [\![r^{(i)}(q, q')]\!]$, for each pair of states q, q'. By limiting our attention to strings in $[\![r^{(i)}(q, q')]\!]$, we have fixed the effect of w on the control state. The idea is to write DReX expressions which describe the effect of w on the register values. There are possibly many registers in the SST, and so we will construct a collection of DReX expressions, i.e. an "expression vector". Informally, the expression vectors will together summarize the effect of all strings $w \in [\![r^{(i)}(q, q')]\!]$.

Let us first concentrate on the patterns in which register values are updated during computation. For the strings $baab$ and aab, these are, respectively, $\{x \mapsto s_1 x s_2 y s_3 z s_4, y \mapsto s_5, z \mapsto s_6\}$ and $\{x \mapsto s_1 x s_2 y s_3, y \mapsto s_4 z s_5, z \mapsto s_6\}$, for some input-string dependent constants $s_1, s_2, \ldots, s_6 \in \mathbb{D}$. We call these patterns, $S : V \to V^*$, the "shapes" of the input strings. Observe that all input strings w of the form $a^* \cdot b$ starting from the state q_2 have the same shape, as we see in figure 5.2d.

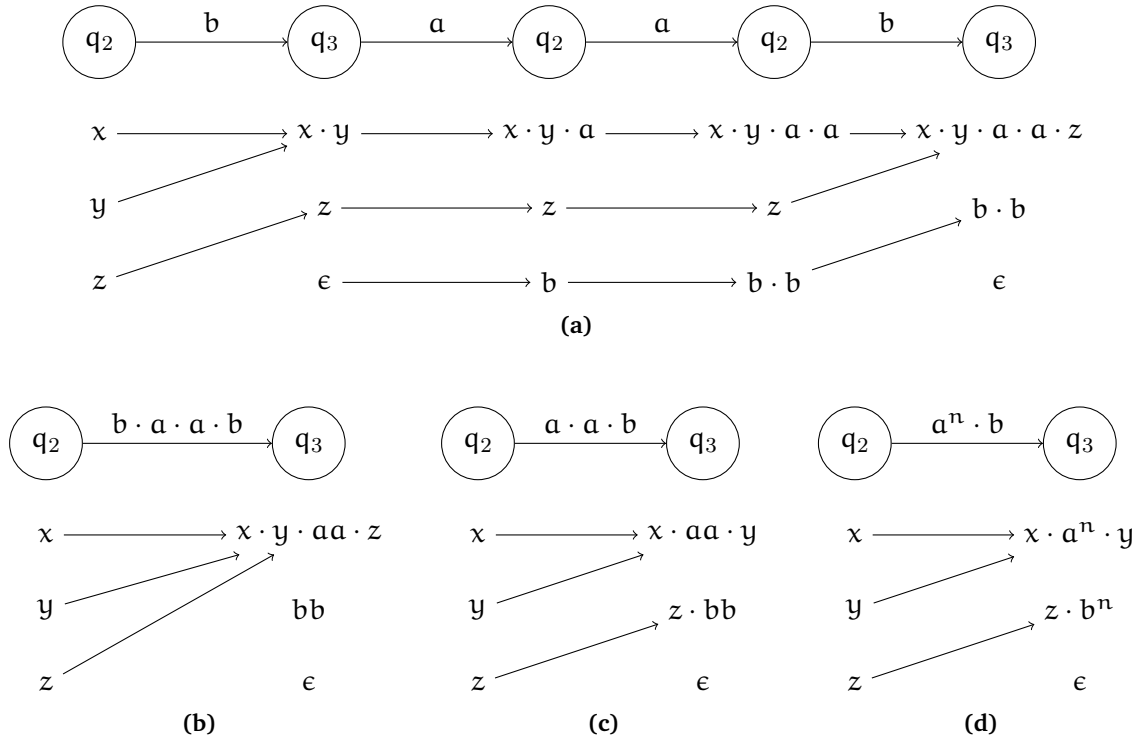


Figure 5.2: We are referring to the SST M_2 from figure 3.2. In figure 5.2a, we see the run of the string $baab$ starting from the state q_2. In figure 5.2b, we have collapsed it into a summary, showing only the final state and the final register values in terms of their original values. Figure 5.2c shows a similar summary for the string aab, starting from the state q_2. In fact, all strings of the form $a^* \cdot b$ share the same shape, as we see in figure 5.2d.

We will gather all strings w in $r^{(i)}(q, q')$ with a given shape S, and construct function expressions which map the input string w to the specific constants s_i in the shape. For example, consider strings w of the form $a^* \cdot b$, starting from the state q_2. The constants s_1, s_2, ..., s_6 are then computed by the following function expressions:

$$e_1 = e_3 = e_4 = e_6 = a^* \cdot b \mapsto \epsilon,$$
$$e_2 = \text{split}(\text{iter}(a \mapsto a), b \mapsto \epsilon), \text{ and}$$
$$e_5 = \text{split}(\text{iter}(a \mapsto b), b \mapsto \epsilon).$$

In step i of the transducer-to-function expression translation, for each shape S, we construct an *expression vector* $\mathbf{R}_S^{(i)}(q, q')$: this is a collection of DReX expressions, one for each string constant s_1, s_2, ... For each string $w \in [\![r^{(i)}(q, q')]\!]$ with shape S, the k-th element of the expression vector $e_k = \mathbf{R}_{S,k}^{(i)}(q, q')$ computes the value of s_k: $s_k = [\![e_k]\!](w)$.

As another example, consider the self-loop in M_2 at the state q_1 on the symbol a. All strings $w = a^n$ have the same shape $\{x \mapsto s_1 x s_2, y \mapsto s_3 y s_4, z \mapsto s_5 z s_6\}$. The specific string constants are:

$$s_1 = s_2 = s_3 = s_4 = s_5 = \epsilon, \text{ and}$$
$$s_6 = b^n.$$

Each constant s_m is computed by the function expression e_m, where:

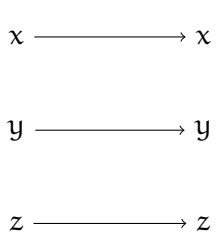
$$e_1 = e_2 = e_3 = e_4 = e_5 = a^* \mapsto \epsilon, \text{ and}$$
$$e_6 = \text{iter}(a \mapsto b).$$

Chapter outline. We will formally define shapes and expression vectors in section 5.2, and describe some basic operations over them in section 5.3. In section 5.4, we will explicitly construct the base expression vectors, $\mathbf{R}_S^{(0)}(q, q')$. In sections 5.5–5.10, we will describe the inductive construction of $\mathbf{R}_S^{(i+1)}(q, q')$. The chained sum is crucial to this construction: it will appear in the final step of the construction in section 5.9. In section 5.11, we will show how the proof technique can be generalized to handle the case of term transducers.

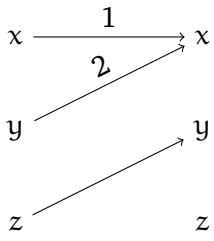
5.2 A Theory of Shapes

Definition 5.2. A SHAPE $S : V \to V^*$ is a copyless function over the set of registers V. Given an initial state q and an input string $w \in \Sigma^*$, let $\sigma = q \to^w q'$ be the corresponding path through M. The shape of the path σ is the function $S_\sigma : V \to V^*$ such that for all registers $v \in V$, $S_\sigma(v)$ is the projection onto V of the register update expression, $\mu(q, w, v)$: $S_\sigma(v) = \pi_V(\mu(q, w, v))$.

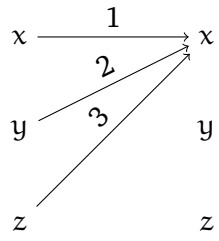
Because the register set V is finite, there are only a finite number of copyless functions, $S : V \to V^*$. The space of shapes is therefore finite.



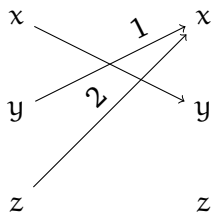
(a) $q_1 \to^a q_1$, $S_\bot$.



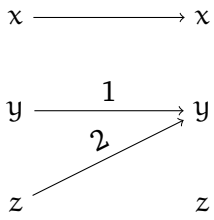
(b) $q_2 \to^b q_3$.



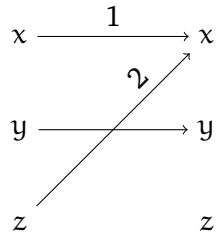
(c) $q_2 \to^{bb} q_3$, $S_\top$.



(d) Shape of the update $x \coloneqq yz$, $y \coloneqq x$, $x \coloneqq \epsilon$.



(e) Shape S_1 of the update $x \coloneqq x$, $y \coloneqq yz$, $z \coloneqq \epsilon$.



(f) Shape S_2 of the update $x \coloneqq xz$, $y \coloneqq y$, $z \coloneqq \epsilon$.

Figure 5.3: Visualizing shapes as bipartite graphs. Figures 5.3a–5.3c describe the shapes of some paths in the SST M_2 from figure 3.2. With the order $x \prec y \prec z$, only the shape in figure 5.3d is not upward-flowing. We will refer to the shapes in figures 5.3a, 5.3c, 5.3e, and 5.3f later in this chapter. For convenience, we name them $S_\bot$, $S_\top$, S_1, and S_2 respectively. The names $S_\bot$ and $S_\top$ are motivated by their position in the pre-order $\sqsubseteq$ defined in section 5.7.

Example 5.3. It is helpful to visualize shapes as bipartite graphs, as in figure 5.3. When multiple edges lead to the same vertex, such as $x \to x$, $y \to x$, and $z \to x$ in figure 5.3c, the numbers on the edges disambiguate the order: so $S_\top(x) = xyz$. An edge $u \to v$ can be informally read as "The value of u flows into v". Because of the copylessness restriction, every node on the left is connected to at most one node on the right. △

When two paths are concatenated, their shapes are combined. We define the concatenation $S_1 \cdot S_2$ of two shapes S_1 and S_2 as follows. Given a register $v \in V$, let $S_2(v) = v_1 v_2 \cdots v_k$. For each $i \in \{1, 2, \ldots, k\}$, define $s_i = S_1(v_i)$. Notice that $s_i \in V^*$ is a sequence of register names. Our definition of $(S_1 \cdot S_2)(v) \in V^*$ should also be a sequence of register names. We define

$$(S_1 \cdot S_2)(v) = s_1 s_2 \cdots s_k,$$

i.e. the concatenation of the individual sequences s_i. Informally, the concatenation of shapes corresponds to the composition of their bipartite graph visualizations. By definition, therefore:

Proposition 5.4. *Let $\sigma = q \to^{w} q'$ and $\sigma' = q' \to^{w'} q''$ be two paths through an SST M, such that the final state q' of σ is the same as the initial state of σ'. Then, for all registers v, $S_{\sigma\cdot\sigma'}(v) = (S_\sigma \cdot S_{\sigma'})(v)$.*

Let S be the shape of a path $\sigma = q \to^{w} q'$ through an SST. Consider a register v so that $S(v) = v_1 v_2 \cdots v_k$. The summary update for register v is therefore of the form $\mu(q, w, v) = s_1 v_1 s_2 v_2 \cdots v_k s_{k+1}$. We call each position (v, m), for $m \in \{1, 2, \ldots, |S(v)| + 1\}$ a PATCH in the corresponding shape.

Definition 5.5. An EXPRESSION VECTOR $\mathbf{A}$ for a shape S is a map from patches (v, m) to consistent DReX expressions, $\mathbf{A}_{v,m} : \Sigma^* \rightsquigarrow \mathbb{D}$, such that all the expressions have the same domain: $\mathrm{Dom}(\mathbf{A}_{v,m}) = \mathrm{Dom}(\mathbf{A}_{v',m'})$ for all patches (v, m) and (v', m').

Pick a language $L \subseteq \Sigma^*$, and a state $q \in Q$ such that all strings w when processed starting from q have the same shape S. An expression vector $\mathbf{A}$ SUMMARIZES L if (*a*) for each string $w \in L$ each patch (v, m) of S,

$$[\![\mathbf{A}_{v,m}]\!](w) = s_{v,m},$$

where $s_{v,m}$ is the constant appearing at position m in $\mu(q, w, v)$, and (*b*) for each component expression, $\mathbf{A}_{v,m}$, $\mathrm{Dom}(\mathbf{A}_{v,m}) = L$.

Example 5.6. Consider the loop a^* at the state q_2 of M_2. Consider the concrete string a^k. The effect of this string is to update $x := x \cdot a^k$, $y := y$, and $z := z \cdot b^k$. The shape of this set of paths is therefore the identity function $S = \{x \mapsto x, y \mapsto y, z \mapsto z\}$. Define the expression vector $\mathbf{A}$ as follows:

$$\mathbf{A}_{x,1} = \mathbf{A}_{y,1} = \mathbf{A}_{y,2} = \mathbf{A}_{z,1} = a^* \mapsto \epsilon,$$
$$\mathbf{A}_{x,2} = \mathrm{iter}(a \mapsto a), \text{ and}$$
$$\mathbf{A}_{z,2} = \mathrm{iter}(a \mapsto b).$$

Then $\mathbf{A}$ summarizes the set of paths a^* starting from the state q_2. △

We now restate the desired invariant (informally described in the proof outline in section 5.1):

Invariant 5.7. In step i of the SST-to-DReX expression translation, the expression vector $\mathbf{R}_S^{(i)}(q, q')$ summarizes all paths in $r^{(i)}(q, q')$ with shape S.

5.3 Operations on Expression Vectors

Choice. Let $\mathbf{A}$ and $\mathbf{B}$ be expression vectors, both for the same shape S, but with disjoint domains, $\mathrm{Dom}(\mathbf{A}) \cap \mathrm{Dom}(\mathbf{B}) = \varnothing$. The conditional choice, $\mathbf{A}$ else $\mathbf{B}$ is an expression vector for S defined as follows: for each patch (v, m) of S,

$$(\mathbf{A} \text{ else } \mathbf{B})_{v,m} = \mathbf{A}_{v,m} \text{ else } \mathbf{B}_{v,m}. \tag{5.3.1}$$

Each component expression $(\mathbf{A} \text{ else } \mathbf{B})_{v,m}$ is consistent because the sub-expressions have disjoint domains.

Proposition 5.8. *If* $L, L' \subseteq \Sigma^*$ *are disjoint sets of paths starting from the same state* q *such that all strings* $w \in L \cup L'$ *have the same shape* S*, and are summarized by the expression vectors* $\mathbf{A}$ *and* $\mathbf{A}'$ *respectively, then* $\mathbf{A}$ *else* $\mathbf{A}'$ *summarizes all paths in* $L \cup L'$*.*

If $\mathbf{A}_1$, $\mathbf{A}_2$, ... is a family of expression vectors with pairwise-disjoint domains, then we write $\sum \mathbf{A}_j$ for their combination using the else combinator ($\sum \mathbf{A}_j$ = bot if the family of component expression vectors is empty).

Shifting domains. Given a DReX expression e and a regular expression r, we write $\text{shift}(e, r)$ for the "left-shifted" expression which splits the input stream w into $w = w_1 \cdot w_2$, such that the suffix w_2 matches r, and applies e to the prefix w_1:

$$\text{shift}(e, r) = \begin{cases} \text{split}(e, r \mapsto \epsilon) & \text{if } [\![r]\!] \neq \varnothing, \text{ and} \\ \text{bot} & \text{otherwise.} \end{cases}$$

The expression is consistent if e is consistent, r is unambiguous, and $\text{Dom}(e)$ and $[\![r]\!]$ are unambiguously concatenable. Similarly, the "right-shifted" expression $\text{shift}(r, e)$ is defined as:

$$\text{shift}(r, e) = \begin{cases} \text{split}(r \mapsto \epsilon, e) & \text{if } \text{Dom}(e) \neq \varnothing, \text{ and} \\ \text{bot} & \text{otherwise.} \end{cases}$$

These "expression-level" operations, $\text{shift}(e, r)$ and $\text{shift}(r, e)$ can be naturally lifted to entire expression vectors $\text{shift}(\mathbf{A}, r)$ and $\text{shift}(r, \mathbf{A})$:

$$\text{shift}(\mathbf{A}, r)_{v,m} = \text{shift}(\mathbf{A}_{v,m}, r), \text{ and}$$
$$\text{shift}(r, \mathbf{A})_{v,m} = \text{shift}(r, \mathbf{A}_{v,m}).$$

Concatenation. Pick three states q, q', q'' in the given SST, and let L be a set of strings from q to q' with shape S, and L' be a set of strings from q' to q'' with shape S'. Then strings $w \in L \cdot L'$ transition from q to q'' and have shape $S \cdot S'$. Say the expression vectors $\mathbf{A}$ and $\mathbf{A}'$ summarize all paths in L and L' respectively. We will now construct an expression vector $\mathbf{A} \cdot \mathbf{A}'$ which summarizes all paths in $L \cdot L'$.

The idea is to left-shift the component expressions in $\mathbf{A}$ by $\text{Dom}(\mathbf{A}')$, right-shift the component expressions in $\mathbf{A}'$ by $\text{Dom}(\mathbf{A})$ and appropriately combine the results.We first demonstrate the construction with an example.

Example 5.9. As before, we are referring to the SST M_2 from figure 3.2. Consider the set of strings $L = [\![a^*]\!]$ which loop in the state q_2, and the set of strings $L' = [\![b \cdot b \cdot b^*]\!]$ which transition from q_2 to q_3. The shape of strings $w \in L$ is the identity function $S = \{x \mapsto x, y \mapsto y, z \mapsto z\}$, and the shape of strings $w' \in L'$ is $S' = \{x \mapsto x \cdot y \cdot z, y \mapsto \epsilon, z \mapsto \epsilon\}$. We have already constructed the expression vector $\mathbf{A}$ which summarizes L in example 5.6. The following expression vector $\mathbf{A}'$ summarizes strings in L':

$$\mathbf{A}'_{x,1} = \mathbf{A}'_{x,2} = \mathbf{A}'_{x,3} = \mathbf{A}'_{x,4} = \mathbf{A}'_{y,1} = \mathbf{A}'_{z,1} = b \cdot b \cdot b^* \mapsto \epsilon.$$

Now pick a string $w \cdot w'$ such that $w \in L$ and $w' \in L'$ and consider the run of M_2 starting from the state q_2. After processing w, the register values x', y' and z' are given by:

$$\begin{aligned}
x' &= x \cdot [\![\text{iter}(a \mapsto a)]\!](w) \\
&= x \cdot [\![\text{shift}(\text{iter}(a \mapsto a), b \cdot b \cdot b^*)]\!](w \cdot w'), \\
y' &= y, \text{ and} \\
z' &= z \cdot [\![\text{iter}(a \mapsto b)]\!](w) \\
&= z \cdot [\![\text{shift}(\text{iter}(a \mapsto b), b \cdot b \cdot b^*)]\!](w \cdot w').
\end{aligned}$$

After processing w', the register values x'', y'', and z'' are given by:

$$\begin{aligned}
x'' &= sx's y' s z' s, \\
y'' &= s, \text{ and} \\
z'' &= s, \text{ where} \\
s &= [\![b \cdot b \cdot b^* \mapsto \epsilon]\!](w') \\
&= [\![\text{shift}(a^*, b \cdot b \cdot b^* \mapsto \epsilon)]\!](w \cdot w').
\end{aligned}$$

The combined expression vector $\mathbf{A} \cdot \mathbf{A}'$, defined as

$$\begin{aligned}
&(\mathbf{A} \cdot \mathbf{A}')_{x,1} = (\mathbf{A} \cdot \mathbf{A}')_{x,3} = (\mathbf{A} \cdot \mathbf{A}')_{y,1} = (\mathbf{A} \cdot \mathbf{A}')_{z,1} = \text{shift}(a^*, b \cdot b \cdot b^* \mapsto \epsilon), \\
&(\mathbf{A} \cdot \mathbf{A}')_{x,2} = \text{combine}(\text{shift}(\text{iter}(a \mapsto a), b \cdot b \cdot b^*), \text{shift}(a^*, b \cdot b \cdot b^* \mapsto \epsilon)), \text{ and} \\
&(\mathbf{A} \cdot \mathbf{A}')_{x,4} = \text{combine}(\text{shift}(\text{iter}(a \mapsto b), b \cdot b \cdot b^*), \text{shift}(a^*, b \cdot b \cdot b^* \mapsto \epsilon))
\end{aligned}$$

summarizes all strings $w \cdot w' \in L \cdot L'$. △

We will now define the concatenation operation $\mathbf{A} \cdot \mathbf{A}'$ for arbitrary expression vectors $\mathbf{A}$ and $\mathbf{A}'$. As a first step, we define "shifted" expression vectors:

$$\begin{aligned}
\mathbf{A}_s &= \text{shift}(\mathbf{A}, \text{Dom}(\mathbf{A}')), \text{ and} \\
\mathbf{A}'_s &= \text{shift}(\text{Dom}(\mathbf{A}), \mathbf{A}').
\end{aligned}$$

Pick a register v and let

$$v := e_1 v_1 e_2 v_2 \cdots v'_l e_{l'+1}$$

be the update expression for v in $\mathbf{A}'_s$. For each register v_m in the right-hand side, let

$$v_m := e_{m,1} v_{m,1} e_{m,2} v_{m,2} \cdots v_{m,l} e_{m,l+1}$$

be the update expression in $\mathbf{A}_s$. View string concatenation as the expression combinator "combine", and substitute the expression for each v_m in $\mathbf{A}_s$ into the expression for v in $\mathbf{A}'_s$. Define $(\mathbf{A} \cdot \mathbf{A}')_{v,m}$ as the m-th expression in the string that results. The component expressions are consistent because $\text{Dom}(\mathbf{A}_s) = \text{Dom}(\mathbf{A}'_s)$.

Proposition 5.10. *For all states q, q' and q'', for every pair of states S, S', if the expression vectors $\mathbf{A}$ and $\mathbf{A}'$ summarize paths $L \subseteq \{w \in \Sigma^* \mid q \to^w q', S_w = S\}$ and $L' \subseteq \{w \in \Sigma^* \mid q' \to^w q'', S_w = S'\}$ respectively, then the expression vector $\mathbf{A} \cdot \mathbf{A}'$ summarizes all strings $w \in L \cdot L'$.*

5.4 The Base Case: $\mathbf{R}_S^{(0)}(q, q')$

Pick a symbol $a \in \Sigma$, and a pair of states $q, q' \in Q$. If $q \to^a q'$, and the shape of the (single transition) path is S, then define $\mathbf{R}_S^{(a)}(q, q')$ as follows. For each patch (v, k) in S:

$$\mathbf{R}_{S,v,k}^{(a)}(q, q') = a \mapsto s_k,$$

where s_k is the k-th constant appearing in the update expression $\mu(q, a, v)$. Otherwise, $\mathbf{R}_{S,v,k}^{(a)}(q, q') = \text{bot}$. The base expression vectors are defined as:

$$\mathbf{R}_S^{(0)}(q, q') = \sum_{a \in \Sigma} \mathbf{R}_S^{(a)}(q, q'). \tag{5.4.1}$$

By construction, we have the following proposition:

Proposition 5.11. *For each pair of states $q, q' \in Q$, and each shape S, $\mathbf{R}_S^{(0)}(q, q')$ summarizes all paths $w \in r^{(0)}(q, q')$ with shape S.*

5.5 Motivating the Inductive Step: $\mathbf{R}_S^{(i+1)}(q, q')$

We now have to construct $\mathbf{R}_S^{(i+1)}(q, q')$, which summarizes strings of the form $r^{(i+1)}(q, q')$ and with shape S. In section 5.3, we defined the choice operator over expression vectors: if $\mathbf{C}_S(q, q')$ summarizes strings of the form $r^{(i)}(q, q_{i+1}) \cdot r^{(i)}(q_{i+1}, q_{i+1})^* \cdot r^{(i)}(q_{i+1}, q')$, then we can define:

$$\mathbf{R}_S^{(i+1)}(q, q') = \mathbf{R}_S^{(i)}(q, q') \text{ else } \mathbf{C}_S(q, q'). \tag{5.5.1}$$

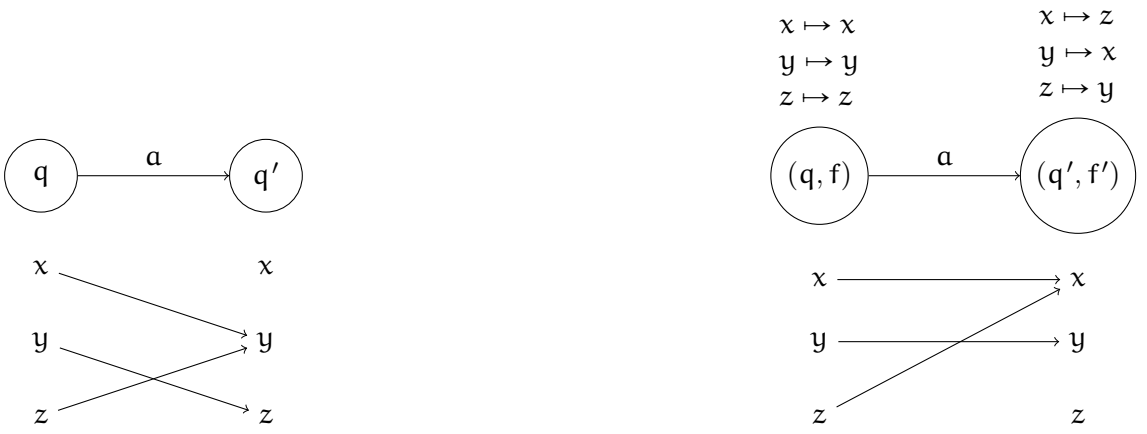
We have also defined the concatenation operator over expression vectors: if for each intermediate shape S_2, $\mathbf{B}_{S_2}$ summarizes strings matching $r^{(i)}(q_{i+1}, q_{i+1})^*$ with shape S_2, then we can write:

$$\mathbf{C}_S(q, q') = \sum_{S_1 \cdot S_2 \cdot S_3 = S} \mathbf{R}_{S_1}^{(i)}(q, q_{i+1}) \cdot \mathbf{B}_{S_2} \cdot \mathbf{R}_{S_3}^{(i)}(q_{i+1}, q'). \tag{5.5.2}$$

The consistency of $\mathbf{C}_S(q, q')$ and $\mathbf{R}_S^{(i+1)}(q, q')$ follows from the unambiguity of $r^{(i+1)}(q, q')$ in equation 5.1.2. Therefore, in the rest of the construction, our goal is to construct $\mathbf{B}_S$, for each shape S.

5.6 Ordering the Registers

Register values may flow in complicated ways: consider for example, the shape in figure 5.3d. The construction of $\mathbf{B}_S$ is greatly simplified if we can assume that register values only flow "upwards":



(a) Before, M. **(b)** After, M′.

Figure 5.4: Modifying an SST to mandate a register order. The annotations on each state of M' indicate the mapping between the "logical" registers and the "physical" registers. Proposition 5.13 says that, in this way, we can transform an arbitrary SST into one which is upward-flowing.

Definition 5.12. Fix a total order over the registers $\preceq$. A path $\sigma = q \to^w q'$ with shape S is UPWARD-FLOWING if for each pair of registers x, y, if y occurs in $S(x)$, then $x \preceq y$. The entire SST M is upward-flowing if each of its update expressions is also upward-flowing.

For example, in figure 5.3, with the order $x \prec y \prec z$, only the shape in figure 5.3d is not upward-flowing. If each of the transitions in a path is upward-flowing, then the entire path is itself upward-flowing.

Proposition 5.13. *For every SST M, there is an equivalent upward-flowing SST M'.*

Proof. The idea is to decouple the "logical" registers of M from the "physical" locations in which intermediate data is stored. We construct states which are pairs (q, f), where q is the corresponding state in M and the "register renaming" function $f : V \to V$ associates each logical register of M' with a physical register. We will construct the transition function $\delta' : Q' \times \Sigma \to Q'$, so that the register values flow only upwards during each transition. See figure 5.4.

Formally, we will construct a machine $M' = (Q', V, \Sigma, \mathbb{D}, \delta', \mu', q'_0, F', \nu')$, where the set of states, Q' consists of pairs (q, f), where $q \in Q$, and $f : V \to V$ is a bijection.

Let $f : V \to V$ be the register renaming function in the state (q, f). Consider a state transition $q \to^a q'$ in the original machine. The successor register renaming $f' : V \to V$ is obtained as follows. If $\mu(q, a, v)$ contains at least one register v', then define:

$$f'(v) = \min\{f(v') \mid v' \text{ occurring in } \mu(q, a, v)\}. \tag{5.6.1}$$

In the above expression, the minimum is taken over the register ordering $\preceq$. Informally, we have defined the physical register for v to be the minimum of the physical locations of all logical registers on which it depends. For all other registers v such that no registers occur in $\mu(q, a, v)$, arbitrarily assign $f'(v)$ so that f' is a bijection. This is always possible, because for all distinct v, v', such that both $f'(v)$ and $f'(v')$ are defined by equation 5.6.1, $f'(v) \neq f'(v')$ because of the copylessness of the original SST.

The register renaming function $f : V \to V$ can naturally be lifted to entire strings $\hat{f} : (V \cup \mathbb{D})^* \to (V \cup \mathbb{D})^*$:

$$\begin{aligned}\hat{f}(\epsilon) &= \epsilon,\\ \hat{f}(v \cdot l) &= f(v) \cdot \hat{f}(l), \text{if } v \in V, \text{ and}\\ \hat{f}(d \cdot l) &= d \cdot \hat{f}(l), \text{for } d \in \mathbb{D}.\end{aligned}$$

1. Define the state transition function δ' as follows: if $\delta(q, a) = q'$, then $\delta'((q, f), a) = (q', f')$.
2. Let $v_l = f'^{-1}(v)$ be the "true" logical register corresponding to the physical register v. Define the register update expression $\mu'((q, f), a, v) = \hat{f}(\mu(q, a, v_l))$.
3. Pick an arbitrary initial register renaming $f_0 : V \to V$ and let the initial state be $q'_0 = (q_0, f_0)$.
4. The final states $F' = \{(q, f) \mid q \in F\}$: a state (q, f) is accepting iff the original state q was itself accepting in M.
5. For each final state $(q, f) \in F'$, define the output function $\nu'((q, f)) = \hat{f}(\nu(q))$.

It can be verified that M and M' are equivalent, and that M' is upward-flowing. □

We will now assume that all SSTs and shapes under consideration are upward-flowing, and we elide this assumption in all definitions and theorems.

5.7 A Preorder over Shapes

We will now make the observation that some shapes cannot be used in the construction of other shapes. Consider the shapes S_1 and $S_\top$ from figure 5.3. Let $\sigma = q \to^w q'$ be a path through the SST with shape S_1. As a sample of the main result of this section, we will now argue that no sub-path of σ can have shape $S_\top$.

Assume otherwise, so that $\sigma = q \to^{w_1} q_1 \to^{w_2} q_2 \to^{w_3} q'$ where the sub-path $\sigma' = q_1 \to^{w_2} q_2$ has shape $S_\top$. Then, the values of all three registers at q_1 flow into q_2 at the end of σ', and therefore, along the prefix path $q \to^{w_1} q_1 \to^{w_2} q_2$, the initial value of y flows into x. We also know that the registers are only upward flowing, and therefore, once y has been "promoted" to x, it is impossible for the suffix $q_2 \to^{w_3} q'$ to restore this value to register y. But the shape of the entire path S_1 states that the initial value of y flows into the final value of y, leading to a contradiction.

In this section, we will define a pre-order $\sqsubseteq$ over the space of upward-flowing shapes which captures the notion of "can appear as a sub-path".

Definition 5.14. The SUPPORT of a shape S is defined as:

$$\text{Supp}(S) = \{v \in V \mid v \text{ occurs in } S(v)\}. \tag{5.7.1}$$

If S and S' are two shapes, then we say that $S \sqsubset S'$ iff $\text{Supp}(S) \supsetneq \text{Supp}(S')$. We call the shapes SUPPORT-EQUAL, and write $S \sim S'$ if $\text{Supp}(S) = \text{Supp}(S')$. Finally, $S \sqsubseteq S'$ iff either $S \sqsubset S'$ or $S \sim S'$.

For example, the shape $S_\perp$ from figure 5.3 is the bottom element of $\sqsubseteq$ and $S_\perp \sqsubset S_\top$. $S_1 \sim S_2$, and both shapes are strictly sandwiched between $S_\perp$ and $S_\top$. The relation $\sqsubseteq$ is easily verified to be a preorder: (*a*) every shape is support-equal with itself, and $\sqsubseteq$ is therefore reflexive, and (*b*) if $S \sqsubseteq S' \sqsubseteq S''$, then $\text{Supp}(S) \supseteq \text{Supp}(S') \supseteq \text{Supp}(S'')$, so that $S \sqsubseteq S''$: it is therefore also transitive. The following two claims, 5.15 and 5.16, formalize the intuition that $\sqsubseteq$ and $\sim$ describe the possible shapes of sub-paths.

Proposition 5.15. *Let $\sigma = q \to^w q'$ be a path through the SST M with shape S, and σ' be a subpath of σ with shape S'. Then $S' \sqsubseteq S$.*

Proof. Assume otherwise, so $S' \not\sqsubseteq S$. Then, $\text{Supp}(S') \not\supseteq \text{Supp}(S)$, so there is some register v such that $v \in \text{Supp}(S)$ and $v \notin \text{Supp}(S')$. The effect of the entire path σ is to make the initial value of v flow into itself, but on the sub-path σ', the value of v is either lost or promoted to some upper register v'. The first case (the initial value of v is lost during σ') immediately leads to a contradiction, while the second case (the initial value of v is promoted to some upper register v') leads to a contradiction together with the assumption that all paths through M are upward-flowing. □

Proposition 5.16. *Let there be k registers in the SST M. Pick $k+1$ strings, $w_1, w_2, \ldots, w_k, w' \in \Sigma^*$, and a path $\sigma = q^0 \to^{w_1} q^1 \to^{w_2} q^2 \to^{w_3} \cdots \to^{w_k} q^k \to^{w'} q'$ through the SST such that for each $i \in \{1, 2, \ldots, k\}$, the shape S_i of the subpath $q_{i-1} \to^{w_i} q_i$ is support-equal to the shape S of the entire path σ. Then the prefix sub-path $q^0 \to^{w_1 w_2 \cdots w_k} q^k$ also has shape S.*

Proof. Assume otherwise. Let $w = w_1 w_2 \cdots w_k$, so that $\sigma' = q^0 \to^w q^k$ has some shape $S' \neq S$. The concatenation of paths with support-equal shapes also produces a path with the same support: we conclude that $S' \sim S$. There are now three cases:

1. For some register $u \notin \text{Supp}(S)$, u flows into v in S, but does not flow into any register in S'. This indicates that the initial value of u is lost while processing the prefix w, but somehow regained while processing the suffix w' so that it can eventually flow into v. This is a contradiction.

2. For some register $u \notin \text{Supp}(S)$, there exist distinct registers $v \neq v'$, such that $u \to v$ in S and $u \to v'$ in S'.

Starting with u, consider the sequence of registers $u \to^{S_1} u_1 \to^{S_2} u_2 \to^{S_3} \cdots \to^{S_k} u_k$ through which the initial value of u flows. Notice that $u_k = v'$. All the shapes are upward-flowing, so according to the total order over the registers,

$$u \geq u_1 \geq u_2 \geq \cdots \geq u_k = v'.$$

If any of the above inequalities is non-strict, i.e. $u_l = u_{l+1}$, then $u_l \in \text{Supp}(S_{l+1})$, and the chain collapses by the assumption of support equality: $u_l = u_{l+1} = u_{l+2} = \cdots u_k = v'$, and therefore also $v = v'$, contradicting the assumption that $v \neq v'$. Therefore, all inequalities have to be strict. Now notice that the register-promotion chain $u \gneq u_1 \gneq u_2 \gneq \cdots \gneq u_k$ is $k+1$ elements long, but there are only but there are only k registers in the SST. This again leads to a contradiction.

3. For some register $v \in \text{Supp}(S)$, the order of registers in $S(v)$ and $S'(v)$ are different. For some registers u, w, the register u occurs before w in $S(v)$, but u occurs after w in $S'(v)$. However, once the values of u and w have been appended to v in the order wu, the suffix $q^k \to^{w'} q'$ cannot separate them to be recast in the order uw. It is thus a contradiction that u occurs before w in $S(v)$. □

The previous proposition also highlights the importance of stable shapes: an upward-flowing shape S is STABLE if for all registers u, v, if $u \to v$ in S, then $v \to v$. By an argument similar to that used in case 2 above, it can be proved that:

Proposition 5.17. *Let there be k registers in the SST M. Let $\sigma = q^0 \to^{w_1} q^1 \to^{w_2} q^2 \to^{w_3} \cdots \to^{w_k} q^k$ be a path through the SST such that for each $j \in \{1, 2, \ldots, k\}$, the shape S_j of the subpath w_j is support-equal to the shape S of the entire path σ. Then the shape S is stable.*

Informally, non-stable shapes cannot be generated by "very long" loops, and therefore, the most important case in the construction of $\mathbf{B}_S$ is that of stable shapes.

5.8 Decomposing Loops: The S-Decomposition

Starting with section 5.5, our goal has been to construct $\mathbf{B}_S$, for each shape S. $\mathbf{B}_S$ summarizes all strings of the form $r^{(i)}(q_{i+1}, q_{i+1})^*$, and with shape S. In section 5.6 we restricted the space of shapes to only those that are upward-flowing, and in section 5.7, we defined a partial order $\sqsubseteq$ over shapes which constrained the possible shapes of sub-paths. In this section and in section 5.9, we will construct $\mathbf{B}_S$. Our construction of $\mathbf{B}_S$ is inductive: we assume that $\mathbf{B}_{S'}$ is known for all strictly smaller shapes $S' \sqsubsetneq S$. Furthermore, since each string in $r^{(i)}(q_{i+1}, q_{i+1})$ is non-empty, $\epsilon \notin [\![r^{(i)}(q_{i+1}, q_{i+1})^+]\!]$. We therefore separately handle the case of ϵ and we will now construct an expression vector $\mathbf{B}^+_S$ which summarizes strings (all necessarily non-empty) in $r^{(i)}(q_{i+1}, q_{i+1})^+$ and with shape S. Define the expression vector $\mathbf{B}^\epsilon_S$ as follows:

$$\mathbf{B}^\epsilon_{S,v,i} = \begin{cases} \epsilon \mapsto \epsilon & \text{if } S = \{x \mapsto x, y \mapsto y, z \mapsto z\}, \text{ and} \\ \text{bot} & \text{otherwise.} \end{cases}$$

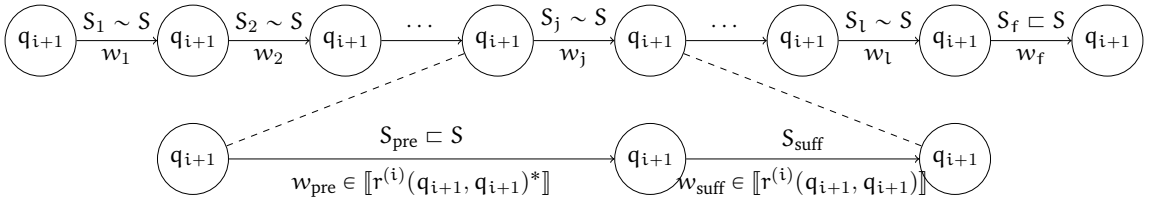


Figure 5.5: Decomposing paths in $r^{(i)}(q_{i+1}, q_{i+1})^+$ whose shape is S. Each $w \in \{w_1, w_2, \ldots, w_l\}$ has shape $S_w \sim S$, and each of its proper prefixes w_{pre} has shape $S_{pre} \sqsubsetneq S$. Equivalently $w \in L_{first}(S_w)$. Each string $w \in \{w_1, w_2, \ldots, w_l\}$ can itself be unambiguously written as $w = w_{pre} \cdot w_{suff}$, with $w_{suff} \in [\![r^{(i)}(q_{i+1}, q_{i+1})]\!]$ being a single iteration of the $q_{i+1} \to q_{i+1}$ loop.

Then we can write:

$$\mathbf{B}_S = \mathbf{B}_S^{\epsilon} \text{ else } \mathbf{B}_S^{+}. \tag{5.8.1}$$

S-decompositions. Consider any path $w \in [\![r^{(i)}(q_{i+1}, q_{i+1})^+]\!]$ with shape S. From propositions 5.15 and 5.16, we can unambiguously decompose $w = w_1 w_2 \cdots w_l w_f$, where:

1. each substring $w' \in \{w_1, w_2, \ldots, w_l, w_f\}$ is a self-loop at q_{i+1}: $w' \in [\![r^{(i)}(q_{i+1}, q_{i+1})^+]\!]$,
2. each sub-path $q_{i+1} \to^{w'} q_{i+1}$ for $w' \in \{w_1, w_2, \ldots, w_l\}$ has shape $S_{w'}$ such that $S_w \sim S$, and for each proper looping prefix w'' of w' (i.e. such that $|w''| \lneq |w'|$ and $q_{i+1} \to^{w''} q_{i+1}$), $S_{w''} \sqsubsetneq S$, and
3. the final sub-path w_f is also a self-loop: $w_f \in [\![r^{(i)}(q_{i+1}, q_{i+1})^*]\!]$, but has a strictly smaller shape $S_f \sqsubsetneq S$. (Note that the final sub-path w_f is possibly even empty.)

We call this split $w = w_1 w_2 \cdots w_l w_f$ the S-DECOMPOSITION of w. See figure 5.5.

Summarizing the segments of the S-decomposition. We will now construct expression vectors $\mathbf{A}_{S'}$ which summarize these minimal sub-paths $w' \in \{w_1, w_2, \ldots, w_l\}$. For each shape $S' \sim S$, let $L_{first}(S')$ be the set of non-empty strings $w' \in [\![r^{(i)}(q_{i+1}, q_{i+1})^+]\!]$ such that all proper prefixes w_{pre} of w' have shape $S_{pre} \sqsubsetneq S'$. Observe that for each $w' \in \{w_1, w_2, \ldots, w_l\}$, $w' \in L_{first}(S_{w'})$. The string w' can itself be unambiguously written as $w' = w_{pre} w_{suff}$, where $w_{suff} \in r^{(i)}(q_{i+1}, q_{i+1})$ is a single iteration of the $q_{i+1} \to q_{i+1}$ loop. If w_{pre} has shape S_{pre}, and w_{suff} has shape S_{suff}, then they are summarized by the expression vectors $\mathbf{B}_{S_{pre}}$ and $\mathbf{R}^{(i)}_{S_{suff}}(q_{i+1}, q_{i+1})$ respectively. We have already constructed both of these expression vectors by the induction hypothesis. Therefore, we can define:

$$\mathbf{A}_{S'} = \sum_{S_1 \cdot S_2 = S', S_1 \sqsubsetneq S'} \mathbf{B}_{S_1} \cdot \mathbf{R}^{(i)}_{S_2}(q_{i+1}, q_{i+1}). \tag{5.8.2}$$

By construction, we have:

Proposition 5.18. *For each shape $S' \sim S$, the expression vector $\mathbf{A}_{S'}$ summarizes all paths in $L_{first}(S')$.*

Short and long S-decompositions. Pick a small constant l_0, and consider strings w such that $q_{i+1} \to^w q_{i+1}$ with shape S and with short S-decomposition $w = w_1 w_2 \cdots w_l w_f$, i.e. $l \leqslant l_0$. All such strings are easy to summarize: for each $l \in \{1, 2, \ldots, l_0\}$, and for each shape $S' \sim S$, we define:

$$\begin{aligned}
\mathbf{A}^1_{S'} &= \mathbf{A}_{S'}, \text{ and} \\
\mathbf{A}^{l+1}_{S'} &= \sum_{S_1 \cdot S_2 = S'} \mathbf{A}^i_{S_1} \cdot \mathbf{A}_{S_2}, \text{ so that} \\
\mathbf{A}^{\leq l}_{S'} &= \sum_{j=1}^{l-1} \mathbf{A}^j_{S'}.
\end{aligned} \tag{5.8.3}$$

This still leaves the problem of strings with arbitrarily long S-decompositions. The following result, which is an immediate consequence of proposition 5.17, is an important first step. Recall that we defined a stable shape S as one with the property that if $u \to^S v$ then $v \to^S v$.

Proposition 5.19. *Let there be k registers in the SST M. Pick a string $w \in [\![r^{(i)}(q_{i+1}, q_{i+1})^+]\!]$ with shape S. Let $w = w_1 w_2 \cdots w_l w_f$ be its S-decomposition. Then for each sequence of k consecutive segments, $w' = w_j w_{j+1} \cdots w_{j+k-1}$, $S_{w'}$ is stable.*

There are two main consequences of this result:

1. For non-stable shapes, $l < k$, and they are therefore easy to summarize.
2. If the S-decomposition is divided into "k-segments", each composed of k consecutive segments, $w = (w_1 w_2 \cdots w_k) \cdot (w_{k+1} w_{k+2} \cdots w_{2k}) \cdots (w_{l'}) w_f$, then each k-segment

   $$w_{s,m} = w_{mk+1} w_{mk+2} \cdots w_{(m+1)k}$$

   attains a stable shape. See figure 5.6.

   (a) Every non-support register $u \notin \text{Supp}(S)$ is reset while processing $w_{s,m}$. The ultimate value of u is therefore a function of the suffix $w_{l-k+1} w_{l-k+2} \cdots w_l w_f$.

   (b) If $v \in \text{Supp}(S)$, then every incoming register while processing $w_{s,m}$ was reset while processing the previous segment $w_{s,m-1}$. The value appended to v while processing $w_{s,m}$ is therefore *entirely determined* by $w_{s,m-1} \cdot w_{s,m}$.
   We will write an expression f which maps $w_{s,m-1} w_{s,m}$ to the value appended to v after processing $w_{s,m}$. The expression of interest will then be $\text{chain}(f, r)$, with $r = \sum_{S' \sim S} \text{Dom}(\mathbf{A}^k_{S'})$.

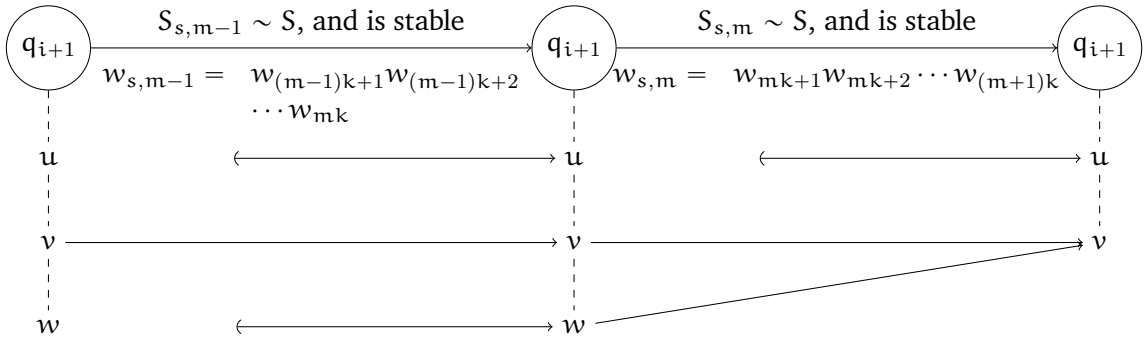


Figure 5.6: Analyzing data flows while iterating k-segments. The entire path $q_{i+1} \to^w q_{i+1}$ has shape S, which is stable. The S-decomposition $w = w_1 w_2 \cdots w_l w_f$ is divided into k-segments $w_{s,m} = w_{mk+1} w_{mk+2} \cdots w_{(m+1)k}$ of k consecutive iterations. In this figure, we show two consecutive k-segments, $w_{s,m-1} w_{s,m}$. By proposition 5.19, each k-segment itself has a stable shape.

5.9 Constructing $\mathbf{B}_S^+$

Recall that the expression vector $\mathbf{B}_S^+$ should summarize non-empty strings w of the form $r^{(i)}(q_{i+1}, q_{i+1})^+$ and with shape S. We currently have two induction hypotheses:

1. $\mathbf{B}_{S'}$ is known for all strictly smaller shapes, $S' \sqsubsetneq S$, and
2. $\mathbf{R}_{S'}^{(i)}(q, q')$ is known for each shape S' and each pair of states q, q'.

For each support-equal shape $S' \sim S$, we have constructed the expression vector $\mathbf{A}_{S'}$ in equation 5.8.2: $\mathbf{A}_{S'}$ summarizes minimal strings w such that (*a*) $q_{i+1} \to^w q_{i+1}$, (*b*) $S_w = S'$, and (*c*) for all looping prefixes w' of w, $S_{w'} \sqsubsetneq S$. In this section, we will construct $\mathbf{B}_S^+$.

Non-stable shapes. Let S be a non-stable shape, and $w \in [\![r^{(i)}(q_{i+1}, q_{i+1})^*]\!]$ be a loop with shape S. Let the S-decomposition of w be $w = w_1 w_2 \cdots w_l w_f$. By proposition 5.17, $l < k$, where k is the number of registers in the machine. $\mathbf{A}_S^{\leqslant k}$ from equation 5.8.3 is therefore sufficient to solve this case:

$$\mathbf{B}_S^+ = \sum \{ \mathbf{A}_{S_1}^{<k} \cdot \mathbf{B}_{S_2} \mid S_1 \cdot S_2 = S, S_1 \sim S, S_2 \sqsubsetneq S \}. \tag{5.9.1}$$

S is stable, and $S(v) = \epsilon$. Informally, this is the case when v is reset in S, and therefore also reset during each minimal sub-path w which is support-equal to S. Therefore, the final value of the register v is entirely determined by the substring $w_{l-k+1} w_{l-k+2} \cdots w_l \cdot w_f$. Register

u in figure 5.6 is a visualization of this situation. Define:

$$\begin{aligned}
\mathbf{H} &= \sum\{\mathbf{A}_{S_1}^{<2k} \cdot \mathbf{B}_{S_2} \mid S_1 \cdot S_2 = S, S_1 \sim S, S_2 \sqsubsetneq S\} \text{ (for "short" strings } w\text{)},\\
f &= \sum\{(\mathbf{A}_{S_1}^{k} \cdot \mathbf{B}_{S_2})_{v,1} \mid S_1 \sim S, S_2 \sqsubsetneq S\}, \text{ and}\\
r &= \mathrm{Dom}(\mathbf{A}_S^k) \cdot (\sum_{S' \sim S} \mathrm{Dom}(\mathbf{A}_{S'}))^*, \text{ so that}\\
\mathbf{B}^+_{S,v,1} &= \mathbf{H}_{v,1} \text{ else shift}(r, f).
\end{aligned} \tag{5.9.2}$$

By proposition 5.16, shifting by $\mathrm{Dom}(\mathbf{A}_S^k)$ ensures that the resulting path has shape S. It can be verified that the sub-expressions of the else have disjoint domains ($l < 2k$ and $l \geqslant 2k$ respectively), and mutually exhaust the cases when the total path w has shape S. Consistency follows.

Internal patches: $S(v) \neq \epsilon$, but $1 < m < |S(v)| + 1$. This corresponds to the case when m is an *internal* patch of $S(v)$. Consider the update expressions

$$\begin{aligned}
x &:= xaby,\\
y &:= b,
\end{aligned}$$

and

$$\begin{aligned}
x &:= axbyc,\\
y &:= b.
\end{aligned}$$

Both updates have the same shape $\{x \mapsto xy, y \mapsto \epsilon\}$. Concatenating them also yields an update expression with the same shape:

$$\begin{aligned}
x &:= axabybbc,\\
y &:= b.
\end{aligned}$$

Note that the internal patch $s_{x,2} = ab$, and observe that its value has been fixed by the first update, because once the register x and y have been combined in S, any changes to the register value can only be at the beginning or at the end of the string. It follows that the value of m-th patch in the final update expression for v is determined entirely by $w_1 w_2 \cdots w_k$. We can therefore write:

$$\begin{aligned}
\mathbf{H} &= \sum\{\mathbf{A}_{S_1}^{<k} \cdot \mathbf{B}_{S_2} \mid S_1 \cdot S_2 = S, S_1 \sim S, S_2 \sqsubsetneq S\}, \text{ and}\\
r &= (\sum_{S' \sim S} \mathrm{Dom}(\mathbf{A}_{S'}))^* \cdot \sum_{S' \sqsubsetneq S} \mathrm{Dom}(\mathbf{B}_{S'}), \text{ so that}\\
\mathbf{B}^+_{S,v,m} &= \mathbf{H}_{v,m} \text{ else shift}(\mathbf{A}^k_{S,v,m}, r).
\end{aligned} \tag{5.9.3}$$

The expression is consistent because the sub-expressions have disjoint domains, and they mutually exhaust all cases where the total path has shape S.

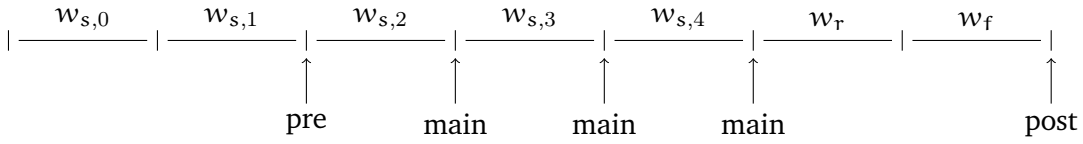


Figure 5.7: Summarizing the extremal patches: S is stable, $S(v) \neq \epsilon$, and $m = |S(v)| + 1$. We are summarizing strings with at least 3 segments, so in the S-decomposition $w = w_1 w_2 \cdots w_l w_f$, $l \geqslant 3k$. We write three expressions pre, main, and post, which compute the value appended to v during each of the iterations of the S-decomposition.

External patches: $S(v) \neq \epsilon$, and $m = 1$ or $m = |S(v)| + 1$. This is the case when m is either the first or the last patch in $S(v)$. As discussed in figure 5.6, the value appended to v while processing $w_{s,m}$ is determined by $w_{s,m-1} w_{s,m}$. We will define $\mathbf{B}^{+}_{S,v,m}$ for the case when $m = |S(v)| + 1$. The case for $m = 1$ is symmetric. We will temporarily assume that the S-decomposition is long, with $l \geqslant 3k$. See figure 5.7.

1. In the first two segments $w_1 w_2 \cdots w_k$ and $w_{k+1} w_{k+2} \cdots w_{2k}$ of the S-decomposition, the value appended to the end of register v is given by:

$$\begin{aligned} \text{pre} &= \text{shift}(\mathbf{A}^{2k}_{S,v,m}, r), \text{ where} \\ r &= \Big(\sum_{S' \sim S} \text{Dom}(\mathbf{A}_{S'})\Big)^{+} \cdot \sum_{S' \sqsubsetneq S} \text{Dom}(\mathbf{B}_{S'}). \end{aligned} \tag{5.9.4}$$

2. Let the segments be $w_{s,0}, w_{s,1}, \ldots, w_{s,l'}$, leaving a suffix $w_r w_f$, where $S_{w_r} \sim S$ (but with an S-decomposition strictly less than k iterations) and $S_{w_f} \sqsubsetneq S$. We now calculate the value appended to v during each segment $w_{s,2}, w_{s,3}, \ldots, w_{s,l'}$. Pick a pair of stable shapes $S_p, S_c \sim S$, and say that the segments $w_{s,m-1}$ and $w_{s,m}$ in figure 5.6 have shapes S_p and S_c respectively. We will now write an expression for the value appended to v at the end of $w_{s,m}$. Define:

$$\begin{aligned} \mathbf{V}_c &= \text{shift}(\text{Dom}(\mathbf{A}^{k}_{S_p}), \mathbf{A}^{k}_{S_c}), \text{ and} \\ \mathbf{V}_p &= \text{shift}(\mathbf{A}^{k}_{S_p}, \text{Dom}(\mathbf{A}^{k}_{S_c})). \end{aligned}$$

   Consider the sequence σ of expressions and registers occurring after v in $\mathbf{V}_{c,v}$, and substitute the expression $\mathbf{V}_{p,u}$ for every register u in σ. View expression concatenation as the DReX operator combine, and define $f(S_p, S_c)$ as the expression which results. The expression main computes the value appended to v after processing each segment

$w_{s,2}, w_{s,3}, \ldots w_{s,l'}$:

$$\begin{aligned} \text{main} &= \text{shift}(\text{Dom}(\mathbf{A}_S^k), \text{chain}(f, r), r_{\text{post}}), \text{ where} \\ f &= \sum\{f(S_p, S_c) \mid S_p, S_c \sim S, \text{ both stable}\}, \\ r &= \sum\{\text{Dom}(\mathbf{A}_{S'}^k) \mid S' \sim S \text{ and stable}\}, \text{ and} \\ r_{\text{post}} &= (\epsilon + \sum_{S' \sim S} \text{Dom}(\mathbf{A}_{S'}^{<k})) \cdot \sum_{S' \sqsubsetneq S} \text{Dom}(\mathbf{B}_{S'}). \end{aligned} \tag{5.9.5}$$

3. In figure 5.7, w_r is the string left over when the S-decomposition is grouped into segments of k iterations each. This is an empty string exactly when l is a multiple of k. Corresponding to the two cases where it is empty and not-empty, we write expressions f_2 and f_3. Pick a shape $S_{l'} \sim S$ and $S_f \sqsubsetneq S$. Define:

$$\begin{aligned} \mathbf{U}_f^2 &= \text{shift}(\text{Dom}(\mathbf{A}_{S_{l'}}^k), \mathbf{B}_{S_f}), \text{ and} \\ \mathbf{U}_{l'}^2 &= \text{shift}(\mathbf{A}_{S_{l'}}^k, \text{Dom}(\mathbf{B}_{S_f})). \end{aligned}$$

Consider the sequence σ of expressions and registers which follow v in $\mathbf{U}_{f,v}^2$, and replace every occurrence of a register u with $\mathbf{U}_{l',u}^2$. View concatenation as the DReX operation combine, and define $f_2(S_{l'}, S_f)$ as the expression which results. Let:

$$f_2 = \sum\{f_2(S_{l'}, S_f) \mid S_{l'} \sim S, S_f \sqsubsetneq S\}.$$

Similarly, pick three shapes $S_{l'}, S_r \sim S$, and $S_f \sqsubsetneq S$. Define:

$$\begin{aligned} \mathbf{U}_f^3 &= \text{shift}(\text{Dom}(\mathbf{A}_{S_{l'}}^k), \mathbf{A}_{S_r}^{<k} \cdot \mathbf{B}_{S_f}), \text{ and} \\ \mathbf{U}_{l'}^3 &= \text{shift}(\mathbf{A}_{S_{l'}}^k, \text{Dom}(\mathbf{A}_{S_r}^{<k} \cdot \mathbf{B}_{S_f})). \end{aligned}$$

Consider the sequence σ of expressions and registers which follow v in $\mathbf{U}_{f,v}^3$, and replace every occurrence of a register u with $\mathbf{U}_{l',u}^3$. View concatenation as the DReX operation combine, and define $f_3(S_{l'}, S_r, S_f)$ as the expression which results. Let:

$$f_3 = \sum\{f_3(S_{l'}, S_r, S_f) \mid S_{l'}, S_r \sim S, S_f \sqsubsetneq S\}.$$

The value appended to the end of v by $w_r w_f$ is then given by:

$$\begin{aligned} \text{post} &= \text{shift}(\text{Dom}(\mathbf{A}_S^{2k}) \cdot \sum_{S' \sim S} \text{Dom}(\mathbf{A}_{S'}^k)^*, f), \text{ where} \\ f &= f_2 \text{ else } f_3. \end{aligned} \tag{5.9.6}$$

Finally, we can write:

$$\begin{aligned} \mathbf{H} &= \sum\{\mathbf{A}_{S_1}^{<3k} \cdot \mathbf{B}_{S_2} \mid S_1 \cdot S_2 = S, S_1 \sim S, S_2 \sqsubsetneq S\}, \text{ so that} \\ \mathbf{B}_{S,v,m}^+ &= \mathbf{H}_{v,m} \text{ else combine}(\text{pre}, \text{main}, \text{post}). \end{aligned} \tag{5.9.7}$$

By construction, we have:

Proposition 5.20. $\mathbf{B}_S^+$ *summarizes all strings* $w \in [\![r^{(i)}(q_{i+1}, q_{i+1})^+]\!]$ *and with shape* S.

5.10 Completing the Proof: Constructing $\mathbf{R}_S^{(i+1)}(q, q')$

We just constructed $\mathbf{B}_S^+$, and by equations 5.8.1, 5.5.2, and 5.5.1, we have completed the construction of $\mathbf{R}_S^{(i+1)}(q, q')$. It can be verified that:

Lemma 5.21. *For all $i \in \{0, 1, 2, \ldots, n-1\}$, shape S and states $q, q' \in Q$, $\mathbf{R}_S^{(i+1)}(q, q')$ summarizes all paths matching $r^{(i+1)}(q, q')$ and whose shape is S.*

This completes the proof of part 1 of theorem 5.1.

Recap. In this chapter so far, we have established that consistent DReX expressions can express all regular string transformations. The principal difficulty introduced by SSTs is that a single input symbol may influence non-contiguous substrings in the output in complicated ways, such as in the function shuffle from example 2.6. The solution was to use the chained sum operation so that symbols of the input could be repeatedly scanned to produce different parts of the output.

The translation from SSTs to function expressions involved an outer induction where, in step i, we summarized all strings w from q to q' while only passing through intermediate states $\{q_1, q_2, \ldots, q_i\}$. We associated paths through the SST with their shapes, indicating the pattern of data flows. We then investigated the possible shapes of sub-paths of a given path, and proved that the notion of shape ordering $\sqsubseteq$ captured this idea. This allowed us to set up a nested inductive construction, where we summarized strings with the shape S, assuming that the summaries for all strings w' with shape $S' \sqsubsetneq S$ is known. The copylessness of SSTs is also essential in this construction, because it forces the space of shapes to be finite.

5.11 The Case of SSTTs

We will now consider part 2 of theorem 5.1, i.e. the expressive completeness of QREs for term transducers. Since the proof very similar to the construction for string transducers, we will only highlight the differences.

5.11.1 Shapes and expression vectors

Recall the coffee shop machine from figure 3.4a. In figure 5.8, we present the summary of the string "SMCCM" starting from the state $q_{\neg S}$. In the previous case, of string-to-string transducers, a shape was a copyless function $S : V \to V^*$.

In figure 5.8c, we summarize all paths of the form "SMC^{n}M": they all share the same shape $\{x \mapsto \{x, z\}, y \mapsto \varnothing, z \mapsto \varnothing\}$, and are summarized by the expression vector $\mathbf{A}$:

$$\mathbf{A}_x = \text{split}(S \cdot M \mapsto x + z, \text{iter}(C \mapsto 2, +), M \mapsto 0, +),$$
$$\mathbf{A}_y = S \cdot M \cdot C^* \cdot M \mapsto 0, \text{ and}$$
$$\mathbf{A}_z = S \cdot M \cdot C^* \cdot M \mapsto 0.$$

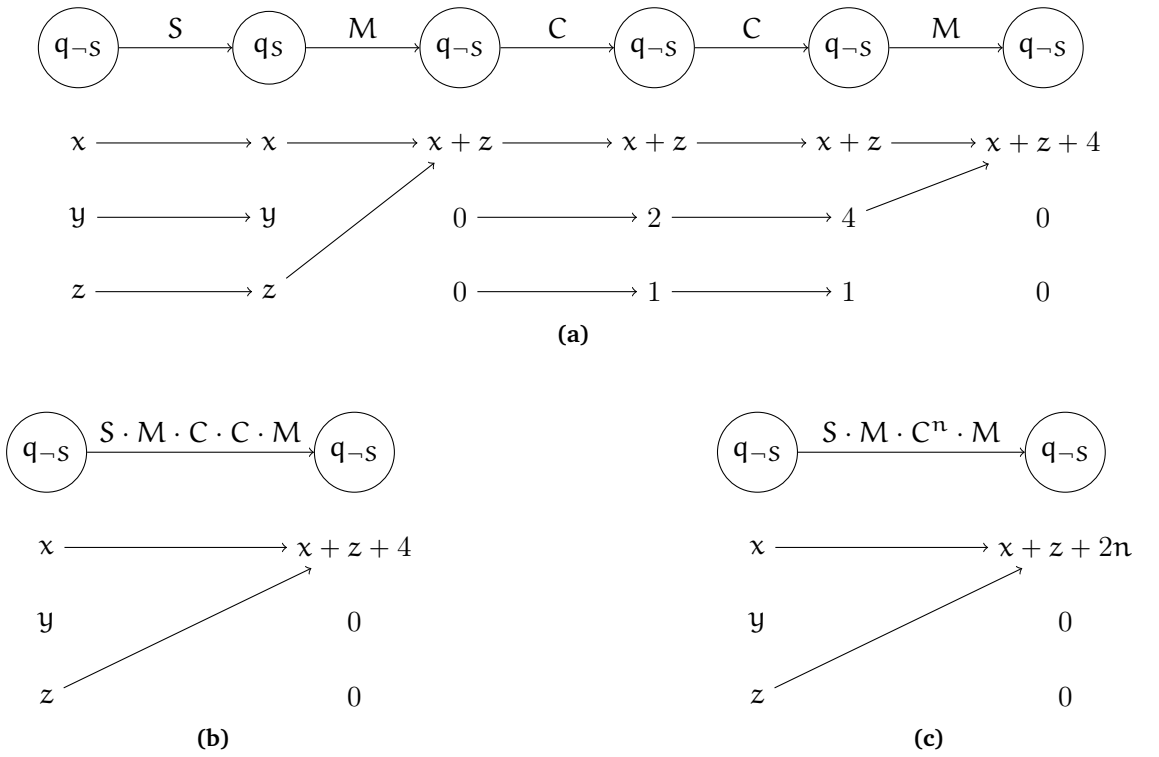


Figure 5.8: Summarizing paths through SSTTs. We are referring to the machine M_3 from figure 3.4a.

The string summary represents the final value of each register in terms of the initial register valuation. In the case of term transducers, a shape $S : V \to \text{UpdSchemes}$ is a function mapping each register v to an "update scheme" $S(v)$. The *update scheme* $S(v)$ is a set of registers, where each register $u \in S(v)$ is associated with a set of parameter substitutions (p, U_p), where p is a parameter and U_p is itself an update scheme, indicating that during the execution of the path σ, the parameter p of the initial value of v has been substituted for some update scheme U_p:

$$\begin{aligned} U \quad ::= \quad & \{v_1[p_{11} := U_{11}, p_{12} := U_{12}, \ldots, p_{1l} := U_{1l}], \\ & \quad v_2[p_{21} := U_{21}, p_{22} := U_{22}, \ldots], \ldots\} \end{aligned} \tag{5.11.1}$$

We will only be interested in single-use update schemes and single-use shapes: note that there are only a finite number of single-use shapes over a finite set of registers V.

Let S be a shape over the registers of an SSTT M. An expression vector $\mathbf{A}$ with shape S is a map from each register v and update scheme U occurring in $S(v)$ to a consistent well-typed, single-use QRE, $\mathbf{A}_{v,U}$.

Let $L \subseteq \Sigma^*$ be a set of paths starting from the state q with the same shape S. $\mathbf{A}$ summarizes L if: (*a*) for each $w \in L$, $[\![\mathbf{A}_v]\!](w) = \mu(q, a, v)$, and (*b*) for all v, $\text{Dom}(\mathbf{A}_v) = L$.

5.11.2 Ordering the registers and shapes

One subtle difficulty in the case of term transducers is that proposition 5.13 does not immediately carry over: since the registers are typed, it is not always feasible to respect a global ordering by renaming registers as we did in the previous construction.

We instead locally associate each state q with a total order over the registers $\preceq_q$. A total order $\preceq_q$ over a finite set of registers V permits the natural definition of "register rank": $\text{rank}_q : V \to \{1, 2, \ldots, k\}$ such that $\text{rank}_q(u) \leqslant \text{rank}_q(v)$ iff $u \preceq_q v$. A transition $q \to^a q'$ is upward-flowing if whenever $\mu(q, a, u)$ contains an occurrence of v, $\text{rank}_{q'}(u) \leqslant \text{rank}_q(v)$. It can be verified that if each transition of a path $\sigma = q \to^w q'$ is upward-flowing, then the whole path σ is itself upward-flowing. With this finer notion of the register ordering, an equivalent of proposition 5.13 can be proved:

Proposition 5.22. *For every SSTT* M*, there is an equivalent upward-flowing SSTT* M'*.*

Next, while our previous definitions of the ordering over shapes $\sqsubseteq$ and support-equality $\sim$ in section 5.7 were quite general, we were eventually only concerned with iterating loops at a single state q_{i+1}. In the present setting, we can therefore restrict our attention to a per-state ordering over shapes $\sqsubset_q$, $\sqsubseteq_q$ and a per-state idea of support-equality $\sim_q$. Apart from only ordering shapes of self-loops at a single state q, both definitions are otherwise identical to those in definition 5.14. We now have the following analogues of propositions 5.15, 5.16 and 5.17:

Proposition 5.23. *Let* $\sigma = q \to^w q$ *be a* looping *path through the SSTT* M *with shape* S*, and* σ' *be a sub-*loop *of* σ *with shape* S'*. Then* $S' \sqsubseteq_q S$*.*

Proposition 5.24. *Let there be* k *registers in the SSTT* M*. Choose* $k+1$ *strings* $w_1, w_2, \ldots, w_k, w' \in \Sigma^*$ *and a* loop $\sigma = q \to^{w_1} q \to^{w_2} q \to^{w_3} \cdots \to^{w_k} q \to^{w'} q$ *through the SST such that for each* $i \in \{1, 2, \ldots, k\}$*, the shape* S_i *of the sub-*loop w_i *is support-equal to the shape* S *of the entire loop* σ*. Then the prefix sub-*loop $q \to^{w_1 w_2 \cdots w_k} q$ *also has shape* S*.*

As before, a shape S is stable if whenever $u \to v$ occurs in S, then $v \to v$ also occurs in S.

Proposition 5.25. *Let there be* k *registers in the SSTT* M*. Let* $\sigma = q \to^{w_1} q \to^{w_2} q \to^{w_3} \cdots \to^{w_k} q$ *be a* loop *through the SSTT such that for each* $j \in \{1, 2, \ldots, k\}$*, the shape* S_j *of the sub-*loop w_j *is support-equal to the shape* S *of the entire* loop σ*. Then the shape* S *is stable.*

5.11.3 Computing $\mathbf{R}_S^{(i+1)}(q, q')$

The construction of $\mathbf{R}_S^{(i+1)}(q, q')$ for term transducers is similar to that for SSTs:

$$\begin{aligned}
\mathbf{R}_S^{(i+1)}(q, q') &= \mathbf{R}_S^{(i)}(q, q') \text{ else } \mathbf{C}_S(q, q'), \text{ where} \\
\mathbf{C}_S(q, q') &= \sum\{\mathbf{R}_{S_1}^{(i)}(q, q_{i+1}) \cdot \mathbf{B}_{S_2} \cdot \mathbf{R}_{S_3}^{(i)}(q_{i+1}, q') \mid S_1 \cdot S_2 \cdot S_3 = S\}, \\
\mathbf{B}_S &= \mathbf{B}_S^\epsilon \text{ else } \mathbf{B}_S^+, \text{ and} \\
\mathbf{B}_{S,v}^\epsilon &= \begin{cases} \epsilon \mapsto v & \text{if } S \text{ is the identity shape, and} \\ \text{bot} & \text{otherwise.} \end{cases}
\end{aligned}$$

As before, $\mathbf{B}_S^\epsilon$ summarizes the empty path $q_{i+1} \to^\epsilon q_{i+1}$ with shape S, and $\mathbf{B}_S^+$ summarizes all non-empty paths $q_{i+1} \to^w q_{i+1}$ with shape S. We now focus on the construction of $\mathbf{B}_S^+$. We proceed by induction on the pre-order $\sqsubseteq_{q+1}$. We already have the following expressions:

1. $\mathbf{R}_{S'}^{(i)}(q, q')$ for each shape S' and each pair of states q, q', and
2. $\mathbf{B}_{S'}$ for all shapes $S' \sqsubsetneq_q S$.

We construct the S-decomposition of self-loops $q_{i+1} \to^w q_{i+1}$ exactly as before, and consider k-segments. The construction of $\mathbf{B}_S^+$ for non-stable shapes is identical to that in equation 5.9.1. For stable shapes, we recall the observation that for every register u such that $u \to v$ along a k-segment, u must have been reset during the previous k-segment, and so the value flowing into v while processing $w_{s,m}$ is entirely determined by $w_{s,m-1} w_{s,m}$. We can use multi-expression iteration to capture this behavior: we include an expression e_u which computes the value of u while processing each k-segment, and the expression e_v which computes v incorporates the value of e_u. This argument can be formalized to obtain $\mathbf{B}_S^+$ and hence $\mathbf{R}_{i+1}^{(S)}(q, q')$ for stable shapes S.

5.12 Notes

In the original construction [19], instead of just defining upward-flowing shapes, we had a more powerful notion of *normalized* shapes. Approximately speaking, a normalized shape was

both upward-flowing and stable. We had claimed that the composition of normalized shapes is also normalized: as pointed out by Dana Fisman, this statement was false. Reworking the proof with just upward-flowing shapes requires a more involved statement in proposition 5.16, and more careful analysis in section 5.8.

In the original proof, we had also mistakenly claimed that the ordering relation between shapes was a partial order: $\sqsubseteq$ is not anti-symmetric (there are many unequal shapes which are support-equal to each other), and so it is only a partial order.

Arjun Radhakrishna recently pointed out that the S-decomposition of a path is very similar to the idea of factorization forests [83, 30]. Informally, in our setting, the theorem states that every homomorphism $\Sigma^* \to$ Shapes admits factorization forests of bounded height [73]. It appears that the deterministic version of the theorem is what is relevant to our case: it is an interesting direction of future work to flesh out this connection and attempt to simplify sections 5.6, 5.7 and 5.8.

Part II

Evaluation Algorithms

Chapter 6

A Fast Evaluation Algorithm for Consistent Expressions

In this part of the thesis, we will study the problem of evaluating function expressions. In chapter 4, we presented an algorithm to translate function expressions into equivalent transducers. Because transducers are an operational model, we have, in a sense, already solved the evaluation problem. However, the product construction is unavoidable for the expressions $\mathrm{op}(e, f)$ and $\mathrm{combine}(e, f)$, and therefore the complexity of that approach is exponential in the expression size $|e|$.

We will now present a fast evaluation algorithm for QREs and DReX expressions. The algorithm will make one left-to-right pass over the input stream $w = w_1 w_2 \cdots w_n$, processing each symbol w_i in $O(|e|^2)$ time and producing a result on the stream read so far, $w_1 w_2 \cdots w_i$.

The algorithms to evaluate DReX and QREs are both very similar, except for the case of the chained sum (lemma 6.12). For concreteness, we will only present the construction for the relevant QREs. The algorithm can be thought of as an implicit lazy construction of the previous SSTT, leading to asymptotic improvements in evaluation performance. Kostas Mamouras has made several insightful observations regarding this algorithm: as a result, the presentation here is much simpler than our earlier descriptions [14, 17], and is similar to the form in which it is presented in [20].

6.1 Formal Statement of Result

$s_l(e, w)$ and $s_t(e, w)$. Consider the DReX expression left-iter(true $\mapsto x$) which encodes the string reversal function. Given an input string w, the output

$$[\![\text{left-iter}(\text{true} \mapsto x)]\!](w) = \text{reverse}(w)$$

has the same length as the input, and cannot be emitted until w has been completely read. Because the algorithm will need to maintain the intermediate result in memory, there is an $\Omega(|w|)$ lower-bound on the memory requirements of a one-pass streaming evaluator.

Therefore, to accurately measure the memory usage of the evaluation algorithm, we introduce the value $s_l(e,w)$[1] to bound the length of the longest intermediate result produced during the computation of $[\![e]\!](w)$. This quantity is defined as:

$$s_l(e,w) = \max\{|[\![e']\!](w')| \mid e' \text{ sub-expression of } e, \text{ and } w' \text{ substring of } w\}. \quad (6.1.1)$$

The DReX evaluation algorithm will require $O(|e|s_l(e,w))$ space for computation. Separately, purely from an analysis of the running time of the algorithm, it will follow that the memory footprint is also bounded by $O(|e|^2|w|)$.

We also need to make similar considerations for the evaluation of QREs. Given a QRE e, and an input stream w, $s_t(e,w)$ is an upper-bound on the size of the largest intermediate term produced during the evaluation of $[\![e]\!](w)$, and is defined exactly as $s_l(e,w)$ in the case for DReX:

$$s_t(e,w) = \max\{|[\![e']\!](w')| \mid e' \text{ sub-expression of } e, \text{ and } w' \text{ substring of } w\}. \quad (6.1.2)$$

The QRE evaluation procedure will require $O(|e|s_t(e,w))$ space. It is easy to show that $s_t(e,w) = O(|e||w|)$, and therefore, the evaluation algorithm always requires no more than $O(|e|^2|w|)$ memory.

Having defined $s_l(e,w)$ and $s_t(e,w)$, we can state the main result of this chapter:

Theorem 6.1. *1. Let e be a consistent DReX expression, and $w = w_1w_2\cdots w_n$ be a sequence of symbols. There is an algorithm which makes one pass over w, and processes each symbol w_i in $O(|e|^2)$ time. During this processing, it will also output the value of $[\![e]\!](w_1w_2\cdots w_i)$, if it is defined. While reading the first i symbols, the algorithm will consume less than $O(|e|s_l(e,w_1w_2\cdots w_i))$ memory.*

2. Let e be a consistent, well-typed, single-use QRE, and $w = w_1w_2\cdots w_n$ be a sequence of input symbols. There is an algorithm which makes one pass over w, and processes each symbol w_i in $O(|e|^2)$ time. During this processing, it will also output the value of $[\![e]\!](w_1w_2\cdots w_i)$, if it is defined. While reading the first i symbols, the algorithm will consume less than $O(|e|s_t(e,w_1w_2\cdots w_i))$ memory.

From the running time of the algorithms, it also follows that the memory usage is never more than $O(|e|^2|w|)$.

A brief comment on the computational model. The input stream is read using a left-to-right read-only head, and the working memory is organized as a RAM. With the memory requirement set at $O(|e|^2|w|)$, one potential way for a one-pass evaluation algorithm to cheat is to copy the entire input string into memory, and then perform arbitrarily many passes over the representation of the string stored in memory. However, such pathological algorithms would fail to satisfy theorem 6.1: After reading each symbol w_i of the input stream, there is a hard $O(|e|^2)$ bound on the time within which the algorithm must both produce $[\![e]\!](w_1w_2\cdots w_i)$ and become ready to accept the next symbol of the input stream.

[1]The subscript "l" in $s_l(e,w)$ is intended to evoke the idea of "string length"

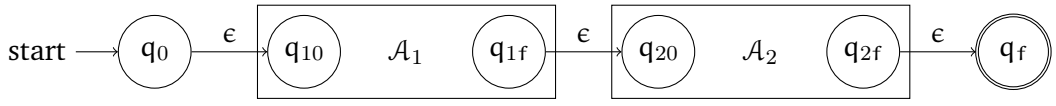


Figure 6.1: The traditional regular expression-to-NFA construction, $\mathcal{A}_{r_1 \cdot r_2}$, for the case of $r_1 \cdot r_2$. The states q_{10} and q_{1f} are the initial and accepting states of $\mathcal{A}_1$, and q_{20} and q_{2f} are the initial and accepting states of $\mathcal{A}_2$ respectively.

This time bound is independent of the number of symbols, i, of the input stream read so far, and insufficient to make even one full pass over any potential copies of the input stream stored in memory.

Also notice that the memory usage of the evaluation algorithm is actually output sensitive: if $s_l(e, w)$ (resp. $s_t(e, w)$) is $o(|e||w|)$ (emphasis on the little-oh), then there is insufficient memory to store a copy of w, and this cuts off any path to subvert the claims of this chapter.

Finally, we note the structure of the strings produced by the DReX evaluator. We encode strings as follows:

$$\text{str} ::= \gamma \mid \text{concat}(\text{str}_1, \text{str}_2), \tag{6.1.3}$$

where $\gamma \in \Gamma^*$ is a string encoded in the traditional list-of-characters representation. We name this representation a "lazy" string. In contrast to the traditional representation, lazy strings offer constant time concatenation, and this is an important assumption we will make while establishing the time complexity of the evaluation routine. Observe that lazy strings can be converted back into character arrays in linear time.

6.2 Motivation

Parsing regular expressions. We recall the traditional construction of NFAs from regular expressions, and specifically the case of concatenation, $r = r_1 \cdot r_2$, in figure 6.1. After constructing this NFA, the process to match an input string w against a regular expression r is like playing a board game: A token $\bullet$ (pronounced "bullet") is first inserted into the machine in state q_0. After reading each input character a, for each token-containing state q, a new token is placed in every state q' such that $q \to^a q'$, and the original token is taken away from q. Ultimately, w is accepted iff there is a token in some accepting state q_f.

The process we just described typically only returns a boolean value indicating whether the string matches the pattern under consideration. It can, however, be extended to return the *parse tree* of the string, and this more involved problem is conceptually closer to the problem of evaluating QREs.

Each token represents a potential (and still *incomplete*) parse tree of the entire string w, as shown in figure 6.2. The edges of the NFA can then be labelled with *actions* to update the incomplete parse trees corresponding to each token.

Recall that we inductively build accepting NFAs from regular expressions: Let r be a sub-expression of a larger regular expression r', and consider the NFA $\mathcal{A}$, corresponding to r, constructed as a component of the NFA $\mathcal{A}'$ which accepts r'. Let the entire input string be $w' = w_{\text{pre}} \cdot w \cdot w_{\text{post}}$, where $w \in [\![r]\!]$. Say also that processing the prefix w_{pre} results in a token $\bullet$ being dropped in the initial state $q_{0,\mathcal{A}}$ of the component machine $\mathcal{A}$:

$$q_{0,\mathcal{A}'} \rightarrow^{w_{\text{pre}}} q_{0,\mathcal{A}}.$$

Finally, say that processing w by $\mathcal{A}$ results in the input token $\bullet$ moving to some accepting state $q_{f,\mathcal{A}}$:

$$\bullet @ q_{0,\mathcal{A}} \rightarrow^{w} \bullet' @ q_{f,\mathcal{A}}.$$

Recall that the input token $\bullet$ contains the partial parse tree for the prefix w_{pre}, and that we resolved to annotate the transitions of the NFA with actions to update these partial parse trees. Here $\bullet'$ is the modified token reaching $q_{f,\mathcal{A}}$. Therefore, to compute parse trees in addition to the boolean accept / reject value, the idea is to include the parse tree t_w of w according to the sub-expression r as part of the output token $\bullet'$:

$$\bullet' = (t_w, \bullet).$$

When $\bullet'$ re-enters the outer machine $\mathcal{A}'$, we can perform appropriate actions to incorporate t_w into the larger parse tree under construction.

More concretely, a token (which represents an incomplete parse tree) is encoded as a tuple $(t_1, t_2, \ldots, t_k)$ of fully-specified parse trees for various sub-expressions of the complete regular expression. Consider, for example, the partial tree of figure 6.2b. This can be represented by the tuple $((*\mathtt{bb}), (*\mathtt{aaa}))$, indicating the maximal fully-specified subtrees of the tree under consideration.

In $\mathcal{A}_{r_1 \cdot r_2}$ from figure 6.1, we label the $q_0 \rightarrow q_{10}$ and $q_{1f} \rightarrow q_{20}$ edges with the empty action. The incoming token $\bullet$ is therefore passed unchanged to the state q_{10} of $\mathcal{A}_1$. On matching a prefix w_{pre} of the input, the token reaching state q_{1f} is of the form $(\tau_{\text{pre}}, \bullet)$, where τ_{pre} is a parse tree of w_{pre} for the sub-expression r_1. This token is then placed in the starting state q_{20} of $\mathcal{A}_2$. On reading a suffix w_{post} which matches r_2, the token reaching q_{2f} is of the form $(\tau_{\text{post}}, \tau_{\text{pre}}, \bullet)$, where τ_{post} is a parse tree of w_{post}. The entire string $w = w_{\text{pre}} w_{\text{post}}$ can then be described by the parse tree $\tau = \mathtt{concat}(\tau_{\text{pre}}, \tau_{\text{post}})$. It is then sufficient to modify the original automaton $\mathcal{A}_{r_1 \cdot r_2}$ as shown in figure 6.3.

Evaluating QREs. The main problem with this approach is the product construction required for $\text{op}(e, f)$, as repeated invocations of the product construction will yield a machine whose size is exponential in the given QRE. Given a consistent, single-use QRE e, we therefore instead construct an "evaluator" M which computes $[\![e]\!](w)$ by lazily traversing the state space. M is a stateful object with two actions:

1. $M(\text{start}, \bullet)$, analogous to the placement of a token in the start state of the NFA, and

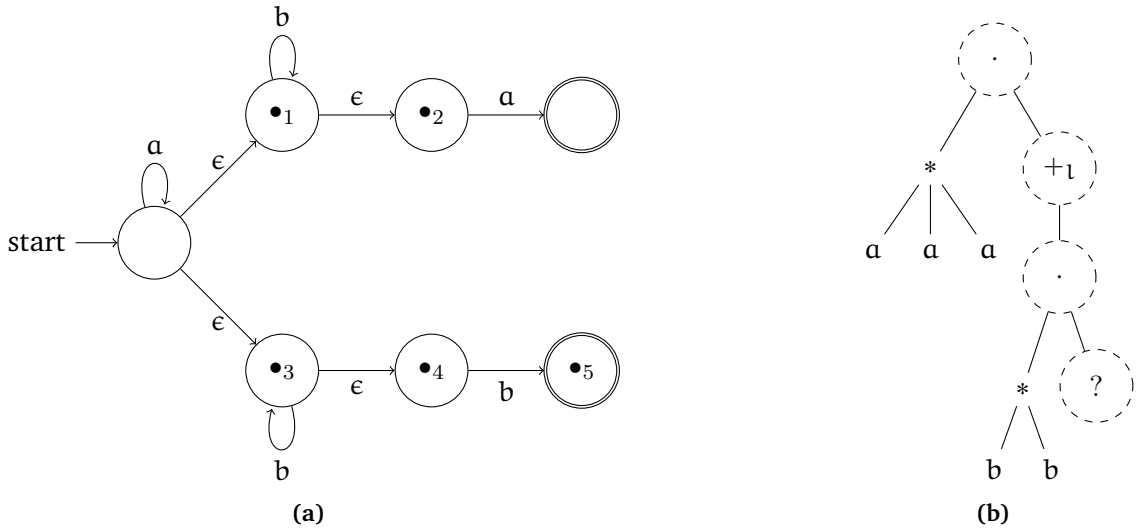


Figure 6.2: Visualizing incomplete parse trees. The regular expression of interest is $r = r_1 \cdot r_2$, where $r_1 = a^*$ and $r_2 = b^*a + b^*b$. The string read so far is $w = a^3b^2$. We show the incomplete parse tree corresponding to the token $\bullet_2$ in figure 6.2b, where incomplete nodes are circled.

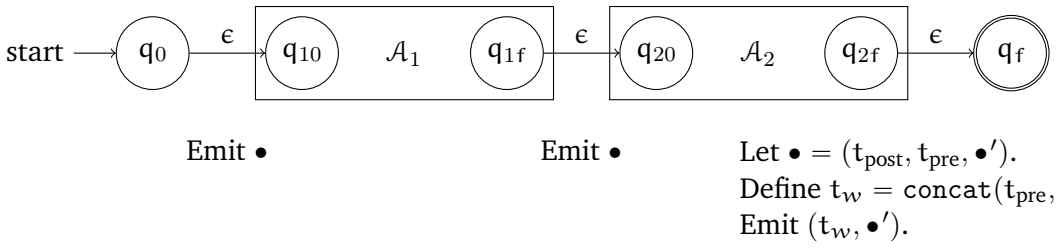


Figure 6.3: Extending the NFA $\mathcal{A}_{r_1 \cdot r_2}$ of figure 6.1 with actions to construct parse trees. On all three marked transitions, $\bullet$ refers to the incoming token. The $q_0 \to q_{10}$ and $q_{1f} \to q_{20}$ transitions simply emit the incoming token unchanged. The final $q_{2f} \to q_f$ transition concatenates the parse trees corresponding to r_1 and r_2 to obtain the parse tree for $r_1 \cdot r_2$.

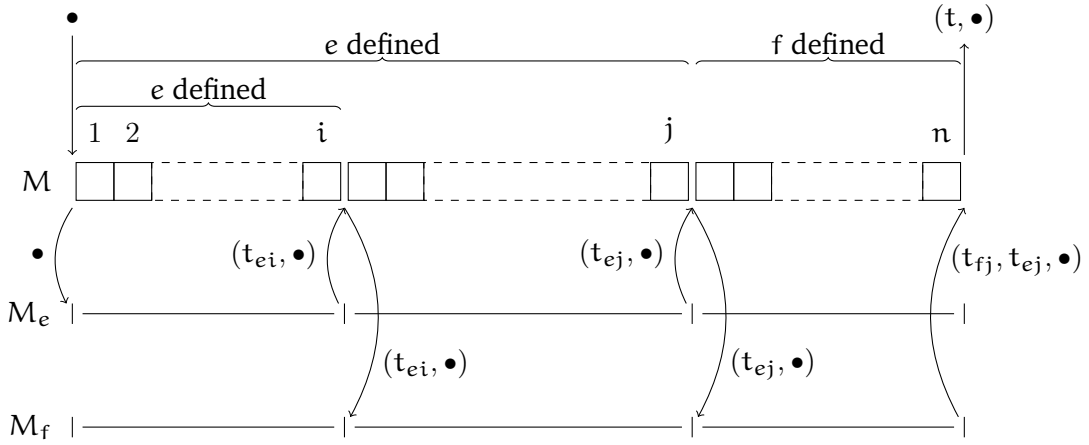


Figure 6.4: Example run of the evaluator M for split($e \to^{p} f$) over a stream w. The sub-expression e is defined for two prefixes $w_1, w_2, \ldots, w_i$ and $w_1, w_2, \ldots, w_j$ of the input stream, and f is only defined for $w_{j+1}, \ldots, w_n$. The token at the top-left corner initiates the causal sequence of starts and results. At the end of the string, when the parent evaluator M receives the result $(t_{fj}, t_{ej}, \bullet)$ from M_f, it computes $t = t_f[p := t_e]$, and itself produces the result $(t, \bullet)$.

2. $M(a)$, for each $a \in \Sigma$, corresponding to the machine reading the symbol a from the input stream.

In response to each action, M may optionally produce a result of the form $(t_e, \bullet)$, where $\bullet$ is some token introduced in the past, and t_e is the result of e on all input characters received after receiving $\bullet$.

We illustrate the input-output behavior of the evaluator M for split($e \to^{p} f$) in figure 6.4. M maintains two sub-evaluators, M_e and M_f, corresponding to each sub-expression. M is given the input sequence $(\text{start}, \bullet), w_1, w_2, \ldots, w_n$. The first sub-expression e is defined on two prefixes of the input:

$$[\![e]\!](w_1 w_2 \cdots w_i) = t_{ei}, \text{ and}$$
$$[\![e]\!](w_1 w_2 \cdots w_j) = t_{ej}.$$

The second sub-expression f is defined on the suffix $w_{j+1} w_{j+2} \cdots w_n$:

$$[\![f]\!](w_{j+1} w_{j+2} \cdots w_n) = t_{fj}.$$

Similar to the NFA of figure 6.1, M_e is given the input sequence: $(\text{start}, \bullet), w_1, w_2, \ldots, w_n$, corresponding to a token being injected in the start state at the beginning of the string. M_e responds by declaring $(t_{ei}, \bullet)$ and $(t_{ej}, \bullet)$ after reading w_i and w_j respectively. In turn, M_f is given the input sequence $w_1, w_2, \ldots, w_i, (\text{start}, (t_{ei}, \bullet)), w_{i+1}, w_{i+2}, \ldots, w_j, (\text{start}, (t_{ej}, \bullet))$,

Algorithm 6.1 $M_{\text{split}(e \to^{p} f)}$.

1. Initialize sub-evaluators M_e and M_f for e and f respectively.
2. On receiving $(\text{start}, \bullet)$, invoke $r_e \leftarrow M_e(\text{start}, \bullet)$, and let $r_f = \text{none}$.
3. On receiving $a \in \Sigma$, invoke $r_e \leftarrow M_e(a)$, and $r_f \leftarrow M_f(a)$.
4. If $r_e \neq \text{none}$, invoke $r'_f \leftarrow M_f(\text{start}, r_e)$. Otherwise, let $r'_f = \text{none}$.
5. Assert $r_f = \text{none} \vee r'_f = \text{none}$. At least one of the calls to M_f must have returned none.
6. If $r_f = (t_f, t_e, \bullet)$ (for some t_e, t_f, and $\bullet$) and $r'_f = \text{none}$, define $t = t_f[p \coloneqq t_e]$. Output $(t, \bullet)$.
7. Otherwise, if $r_f = \text{none}$ and $r'_f = (t_f, t_e, \bullet)$ (for some t_e, t_f, $\bullet$), define $t = t_f[p \coloneqq t_e]$. Output $(t, \bullet)$.
8. Otherwise, if $r_f = r'_f = \text{none}$, output none.

$w_{j+1}, w_{j+2}, \ldots, w_n$. After reading w_n, M_f produces the output $(t_{fj}, t_{ej}, \bullet)$. The parent evaluator M processes this result and itself produces the output $(t, \bullet)$, where $t = t_{fj}[p \coloneqq t_{ej}]$. Algorithm 6.1 describes the final construction of M. We will prove its correctness in section 6.4.

Bounding memory usage and collision-freedom. So far, we have ignored the memory requirements of the constructed evaluators. In the regular expression matching problem, tokens are values without structure. If two tokens reach the same state, either of them can be silently dropped without affecting algorithmic correctness (see figure 6.5). There are therefore at most $O(|r|)$ tokens, independent of the string length.

When we generalize the problem to construct parse trees, tokens have internal structure: each token is a tuple of parse trees, with total size at most $O(|r||w|)$. If we are only interested in computing *some* parse tree, rather than finding *all* parse trees, we can still drop co-located tokens. The total memory usage of parse tree computation is therefore $O(|r|^2|w|)$.

We present the evaluator $M_{\text{op}(e,f)}$ for $\text{op}(e, f)$ in algorithm 6.2. The assertion on line 4 follows from the consistency requirements: e and f are mandated to have equal domains. Observe that the assertion on line 5a is crucial to correctness: otherwise, if M_e and M_f report results corresponding to different threads, $\bullet \neq \bullet'$, there is no meaningful way to combine them. However, the co-located-token-dropping optimization weakens the guarantee to generate some, rather than all, parse trees, and is therefore erroneous.

Consider an input string w which results in the presence of co-located tokens, and a suffix w' such that ww' is accepted by the machine. It follows that there exists a string ww' which admits two different parse trees. Recall that consistency also requires the underlying regular expressions to be unambiguous. For the top-level evaluator, it follows that there are *never* any co-located tokens. During our inductive construction, we will require and ensure that the

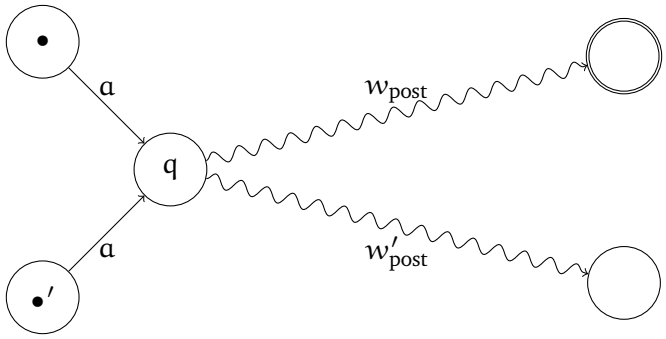


Figure 6.5: Dropping co-located tokens while pattern matching. Once both tokens $\bullet$ and $\bullet'$ reach the same state q, they have equal destinies for all suffixes w_{post}, w'_{post}. Either token can be safely dropped without sacrificing correctness. The number of live tokens is therefore bounded by $|r|$.

Algorithm 6.2 $M_{\text{op}(e,f)}$.

1. Create sub-evaluators M_e, M_f for e, f respectively.
2. On receiving $(\text{start}, \bullet)$, invoke $r_e \leftarrow M_e(\text{start}, \bullet)$, and $r_f \leftarrow M_f(\text{start}, \bullet)$.
3. On receiving $a \in \Sigma$, invoke $r_e \leftarrow M_e(a)$ and $r_f \leftarrow M_f(a)$.
4. Assert $(r_e \neq \text{none} \wedge r_f \neq \text{none}) \vee (r_e = \text{none} \wedge r_f = \text{none})$. Either both sub-evaluators simultaneously report results, or neither does.
5. If $r_e = (t_e, \bullet)$ and $r_f = (t_f, \bullet')$ (for some t_e, t_f, $\bullet$, and $\bullet'$):
   (a) Assert $\bullet = \bullet'$.
   (b) Define $t = \text{op}(t_e, t_f)$ and return $(t, \bullet)$.
6. Otherwise, output none.

input to each sub-evaluator be "collision-free", i.e. that the process of inserting tokens into the start state itself does not cause token co-location. The assertion on line 5a of $M_{\mathrm{op}(e,f)}$ follows from this consideration, as token-dropping is unnecessary for collision-free inputs.

In section 6.4, we will construct M by induction on e. The most important part of the correctness proof will be showing that if a parent evaluator receives a collision-free input sequence, then the children will also receive collision-free input sequences. We will formally define collision-freedom in section 6.3.

6.3 The Formal Specification of a Function Evaluator

Inputs, outputs, and threads. A TOKEN $\bullet$ is a tuple of single-use terms:

$$\bullet ::= \mathrm{nil} \mid (t, \bullet') \tag{6.3.1}$$

We will occassionally treat a token as a stack, with the constructor $(t, \bullet')$ serving as the push operation and tail extraction serving as the pop operation. Tok is the space of all tokens. Let e be a consistent, single-use QRE with input alphabet Σ and which outputs terms of type T. Define $\Theta = (\{\mathrm{start}\} \times \mathrm{Tok}) \cup \Sigma$, and $\Omega = (\mathrm{Terms}(T) \times \mathrm{Tok}) \cup \{\mathrm{none}\}$.

If the i-th symbol of $\gamma \in \Theta^*$ is a start, then we write $\bullet_i$ for the token payload at this location. We denote the standard string projection operation is given by $\pi_\Sigma(\gamma)$:

$$\pi_\Sigma(\epsilon) = \epsilon, \text{ and} \tag{6.3.2}$$

$$\pi_\Sigma(a \cdot \gamma) = \begin{cases} a \cdot \pi_\Sigma(\gamma) & \text{if } a \in \Sigma, \text{ and} \\ \pi_\Sigma(\gamma) & \text{otherwise.} \end{cases} \tag{6.3.3}$$

If the i-th symbol of $\gamma \in \Theta^*$ is a start, i.e. $\gamma = \gamma_{\mathrm{pre}}, (\mathrm{start}, \bullet_i), \gamma_{\mathrm{post}}$, where $|\gamma_{\mathrm{pre}}| = i - 1$, then we call $\pi_\Sigma(\gamma_{\mathrm{post}})$ the THREAD beginning at index i, and denote it as th_i.

Example 6.2. Consider $\gamma = a, b, b, (\mathrm{start}, \bullet_4), b, (\mathrm{start}, \bullet_6), a$. There are two threads, beginning at indices 4 and 6 respectively.

$$\mathrm{th}_4 = \pi_\Sigma(b, (\mathrm{start}, \bullet_6), a) = ba, \text{ and}$$
$$\mathrm{th}_6 = \pi_\Sigma(a) = a. \qquad \triangle$$

Collision freedom. An input sequence $\gamma \in \Theta^*$ is COLLISION-FREE with respect to a QRE e if for each prefix γ' of γ, for all distinct threads th'_i, th'_j, of γ', and for all potential suffixes $w_{\mathrm{post}} \in \Sigma^*$, either $[\![e]\!](\mathrm{th}'_i \cdot w_{\mathrm{post}})$ or $[\![e]\!](\mathrm{th}'_j \cdot w_{\mathrm{post}})$ is undefined.

Example 6.3. Let $e = \mathrm{split}(a^+ \mapsto a, b^+ \mapsto b)$. Informally, e maps strings $w \in a^+b^+$ to the constant output string ab. Consider the input sequence:

$$\gamma_1 = (\mathrm{start}, \bullet_1), a, a, (\mathrm{start}, \bullet_4).$$

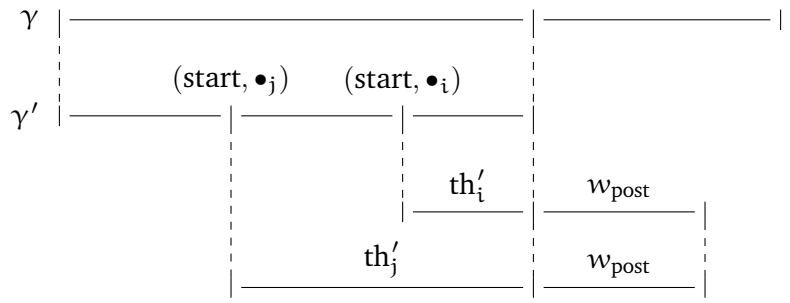


Figure 6.6: Collision-free evaluator inputs. The input stream γ is collision-free with respect to e if for all prefixes γ', for all distinct threads i, j of γ', and for all suffixes w_{post}, either $[\![e]\!](\text{th}'_i \cdot w_{\text{post}})$ or $[\![e]\!](\text{th}'_j \cdot w_{\text{post}})$ is undefined.

For the suffix $w_{\text{post}} = ab$, both $[\![e]\!](\text{th}_1 \cdot w_{\text{post}}) = [\![e]\!](aa \cdot ab)$ and $[\![e]\!](\text{th}_4 \cdot w_{\text{post}}) = [\![e]\!](\epsilon \cdot ab)$ are defined. The sequence γ_1 is therefore not collision-free. Now consider the input sequence:

$$\gamma_2 = (\text{start}, \bullet_1), a, a, b, (\text{start}, \bullet_5).$$

If for some suffix w_{post}, $[\![e]\!](\text{th}_5 \cdot w_{\text{post}}) = [\![e]\!](\epsilon \cdot w_{\text{post}})$ is defined, then w_{post} has to begin with an occurrence of the character a, and consequently, $[\![e]\!](\text{th}_1 \cdot w_{\text{post}}) = [\![e]\!](aab \cdot w_{\text{post}})$ is undefined. It follows that γ_2 is collision-free. △

In figure 6.6, we graphically present the definition of collision-freedom. The following elementary observations help to clarify this idea.

Proposition 6.4. *1. If $\text{Dom}(e) = \emptyset$, then every input sequence $\gamma \in \Theta^*$ is collision-free with respect to e.*

2. If e and e' are consistent QREs, $\text{Dom}(e) \subseteq \text{Dom}(e')$, and γ is collision-free with respect to e', then γ is also collision-free with respect to e.

3. If γ is collision-free with respect to a consistent QRE e, and $w \in \Sigma^*$ is a sequence of input symbols, then $\gamma \cdot w$ is also collision-free with respect to e.

4. There exist consistent QREs e and e', and an input sequence γ such that $\text{Dom}(e) \subseteq \text{Dom}(e')$, and γ is collision-free with respect to e, but not collision-free with respect to e'.

Proof of part 4. The witnesses e, e' and γ are as follows:

$$e = \text{split}(a^+ \mapsto a, b^+ \mapsto b, \cdot),$$
$$e' = \text{iter}(a \mapsto b \text{ else } b \mapsto a), \text{ and}$$
$$\gamma = (\text{start}, \bullet_1), a, a, b, (\text{start}, \bullet_5).$$

Both e, e' are consistent, and $\text{Dom}(e) \subseteq \text{Dom}(e')$. In example 6.3, we saw that γ is collision-free with respect to e, and it can be shown that γ is not collision-free with respect to e'. □

Algorithm 6.3 $M_{\varphi \mapsto \lambda}$.

1. Maintain a variable $q \in \text{Tok} \cup \{\text{none}\}$. Initialize $q := \text{none}$.
2. On receiving $(\text{start}, \bullet)$:
   (a) Assert $q = \text{none}$.
   (b) Update $q := \bullet$, and return none.
3. On receiving $a \in \Sigma$:
   (a) If $q \neq \text{none}$ and $\varphi(a) = \text{true}$,
      i. define $\bullet = (\lambda(a), q)$,
      ii. update $q := \text{none}$, and
      iii. return $\bullet$.
   (b) Otherwise, update $q := \text{none}$, and return none.

Function evaluators. A function evaluator is a stateful machine M which accepts a sequence of input signals $\gamma \in \Theta^*$ and after reading each signal, produces an output from Ω (Recall from the beginning of this section that $\Theta = (\{\text{start}\} \times \text{Tok}) \cup \Sigma$, and $\Omega = (\text{Terms}(T) \times \text{Tok}) \cup \{\text{none}\}$). An evaluator M COMPUTES a consistent QRE e if after reading each collision-free input sequence γ, the machine outputs $(t, \bullet)$ iff there is a thread i such that $[\![e]\!](\text{th}_i) = t$ and $\bullet_i = \bullet$.

6.4 Inductive Evaluator Construction

In this section, we will inductively construct function evaluators for QREs and DReX expressions. In the previous section, we declared that an evaluator computes an expression if after reading each collision-free input sequence, it produces the appropriate result. In our correctness proofs, we therefore assume that the incoming sequence is collision free. However, before assuming that the values returned by child evaluators are meaningful, we first need to establish that the input sequence supplied to the child by the parent is also collision-free.

Basic evaluators. We define the evaluators $M_{\varphi \mapsto \lambda}$, $M_{\epsilon \mapsto t}$, and M_{bot}, for the basic QREs $\varphi \mapsto \lambda$, $\epsilon \mapsto t$, and bot in algorithms 6.3, 6.4, and 6.5 respectively.

Lemma 6.5. $M_{\varphi \mapsto \lambda}$ *computes* $\varphi \mapsto \lambda$.

Proof. In any collision-free input sequence γ with respect to $\varphi \mapsto \lambda$, there cannot be consecutive start signals. Since every input character $a \in \Sigma$ clears the value of register q, the assertion on line 2a always holds.

Algorithm 6.4 $M_{e \mapsto t}$.

1. On receiving $(\text{start}, \bullet)$, return $(t, \bullet)$.
2. On receiving $a \in \Sigma$, return none.

Algorithm 6.5 M_{bot}.

1. On receiving $(\text{start}, \bullet)$, return none.
2. On receiving $a \in \Sigma$, return none.

Say there is a thread i such that $[\![\varphi \mapsto \lambda]\!](th_i) = t_i$ is defined. From the semantics of the combinator, it follows that $|th_i| = 1$, $\varphi(th_i) = \text{true}$, and a start signal $(\text{start}, \bullet_i)$ immediately preceded the last input character. It follows that the machine responds to the input sequence with the expected result $(\lambda(\gamma_{i+1}), \bullet_i)$.

Conversely, say the machine responds with a result $(t, \bullet)$. It must have been the case that the last symbol $\gamma_n \in \Sigma$ and that before reading this symbol, $q \neq \text{none}$. Since q is reset to none after reading each character $a \in \Sigma$, it follows that γ_{n-1} must have been a start signal $(\text{start}, \bullet_{n-1})$. It also follows that the produced output $t = \lambda(\gamma_n)$ and that $\bullet = \bullet_{n-1}$. □

Lemma 6.6. $M_{e \mapsto t}$ *computes* $e \mapsto t$.

Proof. First, any collision-free input sequence γ to $e \mapsto t$ cannot have simultaneous start signals. There are two cases of interest: (*a*) if the last symbol γ_n of the input sequence γ was a start signal $(\text{start}, \bullet_n)$, then $M_{e \mapsto t}$ correctly responds with $(t, \bullet_n)$, and (*b*) otherwise, if the last symbol $\gamma = a$, where $a \in \Sigma$, then for all threads i of γ, $|th_i| \geqslant 1$, and $[\![e \mapsto t]\!](th_i)$ is undefined. $M_{e \mapsto t}$ again produces the correct expected response of none. □

Lemma 6.7. M_{bot} *computes bot.*

Proof. In this degenerate case, every input sequence is collision-free. The expression bot is undefined for each thread th_i of γ. M_{bot} always produces the response "none", and correctness follows. □

Choice, function combination, and substitution. We presented the evaluator for $\text{op}(e, f)$ earlier in this chapter in algorithm 6.2: we will prove its correctness in lemma 6.9. The evaluators for function combination $\text{op}(e, f)$ and substitution $e[p := f]$ are very similar. To obtain the evaluator for $e[p := f]$, replace the definition $t = \text{op}(t_e, t_f)$ in line 5b of algorithm 6.2 with $t = t_e[p := t_f]$. The full case of $\text{op}(e_1, e_2, \ldots, e_k)$ can be obtained by a straightforward generalization of the presented algorithm. We begin by presenting the evaluator for $M_{e \text{elsef}}$ in algorithm 6.6.

Algorithm 6.6 M_{eelsef}.

1. Create sub-evaluators M_e and M_f for e and f respectively.
2. On receiving $(\text{start}, \bullet)$, invoke $r_e \leftarrow M_e(\text{start}, \bullet)$, and $r_f \leftarrow M_f(\text{start}, \bullet)$.
3. On receiving $a \in \Sigma$, invoke $r_e \leftarrow M_e(a)$, and $r_f \leftarrow M_f(a)$.
4. Assert $r_e = \text{none} \vee r_f = \text{none}$. M_e and M_f do not simultaneously report results.
5. If $r_e \neq \text{none}$, return r_e. Otherwise, return r_f.

Lemma 6.8. *If* M_e *computes* e *and* M_f *computes* f*, then* M_{eelsef} *computes* e *else* f.

Proof. The input sequences γ_e, γ_f to both M_e and M_f are equal to the original input sequence γ: $\gamma_e = \gamma_f = \gamma$. For consistent QREs, $\text{Dom}(e), \text{Dom}(f) \subseteq \text{Dom}(e \text{ else } f)$. From part 2 of proposition 6.4, it follows that if the input to the parent evaluator is collision-free, then the inputs to both sub-evaluators are also collision-free.

Say there is some thread i, such that $[\![e \text{ else } f]\!](\text{th}_i) = t_i$. From the semantics we know that either $[\![e]\!](\text{th}_i) = t_i$, or $[\![f]\!](\text{th}_i) = t_i$. In the first case, by the induction hypothesis, it follows that M_e returns the expected result $(t_i, \bullet_i)$, and that M_{eelsef} also produces the expected result at line 5. The second case is similar. We have shown that whenever $[\![e \text{ else } f]\!](\text{th}_i) = t_i$, for some thread th_i, the evaluator M_{eelsef} also produces the response $(t_i, \bullet_i)$.

Conversely, say the parent evaluator produces a result $(t, \bullet)$ after reading $\gamma \in \Theta^*$. Again, by line 5, this result was causally preceded by a result from either M_e or M_f. Consider the first case, i.e. that $(t, \bullet)$ was produced by M_e. By the induction hypothesis, we can deduce the existence of a thread i, such that $t = [\![e]\!](\text{th}_i)$ and $\bullet = \bullet_i$. For the same thread th_i on the parent evaluator, we know that $t = [\![e \text{ else } f]\!](\text{th}_i)$. The second case (result produced by M_f) is similar. We have therefore shown that whenever M_{eelsef} produces a result $(t_i, \bullet_i)$, it is indeed the case that $[\![e \text{ else } f]\!](\text{th}_i) = t_i$. Correctness follows. □

Lemma 6.9. *If* M_e *computes* e *and* M_f *computes* f*, then* $M_{op(e,f)}$ *computes* $op(e, f)$.

Proof. As in algorithm 6.6, the inputs γ_e, γ_f to M_e and M_f are equal to γ, the input sequence given to the parent evaluator. The consistent expressions e, $\text{op}(e, f)$ are a limiting case, with $\text{Dom}(e) = \text{Dom}(\text{op}(e, f))$, of part 2 of proposition 6.4. If γ is collision-free with respect to $\text{op}(e, f)$, we can now say that γ_e is also collision-free with respect to e, and similarly that γ_f is collision-free with respect to f. We can now calculate:

$$
\begin{aligned}
& \exists \text{ thread } i, \text{ such that } [\![\text{op}(e, f)]\!](\text{th}_i) \text{ is defined} \\
\iff & \exists \text{ thread } i, \text{ such that both } [\![e]\!](\text{th}_i) \text{ is defined and } [\![f]\!](\text{th}_i) \text{ is defined} \\
\iff & M_e(\gamma_e) \text{ returns } (t_e, \bullet_i) \text{ and } M_f(\gamma_f) \text{ returns } (t_f, \bullet_i) \\
\iff & M(\gamma) \text{ returns } (\text{op}(t_e, t_f), \bullet_i).
\end{aligned}
$$

M therefore returns a result for exactly those input streams γ, and those thread indices i such that $[\![\mathrm{op}(e, f)]\!](\mathrm{th}_i)$ is defined. It is straightforward to show that this result is indeed the expected result. □

Function concatenation. We have already seen the construction of $M_{\mathrm{split}(e \to^p f)}$ in algorithm 6.1. $M_{\mathrm{split}(e \leftarrow^q f)}$ is symmetric: the definition $t = t_f[p := t_e]$ on lines 6 and 7 is replaced by $t = t_e[q := t_f]$. We will now prove its correctness.

Lemma 6.10. *If M_e computes e and M_f computes f, then $M_{\mathit{split}(e \to^p f)}$ computes $\mathit{split}(e \to^p f)$.*

Proof. Observe that $\gamma_e = \gamma$, where γ is the input sequence fed to the parent evaluator, and γ_e is the input to M_e. We first show that γ_e is collision-free with respect to e. Assume otherwise, so there are distinct threads i, j, and some suffix w_{post}, such that $[\![e]\!](\mathrm{th}_i w_{\mathrm{post}})$ and $[\![e]\!](\mathrm{th}_j w_{\mathrm{post}})$ are both defined.

From the consistency rules, we know that $\mathrm{Dom}(f) \neq \varnothing$. Say $w_f \in \mathrm{Dom}(f)$. Both threads i and j of the parent evaluator are now defined for the suffix $w_{\mathrm{post}} w_f$:

$$[\![\mathrm{split}(e \to^p f)]\!](\mathrm{th}_i w_{\mathrm{post}} \cdot w_f), \text{ and}$$
$$[\![\mathrm{split}(e \to^p f)]\!](\mathrm{th}_j w_{\mathrm{post}} \cdot w_f).$$

This contradicts the assumption that γ was collision-free with respect to $\mathrm{split}(e \to^p f)$.

We will now show that the input sequence to M_f, γ_f is also collision-free. Assume otherwise, so there exist distinct threads i, j, and an adversarial suffix w_{post} such that both $[\![f]\!](\mathrm{th}_{fi} w_{\mathrm{post}})$ and $[\![f]\!](\mathrm{th}_{fj} w_{\mathrm{post}})$ are defined. Every start signal to M_f, on line 4, is caused by a result produced by M_e. We have already established the collision-freedom of γ_e, and we can therefore claim that this result was in turn caused by a start signal to the parent evaluator. For each thread i, j of γ_f, let i' and j' be the corresponding threads entering the parent evaluator. Now observe that both

$$[\![\mathrm{split}(e \to^p f)]\!](\mathrm{th}_{i'} \cdot w_{\mathrm{post}}), \text{ and}$$
$$[\![\mathrm{split}(e \to^p f)]\!](\mathrm{th}_{j'} \cdot w_{\mathrm{post}})$$

are defined, again contradicting the assumption that the input to the parent evaluator, γ, was collision-free with respect to $\mathrm{split}(e \to^p f)$.

Now that we have proved the collision-freedom of the inputs provided to the children, it is easy to show that the parent evaluator correctly computes $\mathrm{split}(e \to^p f)$. □

Function iteration. In algorithm 6.7, we present an evaluator for $\mathrm{iter}(e_1 \to p_1, e_2 \to p_2)$. An evaluator for the full combinator, $\mathrm{iter}(e_1 \to p_1, e_2 \to p_2, \ldots, e_k \to p_k)$, is similar, and left-iteration $\mathrm{iter}(p_1 \leftarrow e_1, p_2 \leftarrow e_2, \ldots, p_k \leftarrow e_k)$ is symmetric. The key idea is to maintain the previous results from e_1 and e_2 at the top of the token stack fed to the child evaluator M_1. Every time M_1 reports a result (and by the consistency rules, M_2 necessarily reports simultaneously), the top of the token stack is updated appropriately, and the evaluators are restarted.

Algorithm 6.7 $M_{\text{iter}(e_1 \to p_1, e_2 \to p_2)}$.

1. Maintain sub-evaluators M_1 and M_2 for e_1 and e_2 respectively.
2. On receiving $(\text{start}, \bullet)$:
   (a) Define $\bullet' = (p_1, p_2, \bullet)$. Invoke $r_1 \leftarrow M_1(\text{start}, \bullet')$ and $r_2 \leftarrow M_2(\text{start}, \bullet)$.
   (b) Assert $r_1 = r_2 = \text{none}$.
   (c) Return $(p_1, \bullet)$.
3. On receiving $a \in \Sigma$:
   (a) Invoke $r_1 \leftarrow M_1(a)$ and $r_2 \leftarrow M_2(a)$.
   (b) Assert $(r_1 \neq \text{none} \wedge r_2 \neq \text{none}) \vee (r_1 = \text{none} \wedge r_2 = \text{none})$.
   (c) If $r_1 = (t_1, t_{1p}, t_{2p}, \bullet)$, and $r_2 = (t_2, \bullet')$ (for some terms t_1, t_2, t_{1p}, t_{2p}, and tokens $\bullet$, $\bullet'$),
      i. Assert $\bullet = \bullet'$.
      ii. Define $t_{1n} = t_1[p_1 := t_{1p}, p_2 := t_{2p}]$.
      iii. Invoke $r_{1n} \leftarrow M_1(\text{start}, (t_{1n}, t_2, \bullet))$, and $r_2 \leftarrow M_2(\text{start}, \bullet)$.
      iv. Assert $r_{1n} = r_{2n} = \text{none}$.
      v. Return $(t_{1n}, \bullet)$.
   (d) Otherwise, if $r_1 = r_2 = \text{none}$, return none.

Lemma 6.11. *If* M_1 *computes* e_1 *and* M_2 *computes* e_2*, then* $M_{iter(e_1 \to p_1, e_2 \to p_2)}$ *computes* $iter(e_1 \to p_1, e_2 \to p_2)$.

Proof sketch. We first argue that the input sequences fed to the children M_1 and M_2 are both collision-free. Since $\mathrm{Dom}(e_1) = \mathrm{Dom}(e_2)$, and the input sequences to the sub-evaluators, γ_1 and γ_2, agree on the location of starts and each character (the only difference is in the token contents), the collision-freedom of γ_2 with respect to e_2 follows from the collision-freedom of γ_1 with respect to e_1.

We will now show that γ_1 is collision-free with respect to e_1. Assume otherwise, and let γ_1 be the shortest prefix to the child which is not collision-free. Say that $|\gamma_1| = n$. The sequence γ_1 necessarily ends with a start signal, $(\mathrm{start}, \bullet_{1n})$, and there is some other thread i, and suffix w_{post} such that both $[\![e_1]\!](\mathrm{th}_{1i} \cdot w_{\text{post}})$ and $[\![e_1]\!](\mathrm{th}_{1n} \cdot w_{\text{post}})$ are defined. Since $\mathrm{th}_{1n} = \epsilon$, the second term can be simplified: $[\![e_1]\!](\mathrm{th}_{1n} \cdot w_{\text{post}}) = [\![e_1]\!](w_{\text{post}})$. The present start signal, $(\mathrm{start}, \bullet_{1n})$, to M_1 was caused either by M_1 itself reporting a result (line 3(c)iii), or by a start signal from the external environment (line 2a).

Since γ_1 is the shortest non-collision-free sequence, all prefixes are collision-free, and M_1 correctly reported results so far. If we follow the causal chain of start signals back to the original external stimulus, we either (*a*) obtain two distinct threads, i' and n' of the parent evaluator, or (*b*) reach the same parent thread i'. The first case leads to the conclusion that the parent function, $\mathrm{iter}(e_1 \to p_1, e_2 \to p_2)$ is simultaneously defined on both words $\mathrm{th}'_i w_{\text{post}}$ and $\mathrm{th}'_n w_{\text{post}}$, contradicting the assumption that the input to the parent evaluator was collision-free. In the second case, let j be the first time the backward causal chains from i and n converge. Consider the string $w_{\text{cex}} = \mathrm{th}_j w_{\text{post}}$: $\mathrm{iter}(e_1 \to p_1, e_2 \to p_2)$ admits two different parse trees on w_{cex}, contradicting the assumption that $\mathrm{Dom}(e_1)$ is unambiguously iterable.

After establishing collision-freedom of the input sequences to the children, correctness of computation follows by induction on the number of iterations. The idea is that the top of the token stack $(t_1, t_2, \bullet)$ fed to M_1 on each restart contains the most recent results from the child evaluators M_1 and M_2 respectively. □

Chained sum. Consider the expression $\mathrm{chain}(e, r)$. From expression consistency, we have $\mathrm{Dom}(e) = [\![r]\!] \cdot [\![r]\!]$: each run of the parent evaluator can therefore be viewed as simultaneously instantiating runs of both M_e and M_r, a hypothetical evaluator for r. The evaluator for r will restart itself, just as in the case of iteration. However, after the first match of r against the input string, another thread of M_e is initialized. We maintain *two* copies of M_r: the first copy is for threads which have not yet been restarted at least once, and the second copy is for threads which have been restarted at least once. We have not constructed evaluators M_r for regular expressions r: we instead maintain the NFA representation as a dictionary $D : Q \to \mathrm{Tok}$ and thereby simulate the evaluator operation. We present the full evaluation algorithm for $\mathrm{chain}(e, r)$ in algorithm 6.8.

Lemma 6.12. *If* M_e *computes* e*, then* $M_{chain(e,r)}$ *computes* $chain(e, r)$.

Proof sketch. First, the dictionaries D_1 and D_2 are used in a "mutually collision-free" way. D_1 and D_2, even when viewed together, cannot cause a token collision. This is formally captured

Algorithm 6.8 $M_{\text{chain}(e,r)}$.

1. Initialize a sub-evaluator M_e for e. Let $\mathcal{A}$ be an NFA which accepts r. Let D_1 and D_2 be two dictionaries, $D_1, D_2 : Q \to \text{Tok} \cup \{\bot\}$, where Q is the state space of $\mathcal{A}$. Initialize the dictionaries as $D_1(q) = D_2(q) = \bot$, for all states q.
2. Given a dictionary $D : Q \to \text{Tok}$ and an input symbol $a \in \Sigma$, define the successor dictionary $\delta(D, a) : Q \to \text{Tok}$ as follows: $\delta(D, a, q) = \bullet$ iff there exists a unique q' such that $D(q') = \bullet$ and $q' \to^a q$ in $\mathcal{A}$. The function δ models the state transition relation of $\mathcal{A}$.
3. Assert $q_0 \notin F$, where q_0 is the initial state of $\mathcal{A}$ and F is its set of accepting states.
4. On receiving $(\text{start}, \bullet)$:
   (a) Invoke $r_e \leftarrow M_e(\text{start}, \bullet)$. Assert that $r_e = \text{none}$.
   (b) Assert $D_1(q_0) = \bot$.
   (c) Update $D_1 := D_1[q_0 := \bullet]$, i.e. add the mapping $q_0 \mapsto \bullet$ to the dictionary D_1.
   (d) Return none.
5. On receiving $a \in \Sigma$:
   (a) Invoke $r_e \leftarrow M_e(a)$, and update $D_1 := \delta(D_1, a)$, $D_2 := \delta(D_2, a)$.
   (b) Assert $\nexists q_{f1}, q_{f2} \in F, \bullet, \bullet'$ such that $D_1(q_{f1}) = \bullet$ and $D_2(q_{f2}) = \bullet'$. At least one of D_1 and D_2 is non-accepting.
   (c) Assert $r_e = (v', \bullet)$ iff for some $q_f \in F$, $D_2(q_f) = (v, \bullet)$. M_e and D_2 return values at the same time.
   (d) If for some $q_f \in F$, $D_1(q_f) = \bullet$:
      i. Invoke $r_e \leftarrow M_e(\text{start}, \bullet)$ and update $D_2 := D_2[q_0 := \bullet]$.
      ii. Assert $r_e = \text{none}$.
      iii. Output none.
   (e) If $r_e = (v', \bullet)$ and for some $q_f \in F$, $D_2(q_f) = (v, \bullet)$:
      i. Invoke $r_e \leftarrow M_e(\bullet)$. Assert $r_e = \text{none}$.
      ii. Update $D_2 := D_2[q_0 := (v \cdot v', \bullet)]$.
      iii. Output $(v \cdot v', \bullet)$.

by the following invariants: (*a*) for all q, $D_1(q)$ and $D_2(q)$ are never simultaneously defined, (*b*) for all distinct q, q' such that either $D_1(q)$ or $D_2(q)$ and either $D_1(q')$ or $D_2(q')$ are defined, for all potential suffixes w', and potential target states $q_f, q'_f \in Q$, if $q \to^w q_f$ and $q' \to^w q'_f$, then either $q_f \notin F$ or $q'_f \notin F$, and (*c*) on all assignments $D := D[q_0 := \bullet]$, $D(q_0)$ was previously undefined. This can be proved using the fact that $\mathrm{chain}(e, r)$ is consistent, and that the original input sequence γ was collision-free with respect to $\mathrm{chain}(e, r)$.

Next, the input sequence fed to M_e is collision-free. Informally, any potential collision between the threads of M_e can be lifted to a collision between the threads of the regular expression evaluators D_1 and D_2 because $\mathrm{Dom}(e) = [\![r]\!] \cdot [\![r]\!]$. It follows that the sub-evaluator M_e is fed a collision-free input sequence γ.

Finally, except for the entry into D_2 (line 5(d)i), all subsequent threads $(v, \bullet)$ contain the value of $\mathrm{chain}(e, r)$ until the last restart signal. This can be proved by induction on the length of the input stream. From this observation, it follows that $M_{\mathrm{chain}(e,r)}$ computes $\mathrm{chain}(e, r)$. □

Proof of theorem 6.1. Given a QRE (resp. DReX expression) e, first construct the evaluator M_e by structural induction on e. Then initialize M_e by sending it the start signal $(\mathrm{start}, \mathrm{nil})$, where nil is the empty token from equation 6.3.1. The algorithm promised in the theorem is simply the machine M_e. The input sequence is vacuously collision-free because there is only one thread. After reading the input stream $w_1 w_2 \cdots w_i$, the evaluator returns a result (t, nil) iff $[\![e]\!](w_1 w_2 \cdots w_i) = t$.

The only evaluators with memory are those for the base expressions, $\varphi \mapsto \lambda$. There are $O(|e|)$ of these evaluators, each with space for exactly one token. Recall that a token $\bullet = (t_1, t_2, \ldots, t_k)$ is a tuple of terms. The size of the token, $|\bullet| = \sum_j t_j \leqslant s_t(e, w_1 w_2 \cdots w_i)$. The evaluation algorithm therefore needs $O(|e| s_t(e, w_1 w_2 \cdots w_i))$ space.

Let $t_s(n)$ and $t_p(n)$ be upper bounds on the time required by M_e, where $n = |e|$, to process start signals and character inputs $a \in \Sigma$ respectively. On receiving a start signal, M_e recursively starts each sub-evaluator, and performs a constant amount of additional processing. We therefore have $t_s(n) = O(n)$. Next, we catalog the following inequalities by consulting the evaluator descriptions. For all $m, n \geqslant 1$:

$$
\begin{aligned}
t_p(1) &\geqslant 1 && \text{Basic expressions} \\
t_p(1 + m + n) &\geqslant 1 + t_p(m) + t_p(n) && \text{Choice, op, substitution} \\
t_p(1 + m + n) &\geqslant 1 + t_p(m) + t_p(n) + t_s(n) && \text{split} \\
t_p(1 + \sum_i m_i) &\geqslant 1 + \sum_i t_p(m_i) + \sum_i t_s(m_i) && \text{iter}
\end{aligned}
$$

Furthermore, any function satisfying the above inequalities is a valid upper bound on the running time of M_e. It follows that $t_p(n) = O(n^2)$. From $t_s(n)$ and $t_p(n)$, we obtain the $O(|e|^2)$ upper-bound on the processing time needed for each subsequent symbol of the input. □

6.5 On Streaming Composition

Our second result in this chapter is that expressions with streaming composition at the top-level can be efficiently evaluated. While the evaluation of expressions in $\mathrm{QRE}_{\gg}$ is certainly decidable, we do not provide any efficiency guarantees for expression evaluation when the streaming composition operator is arbitrarily nested (rather than just being at the top-level).

Theorem 6.13. *If* $e_1 : \mathsf{T}_0^* \leadsto \mathsf{T}_1$, $e_2 : \mathsf{T}_1^* \leadsto \mathsf{T}_2$, …, $e_k : \mathsf{T}_{k-1}^* \leadsto \mathsf{T}_k$ *are consistent, well-typed, and single-use QREs such that* $e = e_1 \gg e_2 \gg \cdots \gg e_k : \mathsf{T}_0^* \leadsto \mathsf{T}_k$ *is well-defined, and* $w \in \mathsf{T}_0^*$ *is an input stream, then* $[\![e]\!](w)$ *can be computed in time* $O(|e|^2|w|)$ *time and with* $O(|e|^2 s_t(e, w))$ *memory, in one left-to-right pass over* w.

Proof. The construction is exactly that shown in figure 2.12. Evaluators M_1, M_2, …, M_k are constructed for each expression. All evaluators are fed the start signal $(\mathrm{start}, \mathrm{nil})$, where nil is the empty token. Each symbol w_i of w is fed to the initial evaluator M_1 in order. Whenever evaluator M_j produces a result v, it is fed to the subsequent evaluator M_{j+1}. The final result is that produced by M_k after all of w is consumed by M_1. Correctness follows trivially by the definition of streaming composition, and because the evaluators M_1, M_2, …, M_k correctly compute their respective functions. □

Chapter 7

Quantitative Approximate Terms

In the previous chapter, we presented a streaming algorithm to evaluate consistent, single-use QREs: this algorithm processes each input symbol in $O(|e|^2)$ time, and requires $O(|e|s_t(e,w))$ memory. The function $s_t(e,w)$ is the size of the largest intermediate term produced while evaluating $[\![e]\!](w)$: in the worst case, this can grow as $O(|e||w|)$. This chapter is concerned with the efficient representation of terms: for numerical domains with the usual arithmetic operations ($+$, min, max, avg), terms can be compressed so that $s_t(e,w) = O(|e|)$, independent of the length of the input stream.

It is well-known that exactly computing the median of n numbers in one pass requires $\Omega(n)$ space [78], and therefore approximation algorithms for the streaming selection problem have been extensively studied [79]. In our setting, for some QREs which involve computing the median, we can maintain multiset summaries as histograms, so that $s_t(e,w) = \text{poly}(|e|\log(|w|)\log(U)/\log(L))/\epsilon$, for a user-specified multiplicative error tolerance ϵ. Here U and L are the highest and lowest values ever generated by applying an elementary transformation to an element of the input stream.

In this chapter, we will explore two simple ideas: (*a*) the use of term rewriting (such as $\min(p+3,4)+2 = \min(p+5,6)$) to compress copyless terms t into representations of size $O(|\,\text{Param}(t)|)$, and (*b*) the representation of multisets S of positive numbers as approximate histograms $h : \mathbb{Z} \to \mathbb{N}$, where h_i is the number of elements of S in the range $((1+\epsilon)^{i-1}, (1+\epsilon)^i]$.

7.1 Motivation

Compressing arithmetic terms. Consider a "large" term, such as $t = \min(\min(x,3)+2,9)$. By routine algebraic laws such as distributivity, we can write:

$$\begin{aligned} t &= \min(\min(x,3)+2,9) \\ &= \min(\min(x+2,5),9) \\ &= \min(x+2,5). \end{aligned}$$

In fact, any real-valued term t built out of a set of parameters P and the operations "min" and "+" can easily be reduced to the normal form:

$$t = \min_{P' \subseteq P}(c_{P'} + \sum_{p \in P'} p)$$

for some constants $c_{P'} \in \mathbb{R} \cup \{\infty\}$ for each subset of parameters $P' \subseteq P$. Unfortunately, when these rules are fully applied, the resulting term has size $O(2^{|P|})$ and is no longer single-use. For example,

$$\begin{aligned} t' &= \min(x + 2, 5) + \min(y + 8, 7) \\ &= \min(12, x + 9, y + 13, x + y + 10). \end{aligned}$$

Observe, however, that the size of the output term is already bounded, regardless of the size of the input term.

The idea behind our simplification routine $\text{simpl}(t)$ is therefore to only propagate constants and not attempt to completely reduce the term to a normal form: $t = \min(\min(x, 3) + 2, 9)$ is reduced to $\min(x + 2, 5)$, but $t' = \min(x + 2, 5) + \min(y + 8, 7)$ is left unchanged. More specifically, simpl guarantees the following properties:

1. For all inputs t, the output $\text{simpl}(t)$ is equivalent to t.
2. For all sub-terms t' of the output $\text{simpl}(t)$, if $\text{Param}(t') = \varnothing$, then t' has already been fully evaluated, i.e. $t' = c'$, for some constant c'.
3. Whenever $t' + c'$ is a sub-term of the output $\text{simpl}(t)$, where c' is a constant, $t' = p'$, for some parameter p'. In particular, t' *cannot itself* be either of the form $t_1 + t_2$ or of the form $\min(t_1, t_2)$. And symmetrically for each sub-term $c' + t'$ of the output $\text{simpl}(t)$. Informally, this limits the number of constants which can appear immediately under a "+" operator.
4. Finally, if $\min(t', c')$ is a sub-term of the output $\text{simpl}(t)$, then t' is either a parameter p' or an instance of the "+" operator, $t' = t_1 + t_2$. In particular, t' is itself *forbidden* from being of the form $\min(t_1, t_2)$. A symmetric claim is made for potential sub-terms $\min(c', t')$ of the output $\text{simpl}(t)$. Informally, this limits the number of constants which occur immediately under a "min" combinator.

With these properties, we can then show that the output $\text{simpl}(t)$ is of size $O(|\text{Param}(t)|)$.

Representing multisets. The second idea is to compactly represent multisets as approximate histograms. Consider the multiset:

$$C = \{12, 9, 14, 20980, 21046\}.$$

We represent C by the array $h : \mathbb{Z} \to \mathbb{N}$, where h_i is the number of values of C which fall in the bucket $((1 + \epsilon)^{i-1}, (1 + \epsilon)^i]$. The index i of the bucket in which a given value v falls can be computed as $i = \lceil \log(v)/\log(1 + \epsilon) \rceil$. In our case, for $\epsilon = 0.01$, we have:

$$h = \{221 \mapsto 1, 250 \mapsto 1, 266 \mapsto 1, 1001 \mapsto 2\}.$$

The histogram itself is small: $|h| = (\log(\max(C)) - \log(\min(C)))/\log(1+\epsilon)$, but is still sufficient to answer rank queries with multiplicative accuracy: the median discretized bucket corresponds to the index $i = 266$. Reversing the bucket index computation, $(1+\epsilon)^i = 1.01^{266} = 14.1$ yields a value within $\pm 1\%$ of the true median, 14.

7.2 Fixing the Space of Operations

We fix an error tolerance $\epsilon > 0$. We are interested in terms of two types: (*a*) the set $\mathbb{R}^{>0}$ of positive real numbers, and (*b*) the set $\mathrm{MSet}(\mathbb{R}^{>0})$ of finite multisets of real numbers. For simplicity, we assume that all real numbers can be stored in $O(1)$ space. We are only interested in copyless terms drawn from the following grammar:

$$\begin{array}{rcl} t_r & ::= & c \mid p_r \\ & \mid & t_1 + t_2 \mid \min(t_1, t_2) \mid \max(t_1, t_2) \\ & \mid & \mathrm{avg}(t_s) \mid \mathrm{select}_k(t_s) \\ t_s & ::= & \varnothing \mid p_s \mid \{t_r\} \mid t_1 \cup t_2 \end{array} \tag{7.2.1}$$

Here $c \in \mathbb{R}^{>0}$ is an arbitrary positive real number, p_r and p_s are parameters of type $\mathbb{R}^{>0}$ and $\mathrm{MSet}(\mathbb{R}^{>0})$ respectively, $\{t_r\}$ is the operation of constructing a singleton multiset out of a real number, and $\mathrm{select}_k : \mathrm{MSet}(\mathbb{R}) \to \mathbb{R}$ for $0 \leqslant k \leqslant 1$ is the operation of finding the $(|A|k)$-th largest element of the multiset A. By fiat, $\mathrm{avg}(\varnothing) = \mathrm{select}_k(\varnothing) = 1$. Note that we have deliberately excluded 0 and negative numbers from consideration. Including them makes the problems under consideration inherently memory-intensive: we will elaborate on this in section 7.4.

Aggregation count. When operations are nested, for example with the QRE

$$e_1 = \mathrm{iter}(\mathrm{split}(\mathrm{iter}(d \in \mathbb{R} \mapsto d, \mathrm{median}), M \mapsto 0, +), \mathrm{median})$$

which computes the median over all months of the median transaction amount, errors accumulated while evaluating sub-expressions grow and contribute to the total error at the parent expression. Therefore, sub-expressions need to be computed with greater accuracy than the overall error tolerance ϵ. We quantify this degree of nesting as the aggregation count.

The AGGREGATION COUNT $\mathrm{agg\text{-}count}(t)$ of a term t is the number of occurrences of the avg and select operators in t. The aggregation count of a QRE e is the maximum aggregation count over all outputs:

$$\mathrm{agg\text{-}count}(e) = \max\{\mathrm{agg\text{-}count}([\![e]\!](w)) \mid w \in \Sigma^*\}.$$

For example, the QRE e_1 just examined has aggregation count 2, while the expression $e_2 = \mathrm{fold}(d \in \mathbb{R} \mapsto d, 0, (x, y) \mapsto \mathrm{avg}(\{x, y\}))$ has unbounded aggregation count.

Finally, the size of the multiset representation h (which is an array, $h : \mathbb{Z} \to \mathbb{N}$) depends on the largest and smallest real numbers under consideration. Let Λ be the set of all basic

operations $\lambda : \Sigma \to \mathbb{R}^{>0}$ which appears in the QRE e. Given a stream $w = w_1 w_2 \cdots w_n$, we define U_w and L_w as:

$$U_w = \max\{\lambda(w_i) \mid \lambda \in \Lambda \text{ and } 1 \leqslant i \leqslant n\}, \text{ and} \tag{7.2.2}$$
$$L_w = \min\{\lambda(w_i) \mid \lambda \in \Lambda \text{ and } 1 \leqslant i \leqslant n\}, \text{ and} \tag{7.2.3}$$

We can now state the central result of this chapter:

Theorem 7.1. *If e is a consistent, single-use, well-typed QRE over the space of terms defined in equation 7.2.1, and has bounded aggregation count c, then for $\epsilon > 0$, $[\![e]\!]$ can be computed to within a multiplicative factor of ϵ with $|e| s_t(e, w)$ space, where*

$$s_t(e, w) = \frac{1}{\epsilon} \mathit{poly}\left(|e| \log(|w|) \frac{\log(U_w)}{\log(L_w)}\right).$$

7.3 Analysis

We present the full term compression routine in algorithms 7.1 and 7.2. Observe the representation of concrete multisets as arrays $h : \mathbb{Z} \to \mathbb{N}$ in algorithm 7.2: the use of arbitrary (i.e. including negative) integers as indexes is somewhat unusual when compared to the more common zero-based data structure present in most programming languages. This can be easily simulated with two zero-based arrays, $h^+ : \mathbb{N} \to \mathbb{N}$ and $h^- : \mathbb{N} \to \mathbb{N}$, representing the elements at positive and negative indices respectively.

Proposition 7.2. *1. For all inputs t, $\mathit{simpl}(t)$ is equivalent to t.*

2. Running $\mathit{simpl}(t)$ requires time $O(|t|^3)$.

3. If t is a purely arithmetic term, then $|\mathit{simpl}(t)| = O(|\mathit{Param}(t)|)$.

Proof sketch. First, observe that for all real-valued terms t, constants c, and operators op, prop-const$(\mathrm{op}(t, c))$ is equivalent to $\mathrm{op}(t, c)$, and prop-const$(\mathrm{op}(c, t))$ is equivalent to $\mathrm{op}(c, t)$. It follows that prop-const(t) is equivalent to t, for all input terms t. From this, it follows that simpl(t) is equivalent to t, for all input terms t, both real-valued and multiset-valued. The second part, regarding the time complexity of simpl is also straightforward to prove.

Proving the third part requires some care. We need to produce constants a, b_ν, b_μ, b_p and b_c such that:

$$|\,\mathrm{simpl}(t)| \leqslant \begin{cases} am + b_\nu & \text{if } \mathrm{simpl}(t) = \max(t_1, t_2), \\ am + b_\mu & \text{if } \mathrm{simpl}(t) = \min(t_1, t_2), \\ am + b_p & \text{if } \mathrm{simpl}(t) = t_1 + t_2, \text{ and} \\ am + b_c & \text{otherwise.} \end{cases}$$

It is easier to work backwards from the constraints that these values must solve. For example, if simpl$(t) = \max(t_1, t_2)$, and both t_1 and t_2 are non-trivial terms, then it must be the case

Algorithm 7.1 simpl(t_r): Arithmetic term simplification.

1. If $t_r = c$ is a constant, or $t_r = p$ is a parameter, then return t_r.
2. If $t_r = \text{op}(t_1, t_2)$ where $\text{op} \in \{+, \min, \max\}$, then let $t_1' = \text{simpl}(t_1)$ and $t_2' = \text{simpl}(t_2)$. Return prop-const$(\text{op}(t_1', t_2'))$.
3. If $t_r = \text{op}(t_s)$ where $\text{op} \in \{\text{avg}, \text{median}\}$, then let $t_s' = \text{simpl}(t_s)$. If $\text{Param}(t_s') = \varnothing$ (i.e. $t_s' \equiv t_s$ is fully specified), return the fully evaluated constant $[\![\text{op}(t_s')]\!]$. Otherwise (if t_s' still contains incompletely specified parameters), return $\text{op}(t_s')$.

Define the constant propagation subroutine, prop-const$(\text{op}(t, t'))$ as follows:

1. If $t' = c'$ is a constant:
   (a) If $t = c$ is also a constant, then return $[\![\text{op}(c, c')]\!]$.
   (b) Otherwise, if $t = p$, then return $\text{op}(p, c')$.
   (c) Otherwise, if $\text{op} = +$:
      i. If $t = t_1 + t_2$, then return $t_1 +$ prop-const$(t_2 + c')$.
      ii. If $t = \min(t_1, t_2)$, then return $\min($prop-const$(t_1 + c'),$ prop-const$(t_2 + c'))$.
      iii. If $t = \max(t_1, t_2)$, then return $\max($prop-const$(t_1 + c'),$ prop-const$(t_2 + c'))$.
   (d) If $\text{op} = \min$:
      i. If $t = t_1 + t_2$, then return $\min(t_1 + t_2, c')$.
      ii. If $t = \min(t_1, t_2)$, then return $\min(t_1,$ prop-const$(\min(t_2, c')))$.
      iii. If $t = \max(t_1, t_2)$, then return $\max(t_1', t_2')$, where
      $$t_1' = \text{prop-const}(\min(t_1, c')), \text{ and}$$
      $$t_2' = \text{prop-const}(\min(t_2, c')).$$
   (e) If $\text{op} = \max$:
      i. If $t = t_1 + t_2$, then return $\max(t_1 + t_2, c')$.
      ii. If $t = \min(t_1, t_2)$, then return $\max(\min(t_1, t_2), c')$.
      iii. If $t = \max(t_1, t_2)$, then return $\max(t_1,$ prop-const$(\max(t_2, c')))$.
2. Otherwise, if $t = c$ is a constant, then return prop-const$(\text{op}(t', t))$.
3. Otherwise (if neither t nor t' is a constant), return $\text{op}(t, t')$.

Algorithm 7.2 simpl(t_s): Multiset compression.

1. Multiset summaries are represented as tuples $(h, \text{acc}, T_r, T_s)$, where:
   (a) $h : \mathbb{Z} \to \mathbb{N}$ is the approximate histogram of all the concretely specified elements of t_s,
   (b) $\text{acc} \in \mathbb{R}$ is the sum of all the concrete elements of t_s,
   (c) T_r is the set of all incompletely specified members of t_s, and
   (d) T_s is the set of all multi-set parameters occurring in t_s.
2. If $t_s = \varnothing$, then return $(\{\}, 0, \varnothing, \varnothing)$, where $\{\}$ is the empty array, $i \mapsto 0$.
3. If $t_s = p_s$, where p_s is an array parameter, then return $(\{\}, 0, \varnothing, \{p_s\})$.
4. If $t_s = \{t_r\}$, let $t'_r = \text{simpl}(t_r)$. If $t'_r = c'$ is a constant, then return
   $$(\{\log_{1+\epsilon}(c') \mapsto 1\}, c', \varnothing, \varnothing).$$
   Otherwise (if t'_r contains parameters), return $(\{\}, 0, \{t'_r\}, \varnothing)$.
5. If $t_s = t_1 \cup t_2$, then let $t'_1 = \text{simpl}(t_1) = (h_1, \text{acc}_1, T_{r1}, T_{s1})$ and $t'_2 = \text{simpl}(t_2) = (h_2, \text{acc}_2, T_{r2}, T_{s2})$. Return:
   $$(h, \text{acc}_1 + \text{acc}_2, T_{r1} \cup T_{r2}, T_{s1} \cup T_{s2}), \text{ where}$$
   $$h_i = h_{1i} + h_{2i}.$$

that:

$$\begin{aligned} a(m+n) + b_\nu &\geqslant am + b_\nu + an + b_\nu + 1, \\ \text{i.e. } b_\nu &\leqslant -1. \end{aligned}$$

Similarly, if $\text{simpl}(t) = \max(t_1, c)$ for some constant c, then we know that t_1 is either a min, $+$, or a parameter p. Therefore:

$$\begin{aligned} am + b_\nu &\geqslant am + b_\mu + 1, (\text{for } t_1 = \min(t_1', t_1'')) \\ \text{i.e. } b_\nu &\geqslant b_\mu + 1, \\ b_\nu &\geqslant b_p + 1, (\text{for } t_1 = t_1' + t_1''), \text{ and} \\ a + b_\nu &\geqslant 3 (\text{for} t_1 = p). \end{aligned}$$

By assembling all such inequalities, it can be seen that $a = 8$, $b_p = -5$, $b_\mu = -4$, $b_\nu = -3$ and $b_c = 1$ is a simultaneous solution. □

The reasoning used in the above proposition can be generalized into a proof of theorem 7.1.

7.4 Notes

We have deliberately omitted negative numbers and the subtraction operator from the term grammar in equation 7.2.1. Bounding the multiplicative error is crucial to composing errors incurred by compound expressions. As the following proposition shows, computing medians with bounded multiplicative error is memory-intensive when negative numbers are also present.

Proposition 7.3. *Consider the QRE* $e = \mathit{split}(\mathit{iter}(d \in \mathbb{R} \mapsto d, \mathit{median}), d \in \mathbb{R} \mapsto d, +)$ *and pick an* $\epsilon \in (0, 1)$. *Any one-pass algorithm which maps input streams* w *to values* v *such that* $(1-\epsilon)[\![e]\!](w) \leqslant v \leqslant (1+\epsilon)[\![e]\!](w)$ *with probability at least* $3/4$ *requires* $\Omega(|w|)$ *space.*

The proof will proceed by reduction from the index problem in communication complexity. The index problem is defined as follows: Alice has an n-bit binary string $a_1 a_2 \cdots a_n$, and Bob has an index $j \in \{1, 2, \ldots, n\}$. Bob wishes to compute the value a_j. Any one-way randomized protocol which succeeds with probability at least $3/4$ needs to transmit at least $\Omega(n)$ bits [71]. See figure 7.1.

Proof of proposition 7.3. We will show that any algorithm which computes e with low memory requirements can also be used to solve the index problem with small data transfer. Because we already know that this is impossible, this will yield the present result.

Let us restrict our attention to sequences of even length. Informally, given a stream $w = w_1 w_2 \cdots w_{2n-1} w_{2n}$, the original QRE e computes the value $\text{median}(w_1, w_2, \ldots, w_{2n-1}) +$

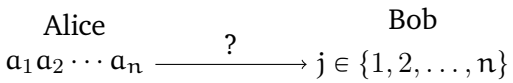


Figure 7.1: The index problem. Alice has a string of n bits, $a_1 a_2 \cdots a_n$, and Bob has an index j. Bob wishes to compute a_j. Only one round of communication is allowed. How many bits does Alice need to send to Bob? In any protocol which succeeds with probability $\geqslant 3/4$, Alice needs to send a message at least $\Omega(n)$ bits long [71].

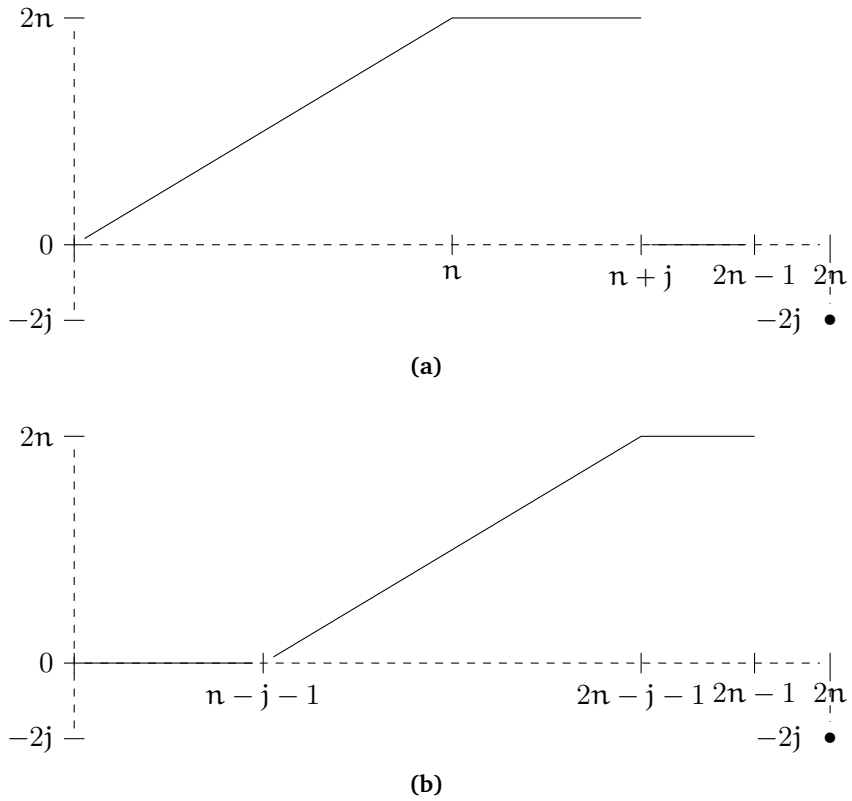


Figure 7.2: Visualizing the malicious input sequence w from the proof of proposition 7.3. In the figure above, we show the original input sequence, and below, we show the same set of values, but with $w_1, w_2, \ldots, w_{2n-1}$ sorted in ascending order. Note that the slope is not really a straight line: the i-th element has y-coordinate $2i + a_i$, i.e. a line with tiny bumps corresponding to the bits a_i. Observe that median$(\{w_1, w_2, \ldots, w_{2n-1}\})$ is the j-th element of the slope, i.e. $2j + a_j$.

w_{2n}. Pick an instance of the index problem $(a_1 a_2 \cdots a_n, j)$, and consider the following input stream to e: $w = w_1 w_2 \cdots w_{2n-1} w_{2n}$, where:

$$w_i = \begin{cases} 2i + a_i & \text{for } i \in \{1, 2, \ldots, n\}, \\ 2n + 2 & \text{for } i \in \{n+1, n+2, \ldots, n+j\}, \\ 0 & \text{for } n + j < i \leqslant 2n - 1, \text{ and} \\ -2j & \text{for } i = 2n. \end{cases}$$

See figure 7.2. Observe that median$(\{w_1, w_2, \ldots, w_{2n-1}\}) = 2j + a_j$. Then, $[\![e]\!](w) = (2j + a_j) - 2j = a_j$. The bit a_j is definitely an element of the set $\{0, 1\}$. Thus, $[\![e]\!](w) \in \{0, 1\}$. Also, the only value within an ϵ-neighborhood of 0 is 0 itself.

Now consider any one-pass algorithm A which computes an ϵ-approximation of $[\![e]\!]$ with probability at least $3/4$. We construct the following protocol for the index problem: Alice runs A on the first n elements of w, all of which are known to her. She passes the state of the machine, i.e. its entire memory footprint, to Bob, who finishes processing A on the remaining n elements of w, all of which are known to him. The size of the Alice-to-Bob message is the number of bits consumed by an execution of A. It follows that any algorithm A to evaluate $[\![e]\!](w)$ with high probability consumes at least $\Omega(n)$ bits of memory. □

Observe that including negative numbers—the final element $-2j$ of the input stream w—is central to the above proof, and it is therefore important to disallow them from the operations of equation 7.2.1.

Chapter 8

Experiments and Case Studies

We have implemented a prototype of the DReX / QRE framework: we will now present some performance measurements, and anecdotally describe the experience of writing QREs and DReX expressions. Our main findings are the following:

1. When the query is expressible in our formalism, very fast evaluation is possible: we routinely observed throughputs of a few hundred thousand symbols per second for small queries, and roughly 10000 symbols per second for large queries ($\approx$ 3600 AST nodes).

2. Representing intermediate results as QATs is a convenient way to control performance: there is a linear relationship between the desired accuracy $1/\epsilon$ and the memory consumed by the algorithm, and increasing the error tolerance ϵ causes measurable increases in throughput.

3. Consistent expressions arise naturally in practice. In fact, the few non-consistent expressions were due to mistakes in the queries we had written, and the consistency checker provided a valuable way to detect bugs.

4. Most natural string processing tasks and many stream-processing queries can be easily expressed as QREs and DReX expressions. The principal limitations appear to be lack of binary predicates and an equivalent of the SQL "`group by`" statement.

8.1 Implementation Details

The programs were written in Java 8, and relied on the symbolic automata library SVPAlib [41] to implement the operations required by the consistency-checking algorithm. The space of characters was all 16-bit UTF-16 code units, and the predicates were unions of character intervals (such as `[a-z, A-Z, 0-9]`). We also made a prototype implementation publicly available at `http://drexonline.com`.

The key component of the polynomial-time consistency checking algorithm is checking equivalence checking of unambiguous NFAs (theorem 2.11). Rather than implementing this

Table 8.1: Evaluated DReX expressions and consistency-checking time.

Expression	Size (# of AST nodes)	Consistency checking (ms)	Description
`delete-comm`	28	12	Delete C++-style ("//...") comments from a file
`insert-quotes`	28	6	Insert quotation marks ("") around each alphabetic substring
`get-tags`	31	6	Extract tags from an XML file
`reverse`	5	1	Reversing a semicolon-separated (";") list
`swap-bibtex`	1663	262	Moves the title of each BibTeX entry to the top
`align-bibtex`	3652	537	Aligning titles in a mis-aligned BibTeX file

algorithm, we instead implemented a naive equivalence checking algorithm based on NFA determinization.

The experiments were run on a typical contemporary desktop computer, running Linux 3.16.7 on a 64-bit Intel Core i7-4770 CPU at 3.40 GHz and with 16 GB of RAM.

8.2 String Processing with DReX

Benchmark expressions. We selected 4 simple string transformation tasks that we believed were representative of typical command-line operations, and two more complicated queries which involved manipulating BibTeX files. We name the queries and their expression sizes in table 8.1.

Baseline. We compared the fast evaluation algorithm of chapter 6 with a straightforward dynamic-programming based evaluation technique suggested by the semantics. Given an expression e, and an input stream w, the idea is to maintain a table $\text{memo}(\mathtt{f}, \mathtt{i}, \mathtt{j}) : \mathbb{D} \cup \{\mathtt{undef}, \mathtt{uncomputed}\}$ which maps sub-expressions $\mathtt{f}$ of e and substrings $w[\mathtt{i}..\mathtt{j}]$ to values $[\![\mathtt{f}]\!](w_\mathtt{i} w_{\mathtt{i}+1} \cdots w_\mathtt{j})$. The semantics of function expressions provide a natural way to compute $\text{memo}(\mathtt{f}, \mathtt{i}, \mathtt{j})$ in terms of its sub-expressions $\mathtt{f}'$ and substrings $w[\mathtt{k}..\mathtt{l}]$ with $\mathtt{i} \leqslant \mathtt{k} \leqslant \mathtt{l} \leqslant \mathtt{j}$. The complexity of this algorithm is $O(|e|, |w|^3)$.

Results. We ran each benchmark expression 16 times, on a range of input strings, and report the average running time in figures 8.1 and 8.2. We randomly generated input strings

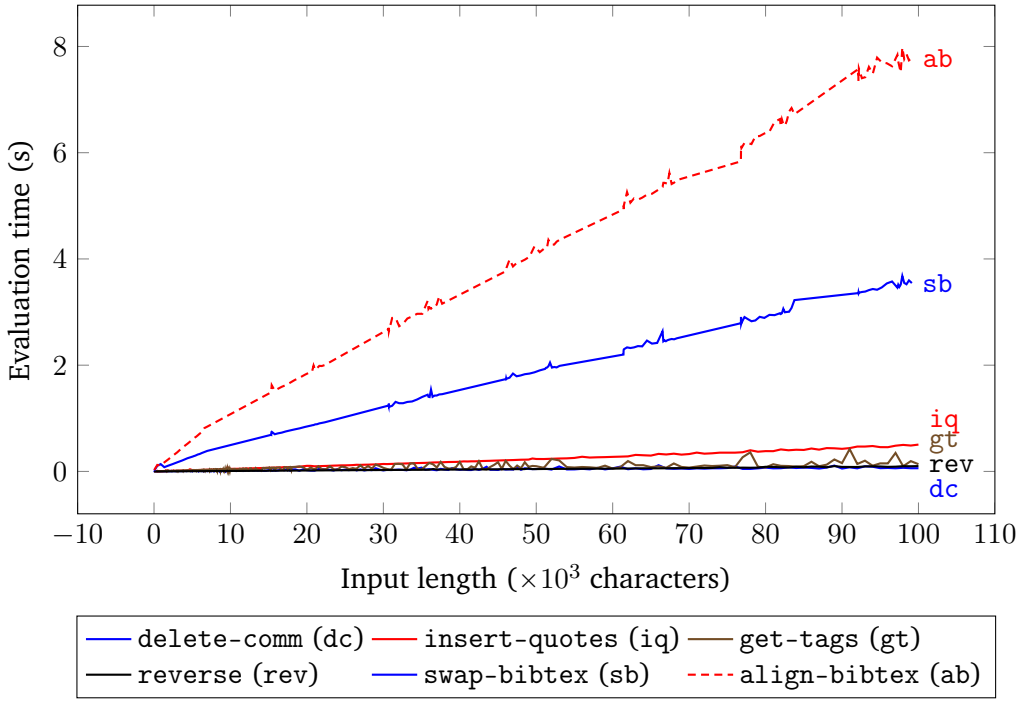


Figure 8.1: Performance of the streaming DReX evaluation algorithm. For comparison, observe that the baseline non-linear evaluation algorithm in figure 8.2 is much slower.

for the four simple expressions, and randomly created actual BibTeX files for `swap-bibtex` and `align-bibtex`. A timeout of 60 seconds was chosen for all experiments.

In table 8.1, observe that (even with the naive non-polynomial implementation) consistency checking always finishes in less than 1 second. Next, the performance of the streaming evaluation algorithm in figure 8.1 is indeed linear, as we would expect from our theoretical analysis. The evaluator finishes computation in less than 8 seconds, even for files as large as 100000 characters, and reasonably large expressions such as `align-bibtex`. In contrast, the baseline dynamic programming algorithm does not scale to strings longer than a few thousand characters.

We also programmed the benchmark transformations in sed, AWK, and Perl. Regular expression based substitution, as present in all of these tools, is very efficient and usable to substitute or delete substrings based on patterns. This includes the benchmark programs `delete-comm` and `insert-quotes`, for which the sed implementations were ≈ 6 times faster than the DReX ones. On the other hand, `reverse`, `swap-bibtex` and `align-bibtex` were hard to express in line-based tools such as sed and AWK. The Perl implementations of these functions were ≈ 2 times faster than ours.

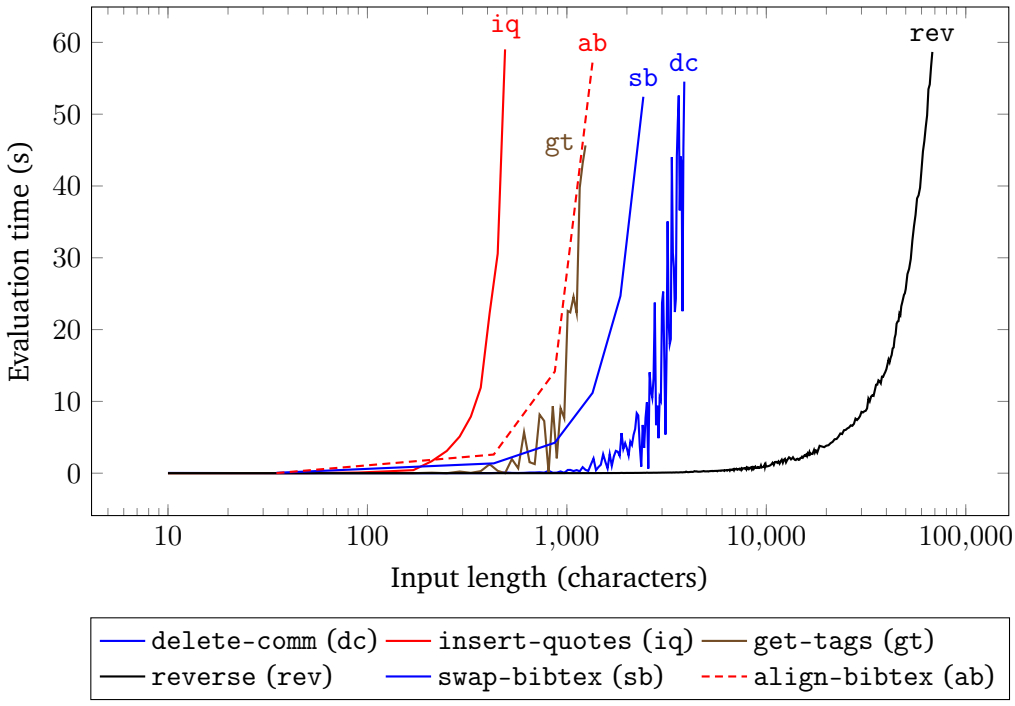


Figure 8.2: Performance of the baseline DReX evaluation algorithm.

Table 8.2: QRE evaluation performance. This table lists the time and space needed to process 10 million tuples of the respective data streams, with $\epsilon = 0.01$.

Query	Space ($\times 10^3$ bytes)	Throughput ($\times 10^3$ tuples / s)	Description
Bank 1	26.3	584	Median amount deposited during each month
Bank 2	21.0	788	Median monthly closing account balance
Bank 3	1.06	12500	Median number of transactions on the "busy day"
Bank 4	26.2	557	Number of consecutive trailing months for which monthly savings are in the top / bottom quartile
NEXMark 1	7.58	60.5	Median bid for items in the category "100"
NEXMark 2	16.4	48.3	Maximum of the median bid price from US and non-US customers
Linear Road	11.1	137	Average over all lanes of the median lane speed

8.3 QREs and Query Approximation

Bank transaction data. We implemented a synthetic data stream indicating the transactions of a customer with a bank. The stream is parameterized by U, the maximum amount that the bank allows the customer to deposit in a single transaction. There are three kinds of values in the data stream: deposits of d dollars, where $1 \leqslant \mathsf{d} \leqslant \mathsf{U}$, end-of-day markers D, and end-of-month markers M.

The elements of the data stream are drawn independently and identically, by the following process: (*a*) with probability $60/91$, the value is a deposit, where the amount d is a real number chosen uniformly at random from the range $[1, \mathsf{U}]$, (*b*) with probability $30/91$, the value is the end-of-day marker D, and (*c*) with probability $1/91$, the value is the end-of-month marker M. We ask four questions of the data stream:

1. Over all calendar months, what is the median amount deposited into the account during a single month?

2. Over all calendar months, what is the median account balance at the end of the month?

3. The bank defines the "busy day" of each month as the day on which the most number

of transactions are made. Over all months, what is the median number of transactions the customer makes on the busy day?

4. To monitor the savings of the customer, the bank computes the number of consecutive trailing months for which monthly savings are in running top / bottom quartile.

We intended to highlight the memory-usage profile of the expression evaluator, and so deliberately included many more medians than would be typically seen.

In table 8.2, we list the time and space needed to evaluate the queries on the bank transaction data stream. The measurements were made for a fixed stream of 10 million tuples, and for a fixed accuracy $\epsilon = 0.01$. In figure 8.3, we plot the performance of the system on the bank transaction data stream, as a function of stream length and desired accuracy. We measured space usage by serializing the evaluator state into a sequence of bytes, and by measuring the length of this serialized representation. The blue, red, brown, and black lines are results obtained from queries 1 to 4 respectively. The first plot measures the space needed for query evaluation as a function of the stream length (for a fixed $\epsilon = 0.01$). The second plot measures the space needed for QRE evaluation as a function of the desired accuracy ϵ (for a fixed stream length of approximately 10 million tuples). The last plot measures the throughput of the evaluation algorithm as a function of the desired accuracy ϵ.

Notice that with increasing stream length, the memory usage initially climbs rapidly, but quickly settles into a region of more conservative growth, consistent with the theoretical claim that the space required is logarithmic in the length of the data stream. Also observe the linear relationship between $1/\epsilon$ and the space needed for evaluation, as predicated by theorem 7.1. Finally, observe that the throughput of the QRE evaluator decreases somewhat with increasing accuracy requirements (larger values of $1/\epsilon$). There are two major components during QRE evaluation: maintenance of state, and updating the terms. This explains the slow (but still extant) drop in throughput with decreasing ϵ.

Query 3, which measures the median number of transactions on the busy day, is notable for its nearly constant memory usage regardless of the stream length or desired accuracy. This is because of the nature of the data source: since the probability of a deposit transaction is twice the probability of an end-of-day marker, the number of transactions in a day is typically a small number. Therefore, there are typically only a few distinct buckets in the approximate multiset summary with non-zero values, regardless of the stream length.

NEXMark. Next, we used the NEXMark data stream [90], which is a simulated sequence of events from an auction house. There are three kinds of tuples: (*a*) Auction tuples, which announce an item to be auctioned. This includes the item id, seller, and the start and end times of the auction. (*b*) Person tuples, including the person's id, name, and address. Persons both put up items for auction, and place bids on items being auctioned. And (*c*) bid tuples, which reference the item considered, the person making the bid, and its value.

We evaluate two queries over the NEXMark data stream: (*a*) What is the median bid made for items in category 100? And (*b*) maximum of the median bid price from the US and non-US countries. The performance of the query evaluator on each of the queries is listed in

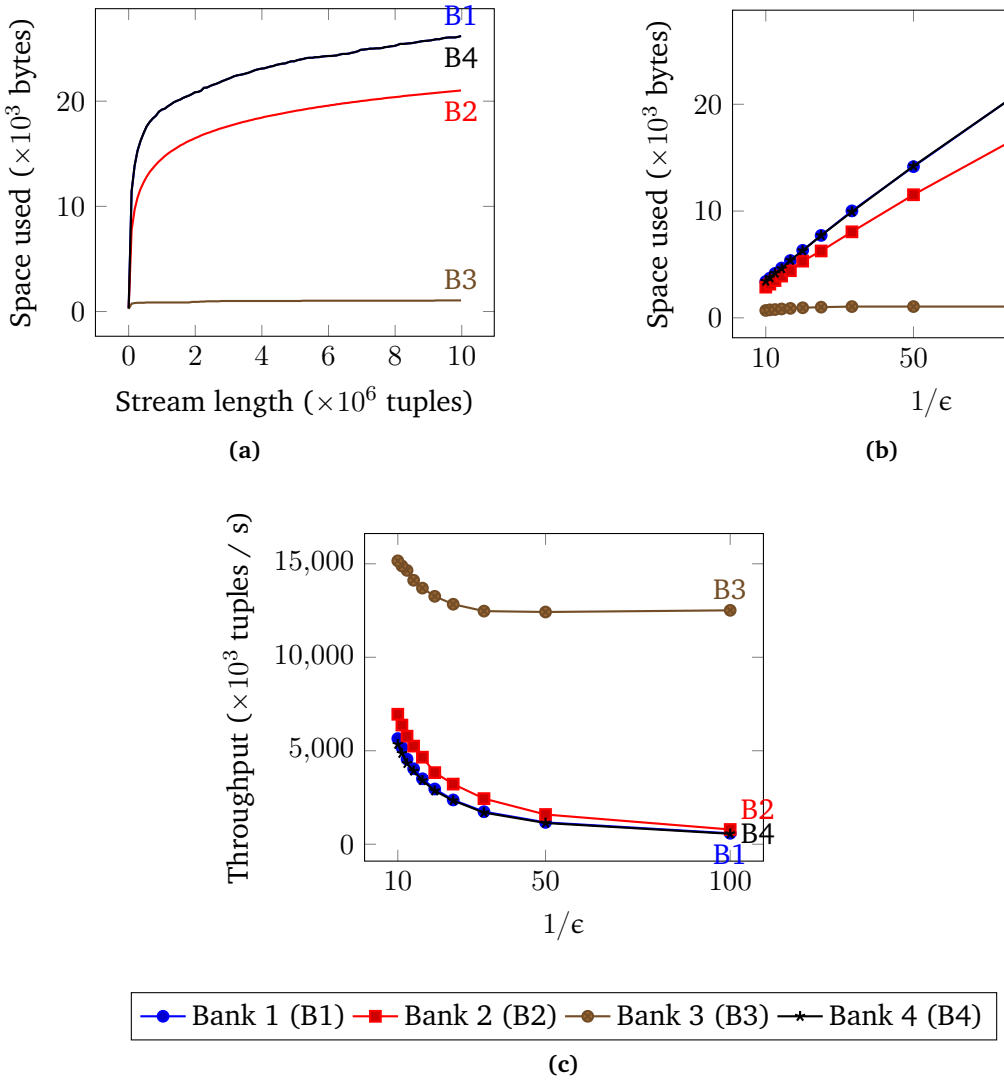


Figure 8.3: QRE evaluation performance on bank transaction data.

Table 8.3: Comparing QRE evaluation performance with a handcrafted implementation.

	Space ratio	Throughput ratio
Bank 1	$1.44\times$	$37.1\times$
Bank 2	$1.71\times$	$36.2\times$
Bank 3	$15.5\times$	$51.5\times$

table 8.2. With varying ϵ and stream length, the trends in performance are similar to those figure 8.3, but have not been plotted.

Linear Road. Our final data source was the Linear Road benchmark [25]. The data is a simulation of an expressway system, and models a city with a reactive tolling system, where the tolls charged are a function of the freeway conditions. There are four types of tuples in the data stream: vehicle position tuples, generated by vehicles on the freeway reporting their position to the server, and the remaining tuples encode queries, including requesting the account balance, daily expenditure, and travel time.

The traffic simulation covers a physical time of three hours, and is parameterized by L, the number of expressways in the system. A flat listing of tuples for $\mathrm{L} = 1.0$ is available at `http://www.it.uu.se/research/group/udbl/lr.html`, which we used in our experiments. We computed the average over all lanes of the median lane speed. In Table 8.2, we list our measurements for all queries considered. Observe that even for large data streams of 10 million tuples, a result with accuracy $\epsilon = 0.01$ can be computed using under 20 KB of memory. Since the data stream covers a physical time of 3 hours, the throughput of of 137000 tuples per second is also sufficient this application.

Results. Finally, in table 8.3, we compare the performance of the QRE evaluation algorithm against a implementation of the query programmed directly against the QAT API. Note that while the QRE implementations are significantly slower than the direct QAT-based implementation, the space usage is not much higher. We anticipate that with better engineering, the time penalty of QREs can be brought down even further (as for example, in [56], where a similarly motivated system for string transformers achieves throughputs in the 1 Gbps range).

8.4 Anecdotal Account of User Experience

We were able to easily program several non-trivial transformations without having to worry about efficiency.

Expression consistency. The most important restriction of the system is that the evaluated expressions be consistent. The natural way to express our example queries always turned

out to be consistent. Moreover, in many cases the consistency check helped us to identify sources of ambiguity that made the query incorrect. For example, we had mistakenly concatenated the sub-program `copy-spaces` $=$ iter(isSpace(d) $\mapsto$ d), which copies all whitespace characters (including tabs and newlines), with itself. In addition to certifying consistent queries, the checker also provides witnesses to non-consistent expressions. In the case of split(`copy-spaces`, `copy-spaces`), the checker warned us that the expression was ambiguous for the input string "\n". This was clearly a bug in our script, would have otherwise led to unexpected behavior, and the counter-example witness helped us to eventually write the correct query.

Function composition. Regular string transformations are closed under function composition (this is similar to but not the same as streaming composition). Unfortunately, we have the following intractability result about DReX$+\{\circ\}$: given an expression e in the class DReX$+\{\circ\}$, determining whether $[\![e]\!](\epsilon)$ is defined is PSPACE-complete [14]. We have therefore omitted function composition from the calculus. While it would be convenient in desugaring the front-facing language presented at `drexonline.com`, we have otherwise not found instances where the operator was crucially required.

Binary predicates, `group by`, and sliding windows. Both NEXMark [90] and Linear Road [25] come with a library of benchmark queries. We were unable to express many of these pre-specified queries as QREs. These inexpressible queries are often similar to "What is the average time a car spends on the freeway?", or "What is the average auction closing price over all items?" As pointed out by Kostas Mamouras, evaluating such queries involves separating the input stream by the car id, or the item id of the article being auctioned, and then performing some computation over this sub-stream. We are unable to express this operation, very similar to `group by` from SQL, using QREs. In our later work on this subject [20], we explicitly add an operator called "map-collect": with this additional operator, all NEXMark and LinearRoad queries are easily expressible.

Another inexpressible query is to recognize increasing sequences of numbers. Our basic expressions are of the form $\varphi \mapsto \lambda$, where $\varphi : \mathbb{D} \to$ Bool is a unary predicate, but the pattern "Increasing sequence of numbers" crucially involves the two-place predicate $\varphi(x, y) =$ "$x < y$".

Sliding windows are a common idiom in many stream-processing operations. For example, "What is the average vehicle speed over the last 60 minutes?". While these operations are expressible as QREs, they typically involve concatenating an expression with itself many times. Both syntactic sugar and evaluator-level heuristics to improve performance on these expressions would be very useful.

Chapter 9

Related Work

We may broadly categorize the relevant research literature into four buckets: (*a*) *parsing technologies*, ranging from regular-expression based tools such as sed, AWK, Perl, and SNOBOL, to more complex pieces of software, such as parser generators and combinators; (*b*) foundational work on *transducers and string constraint solvers*, motivated by applications in biology and natural language processing, and more recently by applications in the static analysis of web-applications and other string-processing programs; (*c*) *streaming databases*, targeted towards applications in processing internet-scale data, and which, in contrast to traditional relational databases, view the data as being continuously updated, and the queries answered as being mostly constant; and (*d*) fundamental work on *streaming algorithms*, which seeks to answer specific queries with extremely tight space- and time-guarantees. We will now summarize the related work in each of these areas.

9.1 Parsing Technologies

sed. The command line tool sed extends traditional regular expressions with primitives for substitution. The result is a powerful text-processing system which can express many of the same text processing operations that we considered in chapter 8. The underlying computational model consists of two buffers, named the "hold space" and "pattern space" respectively, and various commands to move data between these buffers. In particular, these commands include the ability to loop and repeatedly perform another sequence of commands. Turing-completeness arises because of these looping commands and because the two buffers can be used as stacks. Consider the string reversal function:[1]

```
sed '/\n/!G;s/\(.\)\(.*\n\)/&\2\1/;//D;s/.//'
```

Informally, the script may be read as:

1. Split the string into a one-character-long head ("\1") and the remaining tail ("\2").
2. Store "\2\n\1" on the pattern space.

[1] https://groups.google.com/forum/#!topic/comp.unix.shell/s4hfVpJYX2Q

3. Repeat on "\2" until it is empty.

Observe that the presence of both buffers and of the command to repeat steps 1 and 2 ("`//D`") is essential to the working of this script.

SNOBOL [59]. SNOBOL is an early system for string processing. The basic constructs include matching against the following patterns (note the absence of full Kleene-*):

$$P ::= w \in \Sigma^* \mid P_1 \cdot P_2 \mid P_1 + P_2,$$

and goto-based control flow. As with sed, Turing-completeness seems essential to the expressive power of the language.

One immediate way to limit the expressive power and allow static analysis is to eliminate loops and arbitrary gotos. This however makes most interesting functions, such as replace every occurrence of a character with another, immediately inexpressible. Another way to limit its expressive power is to allow arbitrary gotos, but disallow other features such as integer-valued registers. It is not immediately clear that this class of SNOBOL programs always terminates: it is possible that various two-stack-based tricks, such as those used in sed, can still be applied in this restricted system. Another possibility for this restricted class is that programs always terminate, but often require super-linear time. It is an interesting research problem whether equivalence-checking an pre-/post-condition checking is decidable for this restricted class of SNOBOL programs.

Summary. Other tools in this general category include SNOBOL [59], AWK, Perl, and PCRE [65]. The notion of "regular expressions" in these practical systems usually includes many powerful extensions such as back-references: this makes the parsing problem NP-hard [88]. The usual approach is to design an backtracking evaluator: these are usually very fast for common cases, but predicting performance degradation is hard. Furthermore, as with sed and SNOBOL, many of these tools are Turing-complete, making mechanical verification very difficult.

All these string-manipulating systems lack the small core of DReX, making any kind of programmer support a challenging task. Furthermore, it appears that the features which cause Turing-completeness are essential to their expressive power. While removing these features will certainly make the main static analysis problems (equivalence and pre-/post-condition checking) decidable, it severely limits their expressiveness, and it is not immediately clear that some of the other possibilities of the QRE / DReX framework, i.e. potential for automatic parallelization, query optimization, better support for debugging etc., can be provided even in these restricted systems.

Kleenex. Kleenex is a tool to determinize non-deterministic SSTs [56]. The input to Kleenex is an EBNF-like format which combines regular parsing with imperative parser actions. The Kleenex compiler converts this to a deterministic machine which guarantees the lexicographically smallest parse tree. Kleenex is an extremely well-optimized system—providing optimization guarantees such as early commits—and sustains throughputs of about 1 Gbps on contemporary desktops.

Parser combinators. Parser construction tools such as parsing expression grammars [54] and parser combinators have received renewed attention in recent years. Some of these systems also come with very good programming language integration, such as the Parsec library [74] and the Boost Spirit framework [44]. One concrete pattern which is difficult to express with traditional parser combinators are expressions such as combine(e, f) or op(e, f). Recall that the feature most similar to these is that of language intersection, and this involves simultaneously computing multiple parse trees and performing several different computations on the same input string. In general, the focus of many of these systems focus is on efficiently constructing parse trees, in contrast with DReX expressions where our focus is on efficiently computing values with provable bounds on time- and space-complexity.

9.2 Transducer Models

Finite state transducers. In chapter 3, we defined regular string transformations operationally, as those machines which can be implemented by streaming string transducers (SST). SSTs were originally introduced by Alur and Černý [9, 10], as a model by which to verify properties of single-pass list processing programs. Their main results include: (*a*) the decidability of SST equivalence and pre-/post-condition checking in PSPACE [10], and (*b*) the equi-expressiveness of SSTs and the functions expressible as formulas in monadic second-order logic (MSO) [9].

The concept of specifying string- (and more generally graph-) transformations in MSO was first considered by Courcelle [38]. Engelfriet and Hoogeboom [51] then showed that MSO-definable string transformations are the same as those which can be implemented by two-way finite state transducers (2-FST). This mirrors the classical results of Büchi [34], Elgot [50], and Trakhtenbrot [89], who showed that languages definable in the MSO-logic of one successor, S1S, are exactly those which are accepted by finite automata. This correspondence prompted [51] to label the class of functions definable by two-way finite state transducers as "regular" string transformations.

It is well-known that, unlike in the case of finite-state acceptors, 2-FSTs are strictly more expressive than one-way transducers: this gap includes several "natural" functions such as string reversal ($w \mapsto \text{reverse}(w)$) and string repetition ($w \mapsto ww$). However, the post-image of a regular language with respect to a 2-FST need not be regular [6]. While the equivalence problem for non-deterministic 2-FSTs is undecidable [58], the equivalence of *deterministic* 2-FSTs is decidable [64]. Closure under composition of deterministic 2-FSTs was established in [37].

Both SSTs and 2-FSTs share many of the same appealing properties as DReX: simple, efficient evaluation algorithms, decidable equivalence and pre-condition computation, and robust closure properties. Still, they are both inherently operational models which force programmers to explicitly think about what state needs to be maintained and how it should be updated. We are not aware of a prior characterization of regular string transformations by a set of combinators analogous to the characterization of regular languages by regular expressions.

Weighted automata and regular cost functions. String-to-number transformations have traditionally been modelled using weighted automata [49]. Weighted automata find application in natural language processing in problems such as transliteration and optical character recognition [69] and in the modeling of probabilistic systems [26]. First, fix a semiring, $(\mathbb{D}, +, \cdot, 0, 1)$. Now, a weighted automaton is an NFA A over the input alphabet Σ, where each transition is labelled with a weight $d \in \mathbb{D}$. Given an input string w and an accepting path $\sigma = q_0 \to^w q_f$ through A, the weight of σ is defined to be the product of all the transition weights along the path. The weight of w is defined to be the sum of the weights of all accepting paths.

Equivalence for weighted automata over the tropical semiring, $(\mathbb{Z} \cup \{\infty\}, \min, +, \infty, 0)$ is known to be undecidable, by a reduction from Hilbert's tenth problem [72], and by a reduction from the halting problem for two-counter machines [7]. Weighted automata over $(\mathbb{Z} \cup \{\infty\}, \min, +, \infty, 0)$ are strictly more expressive than regular cost functions, while single-valued weighted automata are equi-expressive as regular cost functions with $\mathbb{D} = \mathbb{N} \cup \{\infty\}$ and $G = \{+\}$.

We introduced the model of regular cost functions in [13], mostly by a motivation to generalize the previous concepts of regular string transformations and regular tree transformations [12] to quantitative functions. A regular cost function is one which can be implemented by a streaming string-to-term transducer, which maps input strings to terms over the output domain. In certain situations, such as over the cost domain of $(\mathbb{Z}, \min, +)$, there are equivalent models which operate over concrete values: we call these "cost register automata (CRA)". The decidability of various analysis problems for CRAs depends largely on the cost domain: for example, we solved register minimization problem for the simple class of *additive* CRAs in [21], while the register minimization problem for SSTs is still open.

Symbolic automata. In contrast to traditional regular expressions, where the basic expressions are of the form a, where $a \in \Sigma$ ranges over a *small, finite* alphabet, our basic expressions were of the form $\varphi \mapsto \lambda$, where $\varphi : \Sigma \to \text{Bool}$ was a *symbolic single-element predicate*. Allowing such predicates in the basic expressions was extremely helpful to handle large alphabets such as Unicode, and even potentially infinite alphabets such as the set of all measurements that can be emitted by a sensor. This choice was motivated by the recent development of symbolic automata and symbolic transducers [93, 94]. Symbolic automata are similar to traditional automata, except that the edges may be labelled by predicates over the alphabet. For example, an edge may be of the form $q \to^{\text{isVowel}} q'$, which succinctly encodes 5 different transitions: $q \to^a q'$, $q \to^e q'$, $q \to^i q'$, $q \to^o q'$, and $q \to^u q'$. Assuming reasonable properties over these predicates, i.e. that the space of predicates is closed under boolean combination, and that predicate evaluation and satisfiability checking are both decidable, symbolic automata share many of the same appealing properties as their classical counterparts: deterministic and non-deterministic SFAs are expressively identical, and string membership testing, SFA equivalence, containment and minimization [43] are all decidable.

When binary predicates are allowed, such as $q \to^{x,y,\text{if } x \leqslant y} q'$, meaning "Consume two elements x, y on the $q \to q'$ transition, where the transition is enabled if $x \leqslant y$", it is known

that equivalence is undecidable [42]. This result was the primary reason that we disallowed binary predicates in the QRE / DReX formalisms. This is admittedly a severe restriction from a practical perspective, although our later experience with StreamQREs has shown that map-collect (a variant of the SQL `group by`) greatly lessens this burden.

Static analysis of string transformers and constraint solving. One of the important applications of symbolic automata has been in the static analysis of string transformation programs, particularly in security-critical applications. This has resulted in tools such as Bek [67], Kudzu [82], Saner [27] etc. These languages are usually tightly coupled to the underlying transducer model, forcing the programmer to think in terms of low-level primitives such as a machine making a single left-to-right pass over the input. Furthermore, these languages capture a strict subset of the class of regular string transformations, disallowing functions such as string reversal.

A common theme in the formal verification of string manipulating programs is to assert state invariants, and use constraint solvers to find strings which violate these invariants. Starting with the seminal work of Makanin on solving string equations [77], this approach is exemplified by tools such as HAMPI [55], Norn [5], Z3-Str [100], and the string solver in CVC4 [76]. In contrast, our proposed approach to the static analysis of DReX—also adopted by tools such as Bek—is to model sanitizers as transducers and prove properties using properties of the transducer model.

Matching regular expressions. Our definition of consistent DReX and consistent QREs in chapter 2 was guided by several current challenges concerning regular expressions. Matching extended regular expressions (traditional regular expressions extended with complement and intersection as fundamental operations) is a difficult open problem [81]. Unrestricted choice (else) and cost operations (op) are a potential route to encode complementation and intersection respectively. The consistency rules provided a simple technique to avoid these difficult anti-patterns.

Myers [80] applied the four Russians technique to the regular expression matching problem to obtain (surprisingly efficient!) complexity improvements. Veanes et al [95] carefully considered the execution of symbolic finite state transducers and engineered data-parallel evaluators which are much faster on real-world HTML encoders and decoders. We plan to investigate whether any of these techniques can be applied to DReX and QREs.

Expression inference. One of the main open problems with DReX and QREs is of learning function expressions. The inference procedure can learn either from a given set of input-output examples, or in a setting similar to that of Angluin's L* algorithm [22], i.e. where a teacher is able to answer evaluation and counterexample queries. Bojańczyk has made some progress towards solving this problem for streaming string transducers [31]: if the teacher is also able to provide origin information, i.e. informally trace each character of the output back to the specific input character which produced it, then he is able to obtain a machine-independent characterization of SSTs similar to the Myhill-Nerode theorem, and extends this to an L*-like learning algorithm.

Botinčan and Babić investigate the problem of automatically learning symbolic transducers from input-output examples [32], but they are primarily restricted to one-way transducers.

Sumit Gulwani's work [61, 84, 62] on learning string transformations in spreadsheets is a notable example, both for a system that is able to quickly learn complicated string transformations, and for how useful programming-by-example systems can be to non-programmers. The major differences between the FlashFill language and DReX are that: (*a*) every sub-expression consumes the *entire* input string, potentially producing only a small substring of the final output, and (*b*) unlike in DReX, choice operators can only be placed at the top-level, and function essentially like a switch-case statement, where the conditions are regular expressions. It appears that character-wise transformations ("Convert all upper-case to lower-case letters") and functions like shuffle are difficult to express in the FlashFill formalism. However, it is difficult to pin the relative expressive powers of DReX and FlashFill with certainty. The 2016 edition of the SyGuS competition included a track to learn string transformations from input-output examples, inspired primarily by systems such as FlashFill [18].

9.3 Streaming Databases

Compared to traditional databases, where the logical model is of instantaneous query execution, and where the insertion and deletion of records invalidates the results of previous queries, streaming databases usually assume that the queries are (mostly) static, and that the underlying data is incrementally updated. Early work on streaming databases included systems such as Tapestry [87] for email and bulletin-board messages. *Materialized views* have also been proposed as a way to model continuously changing data [63, 68]: a materialized view is an object containing the results of a query, and the technical problem is to maintain and update these materialized views as the underlying relations change. There is also a large body of more recent work on data stream management systems, such as Aurora [4], Borealis [3], and STREAM [23, 24].

Query models. The query models typically supported by these systems is exemplified by CQL (Continuous Query Language) [24]. There are two types of objects in the CQL universe: "streams" and "relations". Sliding-window operators (for e.g., "Collect the last 100 tuples observed on stream S", or "Collect all tuples observed within the last 60 seconds on S") provide a way to convert streams to relations, and the sequence of updates to each relation automatically forms a stream (for e.g., "Emit all elements in R_t which were not present in R_{t-1}"). The full power of a relational query language such as SQL is available to operate on relations.

In terms of expressiveness, CQL and QREs are incomparable: It is possible to express joins over dynamically changing relations in CQL, and it gracefully extends to simultaneously handle multiple input streams. On the other hand, with QREs, we can impose complicated structure over the data stream, and this is difficult to express with CQL's more constrained sliding window model-of-state.

Aurora [4] and Borealis [3] model a continuous query as a data-flow program, where

the programmer specifies a network of stream operators. Each stream operator is a simple transformation such as a filter, project, persistence, or streaming join. In our later work on StreamQREs [20], we introduce a new operator named "map-collect" (similar to `group by` from SQL), and observe that streaming joins are not crucial to practical expressiveness. It is an interesting direction of future work to extend the underlying model of QREs to multi-tape transducers, and attempt to simulate streaming joins.

One important limitation of sliding windows is that the length of the window is determined ahead of time. This is somewhat mitigated in systems such as CEDR [28] and Punctuated Streams [91, 75] where the use of punctuations triggers the closing of a window. Still, our use of regular patterns to specify the parsing makes QREs more general.

Systems. There are a number of robust production-ready stream processing systems. Of these, PipelineDB exemplifies the idea of materialized views as continuous queries [2]. Apache Storm is a distributed computing system specialized for stream processing [1]: the programmer specifies a processing topology, i.e. a graph where each node is a processing element and the edges represent data flows. The result is a very well-engineered system, and involves consideration of issues such as node failures and network topologies. Spark Streaming adds support for micro-batch based stream processing to the seminal Spark framework [98, 99]. All of these are well-engineered systems, and pay careful attention to practical issues such as network latencies and node failures. However, they mostly require manual implementation of individual stream processing elements, and do not provide high-level views of the underlying data stream. One potential direction of future work is to add support for QREs while creating individual stream processing elements (called "bolts") in a system such as Storm.

One prominent commercial product is the IBM Streams platform [66], which motivated projects such as ActiveSheets [92]. ActiveSheets functions as a spreadsheet plugin. Data streams are published by sources such as financial markets: these streams may be "imported" into a spreadsheet as a periodically changing collection of cells. For example, cells "A1" through "A50" may display the values of the last 50 transactions from the stock exchange. A small set of extensions to traditional spreadsheet formulas specifies when to tick the output cells. By allowing the programmer to maintain arbitrary state, ActiveSheets mostly subsumes QREs in expressiveness. However, the programmer has to explicitly reason about what intermediate values need to be computed and what state needs to be maintained, and this lacks the benefits of declarative programming. Nevertheless, by presenting a spreadsheet interface which should be familiar to a much larger group of computer users than just programmers, ActiveSheets highlights the importance of usability in these systems.

Complex event processing. Cayuga [45, 46, 33] is an example of a system for complex event processing (CEP). In contrast to QREs, which deal with computing numerical summaries of data streams, CEP systems typically implement a publish/subscribe query model, where a server publishes a stream of events, and clients can subscribe to event patterns on whose occurrence they wish to be notified. Just as QREs, they can express complicated automaton-based stateful properties. They can often express more involved properties, such as "Notify

on the first sale of the stock for a price lower than the previous sale".

NiagaraCQ [35] uses query grouping and other optimizations to build an efficient system that can simultaneously execute multiple queries at internet scale. CEP has many applications, including in processing streams of RFID readings [96]. See [40] for a broad survey.

Benchmarks. There are several benchmarks produced by the research community working on streaming databases: this includes (*a*) the query repository compiled by the STREAM project [60], (*b*) the NEXMark benchmark describing a sequence of events simulated online auction and an associated set of queries [90], and (*c*) the Linear Road benchmark involving a sequence of events concerning a freeway traffic management system [25]. These are helpful to measure the performance and expressiveness of QREs in realistic settings. The queries that we were unable to express provided a guide to extensions to the QRE framework. We note that these benchmarks were created with existing stream processing platforms in mind: in particular, they do not require temporal and stateful computational primitives such as iteration or global choice, and do not exercise all of the QRE framework.

The recent Yahoo Stream Processing benchmark [36] includes a data stream generator and a target query: the intent is to benchmark distributed stream processing platforms such as Apache Storm, Flink, and Spark Streaming. Because of its focus on measuring datacenter-scale effects such as those caused by network latencies and buffering, it is somewhat less relevant to us than the remaining benchmarks. Still, the single-machine instance of this benchmark, which includes a Storm reference pipeline is a valuable guide to the processing speed of current data stream processing systems.

9.4 Streaming Algorithms

There is a large body of work on streaming algorithms, beginning with the seminal work of Munro and Paterson [78], Flajolet and Martin [53], and the work of Alon, Matias and Szegedy on computing frequency moments [8]. See [79] for a broad survey.

The online computation of quantile summaries is particularly relevant to this thesis: see [57] for a survey. The rank of a number v given a set A is the number of elements of A which are less than v:

$$r_A(v) = |\{a \in A \mid a < v\}|.$$

The histograms of chapter 7 guarantee "approximation-by-value": given $k \in \mathbb{N}$, an error tolerance $\epsilon \in [0, 1)$, and a stream A with n elements, they return a value v such that for some number v_{true} with rank $r_A(v_{\text{true}}) = k$, $(1 - \epsilon)v_{\text{true}} \leqslant v \leqslant (1 + \epsilon)v_{\text{true}}$. In contrast, most algorithms in the literature guarantee "approximation-by-rank": given $k \in \mathbb{N}$, an error tolerance $\epsilon \in [0, 1)$, and a stream A with n elements, they return a value v such that $|r_A(v) - k| \leqslant \epsilon n$.

Let A_1, A_2, ..., A_n be a family of sets with medians v_1, v_2, ..., v_n respectively. Say we are interested in computing the median of these medians: $V = \text{median}\{v_1, v_2, \ldots, v_n\}$.

Consider a sequence of numbers $\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n$, each of which has rank close to that of the median:

$$\left| r_{A_i}(\hat{v}_i) - \frac{|A_i|}{2} \right| \leqslant \frac{\epsilon |A_i|}{2}$$

Then the values V and $\hat{V} = \text{median}\{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n\}$ may be arbitrarily different: i.e. while existing algorithms typically have better space utilization than our representation, the approximation-by-rank guarantee is not compositional.

Most research on streaming algorithms focusses on computing specific functions with highly constrained space and time budgets: this is orthogonal to our goal in this thesis, which is to provide a general framework to hierarchically express a large class of queries. Our hope is that developments in the area of streaming algorithms will automatically allow larger classes of QREs to be efficiently computed.

Chapter 10

Conclusion

10.1 Summary

In this thesis, we considered the problem of specifying stream transformations. We introduced two programming abstractions, DReX and QREs, and argued that:

1. They are a convenient way to describe queries over streaming data. Queries can be composed in a modular way, and can naturally express structured computation over inherently unstructured data such as logs and event streams.

2. Consistent function expressions can be quickly evaluated, and memory-efficient approximation algorithms can be automatically extracted from some queries which are hard to exactly compute.

3. They have appealing theoretical properties and are expressively robust: expressiveness is closed under function composition, input reversal, and regular look-ahead. The class of functions expressible using DReX and QREs coincides with the class of "regular" stream transformations.

10.2 Future Work

Applications. QREs have already been extended to describe network management policies [97]. They use QREs (with a few extensions similar to the "map-collect" operator which we will describe later in this section) to identify patterns such as super-spreaders (nodes which contact a large number of servers during a time period), estimate the entropy of network traffic, and detect malicious traffic patterns such as SYN-flood attacks. There is an opportunity to explore various other domain-specific extensions of QREs, such as in the analysis of data from medical devices such as pacemakers.

Jyotirmoy Deshmukh recently pointed out that the single-pass evaluation of quantitative functions is very similar to certain monitoring problems in the area of cyber-physical systems. Consider an engineer developing an automotive engine controller. Say they wish to assert that the engine speed never exceeds 5000 rpm. There are a variety of quantitative extensions to

temporal logic in which such properties could be specified, such as metric temporal logic [52] and signal temporal logic [48]. In these applications, it is natural to speak of numerical robustness measures—*how well* does the system satisfy the property—rather than just a boolean YES / NO answer. For example, an engine whose maximum speed is 4300 rpm satisfies the $\mathsf{G}(\text{speed} \leqslant 5000)$ specification more comfortably (with 500 rpm to spare) than an engine which occasionally reaches 4970 rpm. The logics under consideration can be naturally extended to map traces to real numbers, which indicate how much distortion the input signal can handle before violating the property. In a large class of real-world settings, because the system under consideration is complex, engineers often resort to testing rather than exhaustive verification. The technical problem is to compute the robustness score given the specification and simulation trace. While there is some work on online monitoring for these robustness measures [47], the general case of unrestricted STL formulas is still open. When extended with robustness measures, STL and MTL are both similar in spirit to QREs: can the techniques we developed for single-pass QRE evaluation in chapter 6 be extended to the quantitative monitoring of cyber-physical systems?

Another important direction of future work is to understand the usability of stream processing engines. The recent paper of Vaziri et al [92], where they present streams as spreadsheets and streaming computations as spreadsheet functions, demonstrates the great potential impact of research in this area. Debugging tools such as RegexBuddy (`http://www.regexbuddy.com/`) might also be very useful in the context of DReX and QREs.

The computational triangle: evaluation, pre-conditions, expression inference. A function expression e maps input streams w to outputs d. There are three natural computational problems in this setting:

1. Given e and w, compute $\mathrm{d} = [\![e]\!](w)$. This is the expression evaluation problem, and we have studied this problem in detail in this thesis.

2. Given e and d, compute an input stream w such that $[\![e]\!](w) = \mathrm{d}$. This is the problem of *pre-image computation*, and has immediate applications in static analysis. For consistent DReX expressions, it can be shown that the pre-image computation problem is PSPACE-complete.

3. Given w and d, compute some expression e such that $[\![e]\!](w) = \mathrm{d}$. This is the *expression inference* problem, and has applications in simplifying end-user programming [61], and in extracting models from black-box systems.

Various generalizations of the pre-image computation problem are also important: for example, compute (some representation of) *all* input streams w such that $\mathrm{d} = [\![e]\!](w)$ satisfies some property $\varphi(\mathrm{d})$. The static analysis of string transformations is very important in computer security [67, 55, 94]. In this setting, the question typically being answered is whether an encoder can ever generate ill-formed output, such as an unescaped backslash character. The SMT-LIB standard has been extended with a theory of strings [29], and the 2016 edition of the SyGuS competition also included a track to learn string transformations

from input-output examples [18]. Another prominent decision problem is that of equivalence-checking: for QREs, depending on the domain, this problem is often undecidable [13]. For DReX expressions this is decidable, but we have been unable to pin down the complexity: we know that the equivalence of SSTs can be determined in PSPACE [10], but we are unable to even show that the equivalence checking problem is NP-hard (either for DReX expressions or for SSTs).

Query optimization. We have some evidence that mechanical query optimization is feasible for QREs [20]: this includes opportunities to eliminate streaming composition by predicate- and projection-pushdowns, and eliminating instances of the combine combinator by careful expression rewriting. There is also a large body of work on equational reasoning for regular expressions, for example [70], and it is conceivable that every regular expression equivalence corresponds to a scheme to obtain equivalent QREs over a sufficiently rich universe of types and operations.

The iteration combinator in our formalisms seems inherently sequential: a potential area for further research seems to be in parallelizing or otherwise optimizing this construct, perhaps by applying ideas from stream fusion [39] and improved algorithms from the area of matching regular expressions [80].

Binary predicates, the `group by` statement, and sliding windows. The main limitation we observed while trying to express the NEXMark and Linear Road benchmarks was the lack of binary predicates, and the lack of a SQL-like `group by` statement. We would ideally like to be express transformations such as e_{inc}, which labels all monotonically increasing sub-sequences of the input stream w. It seems natural to express e_{inc} as:

$$e_{\text{inc}} = \text{chain}(f, \Sigma), \text{ where}$$
$$f = (x, y) \mapsto \begin{cases} 1 & \text{if } x \leqslant y, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

Unfortunately, the binary transformation $(x, y) \mapsto \mathbf{1}(x \leqslant y)$, where $\mathbf{1}$ is the indicator function is inexpressible in our current formalism, and the expression above is not a well-formed QRE. Including binary predicates makes the domain equivalence problem undecidable [42], potentially making consistency checking and streaming evaluation difficult. The lack of binary predicates would be somewhat compensated by the presence of a `group by` statement [20]. In particular, the combinator would be written as:

$$e = \text{map-collect}(f, e, \text{agg}),$$

where $f : \Sigma \to \text{Keys}$ is applied to each incoming element a of the input stream. Depending on the value of the key $k = f(a)$, a sub-stream $w_{\pi k}$ is formed, containing only those elements whose key is k. The inner expression e is independently applied to each sub-stream $w_{\pi k}$, and the values produced by the sub-evaluators are accumulated by the aggregator function agg. The combination of streaming composition and map-collect is very powerful: we are able to

express all of the Linear Road and NEXMark benchmarks, but several technical challenges remain. For example, the immediate semantic question is of determining the domain: is the parent expression defined when $[\![e]\!](w_{\pi \mathtt{k}})$ is defined for all keys $\mathtt{k}$, or is it defined when there exists a key $\mathtt{k}$ such that e is defined for the sub-stream? Both choices lead to interesting unsolved questions regarding regular languages over infinite alphabets.

Another extension we study in [20] is sliding windows. An example query would be to compute the mean trading price of a stock over the last 180 days. While the sliding window idiom is technically already expressible as a native QRE, it typically requires expressions whose size is proportional to the window length, causing a blowup both in consistency checking and evaluation.

First-match, last-match, and other semantic variations. We originally presented DReX expressions e as defining relations, $[\![e]\!] \subseteq \Sigma^* \times \mathbb{D}$, rather than functions $[\![e]\!] : \Sigma^* \to \mathbb{D}$. We then introduced the subclass of consistent expressions, and pointed out that consistent expressions defined (partial) functions $\Sigma^* \to \mathbb{D}$. Another motive in defining consistent expressions was to prevent some hard-to-parse anti-patterns from being easily expressible as DReX expressions. For example, combine$(e, \mathtt{f})$ can be naturally defined as a function whose domain is the intersection of the domains of the sub-expressions e and $\mathtt{f}$. Similarly, the expression e else $\mathtt{f}$ can be alternatively defined as follows:

$$[\![e \text{ else } \mathtt{f}]\!](w) = \begin{cases} [\![e]\!](w) & \text{if } [\![e]\!](w) \neq \bot, \text{ and} \\ [\![\mathtt{f}]\!](w) & \text{otherwise,} \end{cases}$$

evoking the idea of domain complementation ("apply $\mathtt{f}$ if e is undefined"). Matching extended regular expressions (traditional regular expressions with first class complementation) is a difficult problem [81], and the consistency rules conveniently avoid the problems of intersection (in combine$(e, \mathtt{f})$, the sub-expressions have equal domains) and domain complementations (in e else $\mathtt{f}$, e and $\mathtt{f}$ have disjoint domains). One concrete unresolved question is the following: given a functional DReX expression e and an input stream w, what is the complexity of computing $[\![e]\!](w)$? We are already aware of some lower bounds: in this sub-class, the output length $|[\![e]\!](w)|$ is potentially exponential in the expression size $|e|$, but this still leaves open the possibility of efficient output-sensitive algorithms. Also recall that regular string transformations are closed under function composition. If function composition is included as a first-class combinator, then computing $[\![e]\!](\epsilon)$ is PSPACE-complete [14].

There are several operators which, while not increasing expressive power, would be quite useful in practical situations. The restrict combinator is a good example. Given an expression e and a regular expression $\mathtt{r}$, restrict$(e, \mathtt{r})$ is defined as follows:

$$[\![\text{restrict}(e, \mathtt{r})]\!](w) = \begin{cases} [\![e]\!](w) & \text{if } w \in [\![\mathtt{r}]\!], \text{ and} \\ \bot & \text{otherwise.} \end{cases}$$

Other similar extensions include providing ways to specify first-split, greedy-split, and other disambiguation semantics that are commonly implemented by string processing tools such as sed, AWK, and Perl. In all these cases, the complexity of evaluation is unclear.

Generalizing the input domain. Throughout this thesis, we have used the words "stream" and "string" interchangeably. The idea is that we always only see a finite prefix of even (potentially) infinite streams, and it is sufficient if the query language is able to specify properties of these finite prefixes. Perhaps a better view of streams is as ω-strings: there is already some work on extending SSTs to this case [16]. Obtaining an expressively equivalent combinator calculus for this class is an interesting open problem.

A related problem is in extending the idea of combinator-based programming to the input domain of trees. We are particularly interested in trees as they naturally model data stored in XML documents. DReX and QREs are expressively equivalent to the operational models of SSTs and SSTTs respectively: over the input domain of trees, the appropriate machines to target for expressive equivalence appears to be *streaming tree transducers* [12].

Bibliography

[1] Apache Storm. Available at `http://storm.apache.org/`.

[2] PipelineDB. Available at `https://www.pipelinedb.com/`.

[3] Daniel Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley Zdonik. The design of the Borealis stream processing engine. In *2nd Biennial Conference on Innovative Data Systems Research*, CIDR 2005, pages 277–289, 2005.

[4] Daniel Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Phillip Rümmer, and Jari Stenman. String constraints for verification. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, CAV 2014, pages 150–166. Springer, 2014.

[6] Alfred Aho, John Hopcroft, and Jeffrey Ullman. A general theory of translation. *Mathematical Systems Theory*, 3(3):193–221, 1969.

[7] Shaull Almagor, Udi Boker, and Orna Kupferman. What's decidable about weighted automata? In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, ATVA 2011, pages 482–491. Springer, 2011.

[8] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the 28th ACM Symposium on Theory of Computing*, STOC 1996, pages 20–29. ACM, 1996.

[9] Rajeev Alur and Pavol Černý. Expressiveness of streaming string transducers. In Kamal Lodaya and Meena Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[10] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2011, pages 599–610. ACM, 2011.

[11] Rajeev Alur and Loris D'Antoni. Streaming tree transducers. *CoRR*, abs/1104.2599, 2011. Preprint of [12].

[12] Rajeev Alur and Loris D'Antoni. Streaming tree transducers. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming, Part II*, ICALP 2012, pages 42–53. Springer, 2012.

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

[14] Rajeev Alur, Loris D'Antoni, and Mukund Raghothaman. DReX: A declarative language for efficiently evaluating regular string transformations. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015, pages 125–137. ACM, 2015.

[15] Rajeev Alur and Jyotirmoy Deshmukh. Nondeterministic streaming string transducers. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, *Proceedings of the 38th International Colloquium on Automata, Languages, and Programming, Part II*, ICALP 2011, pages 1–20. Springer, 2011.

[16] Rajeev Alur, Emmanuel Filiot, and Ashutosh Trivedi. Regular transformations of infinite strings. In *Proceedings of the 27th ACM/IEEE Symposium on Logic in Computer Science*, LICS 2012, pages 65–74. IEEE Computer Society, 2012.

[17] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular programming for quantitative properties of data streams. In Peter Thiemann, editor, *Programming Languages and Systems: Proceedings of the 25th European Symposium on Programming*, ESOP 2016, pages 15–40. Springer, 2016.

[18] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. SyGuS syntax for SyGuS-COMP'16. Technical report, University of Pennsylvania, 2016. Available at `http://sygus.seas.upenn.edu/files/SyGuS-Syntax-SyGuSCOMP'16.pdf`.

[19] Rajeev Alur, Adam Freilich, and Mukund Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science*, CSL-LICS 2014, pages 9:1–9:10. ACM, 2014.

[20] Rajeev Alur, Sanjeev Khanna, Zachary Ives, Konstantinos Mamouras, and Mukund Raghothaman. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In submission, 2016.

[21] Rajeev Alur and Mukund Raghothaman. DecisionÂ problemsÂ forÂ additiveÂ regularÂ functions. In Fedor Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg, editors, *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming, Part II*, ICALP 2013, pages 37–48. Springer, 2013.

[22] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[23] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[24] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A language for continuous queries over streams and relations. In *Proceedings of the 9th International Workshop on Database Programming Languages*, DBPL 2004, pages 1–19. Springer, 2004.

[25] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A stream data management benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases*, VLDB 2004, pages 480–491. VLDB Endowment, 2004.

[26] Christel Baier, Marcus Größer, and Frank Ciesinski. *Model Checking Linear-Time Properties ofÂ Probabilistic Systems*, pages 519–570. In [49].

[27] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy*, SOSP 2008, pages 387–401, 2008.

[28] Roger Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, CIDR 2007, pages 363–374, 2007.

[29] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `http://www.SMT-LIB.org`.

[30] Mikołaj Bojańczyk. Factorization forests. In Volker Diekert and Dirk Nowotka, editors, *Proceedings of the 13th International Conference on Developments in Language Theory*, DLT 2009, pages 1–17. Springer, 2009.

[31] Mikołaj Bojańczyk. Transducers with origin information. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming, Part II*, ICALP 2014, pages 26–37. Springer, 2014.

[32] Matko Botinčan and Domagoj Babić. Sigma*: Symbolic learning of input-output specifications. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2013, pages 443–456. ACM, 2013.

[33] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2007, pages 1100–1102. ACM, 2007.

[34] Richard Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[35] Jianjun Chen, David DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2000, pages 379–390. ACM, 2000.

[36] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *1st Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware*, 2016.

[37] Michal Chytil and Vojtěch Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In Arto Salomaa and Magnus Steinby, editors, *Proceedings of the 4th Colloquium on Automata, Languages and Programming*. Springer, 1977.

[38] Bruno Courcelle. Monadic second-order definable graph transductions. In Jean-Claude Raoult, editor, *Proceedings of the 17th Colloquium on Trees in Algebra and Programming*, volume 581, pages 124–144. Springer, 1992.

[39] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2007, pages 315–326. ACM, 2007.

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

[41] Loris D'Antoni and Rajeev Alur. Symbolic visibly pushdown automata. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, CAV 2014, pages 209–225. Springer, 2014.

[42] Loris D'Antoni and Margus Veanes. Equivalence of extended symbolic finite transducers. In Natasha Sharyngina and Helmut Veith, editors, *Computer Aided Verification*, CAV 2013, pages 624–639. Springer, 2013.

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

[44] Joel de Guzman and Dan Nuffer. The Spirit parser library: Inline parsing in C++. *Dr. Dobb's Journal*, September 2003. Accessible at `http://www.drdobbs.com/cpp/the-spirit-parser-library-inline-parsing/184401692`.

[45] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In Yannis Ioannidis, Marc Scholl, Joachim Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Proceedings of the 10th International Conference on Extending Database Technology*, EDBT 2006, pages 627–644. Springer, 2006.

[46] Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker White. Cayuga: A general purpose event monitoring system. In *3rd Biennial Conference on Innovative Data Systems Research*, CIDR 2007, pages 412–422, 2007.

[47] Jyotirmoy Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit Seshia. Robust online monitoring of signal temporal logic. In Ezio Bartocci and Rupak Majumdar, editors, *6th International Conference on Runtime Verification*, RV 2015, pages 55–70. Springer, 2015.

[48] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *8th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS 2010, pages 92–106. Springer, 2010.

[49] Manfred Droste, Werner Kuich, and Heiko Vogler, editors. *Handbook of Weighted Automata*. Monographs in Theoretical Computer Science. Springer, 2009.

[50] Calvin Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, 98(1):21–51, 1961.

[51] Joost Engelfriet and Hendrik Jan Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, April 2001.

[52] Georgios Fainekos and George Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.

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

[54] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2004, pages 111–122. ACM, 2004.

[55] Vijay Ganesh, Adam Kieżun, Shay Artzi, Philip Guo, Pieter Hooimeijer, and Michael Ernst. HAMPI: A string solver for testing, analysis and vulnerability detection. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 1–19. Springer, 2011.

[56] Bjørn Bugge Grathwohl, Fritz Henglein, Ulrik Terp Rasmussen, Kristoffer Aalund Søholm, and Sebastian Paaske Tørholm. Kleenex: Compiling nondeterministic transducers to deterministic streaming transducers. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 284–297. ACM, 2016.

[57] Michael Greenwald and Sanjeev Khanna. *Data Stream Management: Processing High-Speed Data Streams*, chapter Quantiles and Equi-depth Histograms over Streams, pages 45–86. Springer, 2016.

[58] Thomas Griffiths. The unsolvability of the equivalence problem for λ-free nondeterministic generalized machines. *Journal of the ACM*, 15(3):409–413, July 1968.

[59] Ralph Griswold, James Poage, and Ivan Polansky. *The SNOBOL 4 programming language*. Prentice-Hall, 1971.

[60] The Stanford STREAM Group. Stream query repository. Accessible at `http://infolab.stanford.edu/stream/sqr/`, 2002.

[61] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–330. ACM, 2011.

[62] Sumit Gulwani, William Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[63] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. In *Bulletin of the Technical Committee on Data Engineering*, volume 18, pages 3–18. IEEE Computer Society, 1995.

[64] Eitan Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. In *Proceedings of 21st Annual Symposium on Foundations of Computer Science*, pages 83–85. IEEE Computer Society, 1980.

[65] Philip Hazel. PCRE: Perl compatible regular expressions. Available at `http://www.pcre.org/`, 1997.

[66] Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7:1–7:11, 2013.

[67] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX Conference on Security*, SEC 2011. USENIX Association, 2011.

[68] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View maintenance issues for the chronicle data model (extended abstract). In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS 1995, pages 113–124. ACM, 1995.

[69] Kevin Knight and Jonathan May. *Applications of Weighted Automata in Natural Language Processing*, pages 571–596. In [49].

[70] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.

[71] Ilan Kremer, Noam Nisan, and Dana Ron. On randomized one-round communication complexity. *Computational Complexity*, 8(1):21–49, 1999.

[72] Daniel Krob. The equality problem for rational series with multiplicities in the tropical semiring is undecidable. In Werner Kuich, editor, *Automata, Languages and Programming*, ICALP 1992, pages 101–112. Springer, 1992.

[73] Manfred Kufleitner. A proof of the factorization forest theorem. *CoRR*, abs/0710.5130, 2007.

[74] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht, 2001.

[75] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2005, pages 311–322. ACM, 2005.

[76] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, CAV 2014, pages 646–662. Springer, 2014.

[77] Gennady Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR-Sbornik*, 32(2):129–198, 1977.

[78] James Munro and Michael Paterson. Selection and sorting with limited storage. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS 1978, pages 253–258. IEEE Computer Society, 1978.

[79] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Foundations and Trends in Theoretical Computer Science. Now Publishers Inc, 2005.

[80] Gene Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, April 1992.

[81] Grigore Roşu. An effective algorithm for the membership problem for extended regular expressions. In Helmut Seidl, editor, *Foundations of Software Science and Computational Structures: 10th International Conference*, FOSSACS 2007, pages 332–345. Springer, 2007.

[82] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*, SOSP 2010, pages 513–528, 2010.

[83] Imre Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65–94, 1990.

[84] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5(8):740–751, 2012.

[85] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3 edition, 2012.

[86] Richard Stearns and Harry Hunt. On the equivalence and containment problems for unambiguous regular expressions, grammars, and automata. In *22nd Annual Symposium on Foundations of Computer Science*, SFCS 1981, pages 74–81, 1981.

[87] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD 1992, pages 321–330. ACM, 1992.

[88] Attributed to Abigail. Reduction of 3-CNF-SAT to Perl regular expression matching. Available at `http://perl.plover.com/NPC/NPC-3SAT.html`.

[89] Boris Trakhtenbrot. Finite automata and the logic of one-place predicates. *Siberian Mathematical Journal*, 3:101–131, 1962.

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

[91] Peter Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[92] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In Richard Jones, editor, *Proceedings of the 28th European Conference on Object-Oriented Programming*, ECOOP 2014, pages 360–384. Springer, 2014.

[93] Margus Veanes and Nikolaj Bjørner. Symbolic automata: The toolkit. In Cormac Flanagan and Barbara König, editors, *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2012, pages 472–477. Springer, 2012.

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

[95] Margus Veanes, Todd Mytkowicz, David Molnar, and Benjamin Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, POPL 2015, pages 139–152. ACM, 2015.

[96] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2006, pages 407–418. ACM, 2006.

[97] Yifei Yuan. *High-level abstractions for programming network policies*. PhD thesis, University of Pennsylvania, 2016.

[98] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 2012, pages 15–28. USENIX Association, 2012.

[99] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP 2013, pages 423–438. ACM, 2013.

[100] Yunhi Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124. ACM, 2013.