Regular Combinators for String Transformations *

Rajeev Alur Adam Freilich Mukund Raghothaman
University of Pennsylvania

Abstract

We focus on (partial) functions that map input strings to a monoid such as the set of integers with addition and the set of output strings with concatenation. The notion of regularity for such functions has been defined using two-way finite-state transducers, (one-way) cost register automata, and MSO-definable graph transformations. In this paper, we give an algebraic and machine-independent characterization of this class analogous to the definition of regular languages by regular expressions. When the monoid is commutative, we prove that every regular function can be constructed from constant functions using the combinators of choice, split sum, and iterated sum, that are analogs of union, concatenation, and Kleene-*, respectively, but enforce unique (or unambiguous) parsing. Our main result is for the general case of non-commutative monoids, which is of particular interest for capturing regular string-to-string transformations for document processing. We prove that the following additional combinators suffice for constructing all regular functions: (1) the left-additive versions of split sum and iterated sum, which allow transformations such as string reversal; (2) sum of functions, which allows transformations such as copying of strings; and (3) function composition, or alternatively, a new concept of chained sum, which allows output values from adjacent blocks to mix.

Categories and Subject Descriptors F.1.1 [*Computation by Abstract Devices*]: Models of Computation—Automata

1. Introduction

To study string transformations, given the success of finite-state automata and the associated theory of regular languages, a natural starting point is the model of finite-state transducers. A finite-state transducer emits output symbols at every step, and given an input string, the corresponding output string is the concatenation of all the output symbols emitted by the machine during its execution. Such transducers have been studied since the 1960s, and it has been known that the transducers have very different properties compared to the acceptors: *two-way* transducers are strictly more expressive than their one-way counter-parts, and the post-image of a regular language under a two-way transducer need not be a regular language [1]. For the class of transformations computed by two-way transducers, [9] establishes closure under composition, [16] proves decidability of functional equivalence, and [13] shows that their expressiveness coincides with MSO-definable string-to-string transformations of [11]. As a result, [13] justifiably dubbed this class as *regular* string transformations. Recently, an alternative characterization using one-way machines was found for this class: *streaming string transducers* [2] (and their more general and abstract counterpart of *cost register automata* [5]) process the input string in a single left-to-right pass, but use multiple write-only registers to store partially computed output chunks that are updated and combined to compute the final answer.

There has been a resurgent interest in such transducers in the formal methods community with applications to learning of string transformations from examples [15], sanitization of web addresses [18], and algorithmic verification of list-processing programs [3]. In the context of these applications, we wish to focus on regular transformations, rather than the subclass of classical one-way transducers, since the gap includes many natural transformations such as string reversal and swapping of substrings, and since one-way transducers are not closed under basic operations such as choice.

For our formal study, we focus on *cost functions*, that is, (partial) functions that map strings over a finite alphabet to values from a monoid $(\mathbb{D}, \otimes, 0)$. While the set of output strings with concatenation is a typical example of such a monoid, cost functions can also associate numerical values (or rewards) with sequences of events, with possible application to *quantitative* analysis of systems [8] (it is worth pointing out that the notion of regular cost functions proposed by Colcombet is quite distinct from ours [10]). An example of such a numerical domain is the set of integers with addition. In the case of a *commutative* monoid, regular functions have a simpler structure, and correspond to *unambiguous weighted automata* (note that weighted automata are generally defined over a semiring, and are very extensively studied—see [12] for a survey, but with no results directly relevant to our purpose).

A classical result in automata theory characterizes regular languages using *regular expressions*: regular languages are exactly the sets that can be inductively generated from base languages (empty set, empty string, and alphabet symbols) using the operations of union, concatenation, and Kleene-*. Regular expressions provide a robust foundation for specifying regular patterns in a *declarative* manner, and are widely used in practical applications. The goal of this paper is to identify the appropriate base functions and combinators over cost functions for an analogous algebraic and machine-independent characterization of regularity.

We begin our study by defining base functions and combinators that are the analogs of the classical operations used in regular expressions. The base function L/d maps strings σ in the base language L to the constant value d, and is undefined when $\sigma \notin L$. Given cost functions f and g, the *conditional choice* combinator $f \triangleright g$ maps an input string σ to $f(\sigma)$, if this value is defined,

* This research was partially supported by NSF Expeditions in Computing grant CCF 1138996. The full version of this paper may be found at http://arxiv.org/abs/1402.3021.

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.

http://dx.doi.org/10.1145/2603088.2603151

and to $g(\sigma)$ otherwise; the *split sum* combinator $f \bullet|\otimes g$ maps an input string σ to $f(\sigma_1) \otimes g(\sigma_2)$ if the string σ can be split *uniquely* into two parts σ_1 and σ_2 such that both $f(\sigma_1)$ and $g(\sigma_2)$ are defined, and is undefined otherwise; and the *iterated sum* $f^{*|\otimes}$ is defined so that if the input string σ can be split uniquely such that $\sigma = \sigma_1\sigma_2\ldots\sigma_k$ and each $f(\sigma_i)$ is defined, then $f^{*|\otimes}(\sigma)$ is $f(\sigma_1) \otimes f(\sigma_2) \otimes \cdots \otimes f(\sigma_k)$, and is undefined otherwise. The combinators conditional choice, split sum, and iterated sum are the natural analogs of the operations of union, concatenation, and Kleene-* over languages, respectively. The uniqueness restrictions ensure that the input string is parsed in an unambiguous manner while computing its cost, and thus, the result of combining two (partial) functions remains a (partial) *function*.

Our first result is that when the operation $\otimes$ is commutative, regular functions are exactly the functions that can be inductively generated from base functions using the combinators of conditional choice, split sum, and iterated sum. The proof is fairly straightforward, and builds on the known properties of cost register automata, their connection to unambiguous weighted automata in the case of commutative monoids, and the classical translation from automata to regular expressions.

When the operation $\otimes$ is not commutative, which is the case when the output values are strings themselves and $\otimes$ corresponds to string concatenation, we need additional combinators to capture regularity. First, in the non-commutative case, it is natural to introduce symmetric *left-additive* versions of split sum and iterated sum. Given cost functions f and g, the *left-split sum* $f \otimes|\bullet\ g$ maps an input string σ to $g(\sigma_2) \otimes f(\sigma_1)$ if the string σ can be split uniquely into two parts σ_1 and σ_2 such that both $f(\sigma_1)$ and $g(\sigma_2)$ are defined. The *left-iterated sum* is defined analogously, and in particular, the transformation that maps an input string to its *reverse* is simply the left-iterated sum of the function that maps each symbol to itself. It can be shown that regular functions are closed under these left-additive combinators.

The *sum* $f \otimes g$ of two functions f and g maps a string σ to $f(\sigma) \otimes g(\sigma)$. Though the sum combinator is not necessary for completeness in the commutative case, it is natural for cost functions. For example, the *string copy* function that maps an input string σ to the output $\sigma\sigma$ is simply the sum of the identity function over strings with itself. It is already known that regular functions are closed under sum [5, 13].

All the combinators described above first split the input string into disjoint patches, and then map these patches into disjoint substrings of the output. There are, however, regular functions where this property does not hold: consider the string transformation *shuffle* that maps a string of the form $a^{m_1}ba^{m_2}b\ldots a^{m_k}b$ to $a^{m_2}b^{m_1}a^{m_3}b^{m_2}\ldots a^{m_k}b^{m_{k-1}}$ (see figure 2.1a). This function is definable using cost register automata, but we conjecture that it cannot be constructed using the combinators discussed so far. We introduce a new form of iterated sum: given a language L and a cost function f, if the input string σ can be split uniquely so that $\sigma = \sigma_1\sigma_2\ldots\sigma_k$ with each $\sigma_i \in L$, then the *chained sum* $(f, L)^{*|\otimes}$ of σ is $f(\sigma_1\sigma_2) \otimes f(\sigma_2\sigma_3) \otimes \cdots \otimes f(\sigma_{k-1}\sigma_k)$. In other words, the input is (uniquely) divided into substrings belonging to the language L, but instead of summing the values of f on each of these substrings, we sum the values of f applied to blocks of adjacent substrings in a chained fashion. The string-transformation *shuffle* now is simply chained sum where L equals the regular language a^*b, and f maps a^iba^jb to a^jb^i (such a function f can itself be constructed using iterated sum and left-split sum). It turns out that this new combinator can also be defined if we allow *function composition*: if f is a function that maps strings to strings and g is a cost function, then the composed function $g \circ f$ maps an input string σ to $g(f(\sigma))$. Such rewriting is a natural operation, and regular functions are closed under composition [9].

The main technical result of the paper is that every regular function can be inductively generated from base functions using the combinators of conditional choice, sum, split sum, either chained sum or function composition, and their left additive versions. The proof in section 5 constructs the desired expressions corresponding to executions of cost register automata. Such automata have multiple registers, and at each step the registers are updated using *copyless* (or single-use) assignments. Register values can flow into one another in a complex manner, and the proof relies on understanding the structure of compositions of *shapes* that capture these value-flows. The proof provides insights into the power of the chained sum operation, and also offers an alternative justification for the copyless restriction for register updates in the machine-based characterization of regular functions.

2. Function Combinators

Let Σ be a finite alphabet, and $(\mathbb{D}, \otimes, 0)$ be a monoid. Two natural monoids of interest are those of the integers $(\mathbb{Z}, +, 0)$ under addition, and of strings $(\Gamma^*, \cdot, \epsilon)$ over some output alphabet Γ under concatenation. By convention, we treat $\perp$ as the undefined value, and express partial functions $f : A \to B$ as total functions $f : A \to B_\perp$, where $B_\perp = B \cup \{\perp\}$. We extend the semantics of the monoid $\mathbb{D}$ to $\mathbb{D}_\perp$ by defining $d \otimes \perp = \perp \otimes d = \perp$, for all $d \in \mathbb{D}$. A *cost function* is a function $\Sigma^* \to \mathbb{D}_\perp$.

2.1 Base functions

For each language $L \subseteq \Sigma^*$ and $d \in \mathbb{D}$, we define the *constant function* $L/d : \Sigma^* \to \mathbb{D}_\perp$ as

$$L/d\ (\sigma) = \begin{cases} d & \text{if } \sigma \in L, \text{ and} \\ \perp & \text{otherwise.} \end{cases}$$

The *everywhere-undefined function* $\perp : \Sigma^* \to \mathbb{D}_\perp$ is defined as the constant function $\emptyset/0$, and therefore, $\perp(\sigma) = \perp$, for all strings σ.

Example 1. Let $\Sigma = \{a, b\}$ in the following examples. Then, the constant function $a/a : \Sigma^* \to \Sigma^*$maps a to itself, and is undefined on all other strings. We will often be interested in functions of the form a/a: when the intent is clear, we will use the shorthand a.

By *base functions*, we refer to the class of functions L/d, where L is a regular expression.

2.2 Choice operators

Let $f, g : \Sigma^* \to \mathbb{D}_\perp$ be two functions. We then define the *conditional choice* $f \rhd g$ as

$$f \rhd g(\sigma) = \begin{cases} f(\sigma) & \text{if } f(\sigma) \neq \perp, \text{ and} \\ g(\sigma) & \text{otherwise.} \end{cases}$$

The *unambiguous choice* $f \uplus g$ is defined as

$$f \uplus g(\sigma) = \begin{cases} f(\sigma) & \text{if } f(\sigma) \neq \perp \text{ and } g(\sigma) = \perp, \\ g(\sigma) & \text{if } f(\sigma) = \perp \text{ and } g(\sigma) \neq \perp, \text{ and} \\ \perp & \text{otherwise.} \end{cases}$$

Example 2. The indicator function $\mathbf{1}_L : \Sigma^* \to \mathbb{Z}$ is defined as $\mathbf{1}_L(\sigma) = 1$ if $\sigma \in L$ and $\mathbf{1}_L(\sigma) = 0$ otherwise. This function can be expressed using the conditional choice operator as $L/1 \rhd \Sigma^*/0$, and using the unambiguous choice operator as $L/1 \uplus (\Sigma^* \setminus L)/0$. Note that the representation using the unambiguous union operator relies on regular expressions being closed under set difference.

Observe that $\rhd$ and $\uplus$ are the analogs of union in regular expressions, with the important difference being that $\rhd$ is non-commutative, and $\uplus$ is non-associative[1].

2.3 Sum operators

The *sum* $f \otimes g$ of two functions $f, g : \Sigma^* \to \mathbb{D}_\perp$ is defined as $f \otimes g(\sigma) = f(\sigma) \otimes g(\sigma)$. If there exist unique strings σ_1 and σ_2 such that $\sigma = \sigma_1\sigma_2$, and $f(\sigma_1)$ and $g(\sigma_2)$ are both defined, then the *split sum* $f \bullet|\otimes g(\sigma) = f(\sigma_1) \otimes g(\sigma_2)$. Otherwise, $f \bullet|\otimes g(\sigma) = \perp$. Over non-commutative monoids, this may be different from the *left-split sum* $f \otimes|\bullet g$: if there exist unique strings σ_1 and σ_2, such that $\sigma = \sigma_1\sigma_2$, and $f(\sigma_1)$ and $g(\sigma_2)$ are both defined, then $f \otimes|\bullet g(\sigma) = g(\sigma_2) \otimes f(\sigma_1)$. Otherwise, $f \otimes|\bullet g(\sigma) = \perp$.

The split sum operators $\bullet|\otimes$ and $\otimes|\bullet$ are similar to the concatenation operator of traditional regular expressions. The key difference between the split sum operators and the concatenation operator of traditional regular expressions is the requirement of a unique split. Observe that if σ could be split in two different ways, $\sigma = \sigma_1\sigma_2 = \sigma_3\sigma_4$, such that all of $f(\sigma_1)$, $f(\sigma_3)$, $g(\sigma_2)$, and $g(\sigma_4)$ are defined, then it may not necessarily be the case that $f(\sigma_1) \otimes g(\sigma_2) = f(\sigma_3) \otimes g(\sigma_4)$, and this would lead to difficulties in defining $f \bullet|\otimes g$ as a function. The easiest way to avoid this difficulty is to define $f \bullet|\otimes g(\sigma)$ only when the split $\sigma = \sigma_1\sigma_2$ into inputs for f and g is unique.

2.4 Iteration

The *iterated sum* $f^{*|\otimes}$ of a cost function is defined as follows. If there exist unique strings $\sigma_1, \sigma_2, \ldots, \sigma_k$ such that $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$ and $f(\sigma_i)$ is defined for each σ_i, then

$$f^{*|\otimes}(\sigma) = f(\sigma_1) \otimes f(\sigma_2) \otimes \cdots \otimes f(\sigma_k).$$

Otherwise, $f^{*|\otimes}(\sigma) = \perp$. The *left-iterated sum* $f^{\otimes|*}$ is defined similarly: if there exist unique strings $\sigma_1, \sigma_2, \ldots, \sigma_k$ such that $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$ and $f(\sigma_i)$ is defined for each σ_i, then

$$f^{\otimes|*}(\sigma) = f(\sigma_k) \otimes f(\sigma_{k-1}) \otimes \cdots \otimes f(\sigma_1).$$

Otherwise, $f^{\otimes|*}(\sigma) = \perp$. The *reverse combinator* f^{rev} is defined as $f^{rev}(\sigma) = f(\sigma^{rev})$, where σ^{rev} is the string reverse of σ. Observe that the left-iterated sum and reverse combinators are interesting in the case of non-commutative monoids, such as string concatenation.

Example 3. The function $|\cdot|_a : \Sigma^* \to \mathbb{Z}$ counts the number of a-s in the input string. This is represented by the function expression $(a/1 \rhd b/0)^{*|\otimes}$. The identity function $id : \Sigma^* \to \Sigma^*$ is given by the function expression $(a \rhd b)^{*|\otimes}$. The function *copy* which maps an input σ to $\sigma\sigma$ is then given by the expression $id \otimes id$: note that the image of a regular language of input strings Σ^* may not be a regular language. On the other hand, the expression $(a \rhd b)^{\otimes|*}$ is the function which reverses its input: $(a \rhd b)^{\otimes|*}(\sigma) = \sigma^{rev}$ for all σ. This is also equivalent to the expression id^{rev}.

Example 4. Consider the situation of a customer who frequents a coffee shop. Every 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 $\Sigma = \{C, S, \#\}$ denoting respectively the purchase of a cup of coffee, completion of the survey, and the passage of a calendar month. Then, the function expression

$$m = (C/2^{*|\otimes}) \rhd ((C/1^{*|\otimes}) \bullet|\otimes S/0 \bullet|\otimes (C/1 \rhd S/0)^{*|\otimes})$$

maps the purchases of a month to the customer's debt. The first sub-expression — $C/2^{*|\otimes}$ — computes the amount provided no survey is filled out and the second sub-expression — $(C/1^{*|\otimes}) \bullet|\otimes S/0 \bullet|\otimes (C/1 \rhd S/0)^{*|\otimes}$ — is defined provided at least one survey is filled out, and in that case, charges \$1 for each cup. The expression

$$coffee = (m \bullet|\otimes \#/0)^{*|\otimes} \bullet|\otimes m$$

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

Example 5. Let $\Sigma = \{a, b, \#\}$, and consider the function *swap* which maps strings of the form $\sigma\#\tau$ where $\sigma, \tau \in \{a, b\}^*$ to $\tau\#\sigma$. Such a function could be used to transform names from the first-name-last-name format to the last-name-first-name format. We can write

$$swap = (\{a, b\}^*\#/\epsilon \bullet|\otimes (a \rhd b)^{*|\otimes}) \otimes (\Sigma^*/\#) \otimes ((a \rhd b)^{*|\otimes} \bullet|\otimes \#\{a, b\}^*/\epsilon).$$

The first subexpression skips the first part of the string — $\{a, b\}^*\#/\epsilon$ — and echoes the second part — $(a \rhd b)^{*|\otimes}$. The second subexpression $\Sigma^*/\#$ inserts the $\#$ in the middle. The third subexpression is similar to the first, echoing the first part of the string and skipping the rest.

Example 6. With $\Sigma = \{a, b, \#\}$, consider the function *strip* which map strings of the form $\sigma_1\#\sigma_2\#\ldots\#\sigma_n$ where $\sigma_i \in \{a, b\}^*$ for each i to $\sigma_1\#\sigma_2\#\ldots\#\sigma_{n-1}$. This function could be used, for example, to locate the directory of a file (such as "`/home`") given its full path (such as "`/home/file.cpp`"). We can write

$$strip = id \bullet|\otimes \#\{a, b\}^*/\epsilon\ .$$

2.5 Chained sum

Let $L \subseteq \Sigma^*$ be a language, and f be a cost function over Σ^*. If there exists a unique decomposition $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$ such that $k \geq 2$ and for each i, $\sigma_i \in L$, then the *chained sum*

$$(f, L)^{*|\otimes}(\sigma) = f(\sigma_1\sigma_2) \otimes f(\sigma_2\sigma_3) \otimes \cdots \otimes f(\sigma_{k-1}\sigma_k).$$

Otherwise, $(f, L)^{*|\otimes}(\sigma) = \perp$. Similarly, if there exist unique strings $\sigma_1, \sigma_2, \ldots, \sigma_k$ such that $k \geq 2$ and for all i, $\sigma_i \in L$, then the *left-chained sum*

$$(f, L)^{\otimes|*}(\sigma) = f(\sigma_{k-1}\sigma_k) \otimes f(\sigma_{k-2}\sigma_{k-1}) \otimes \cdots \otimes f(\sigma_1\sigma_2).$$

Otherwise, $(f, L)^{\otimes|*}(\sigma) = \perp$.

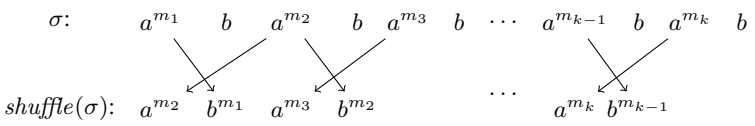
Example 7. Let $\Sigma = \{a, b\}$ and let $shuffle : \Sigma^* \to \Sigma^*$ be the following function: for $\sigma = a^{m_1}ba^{m_2}b\ldots a^{m_k}b$, with $k \geq 2$, $shuffle(\sigma) = a^{m_2}b^{m_1}a^{m_3}b^{m_2}\ldots a^{m_k}b^{m_{k-1}}$, and for all other σ, $shuffle(\sigma) = \perp$. See figure 2.1a.

We first divide σ into patches P_i, each of the form a^*b. Similarly the output may also be divided into patches, P'_i. Each input patch P_i should be scanned twice, first to produce the a-s to produce P'_{i-1}, and then again to produce the b-s in P'_i. Let $L = a^*b$ be the language of these patches. It follows that $shuffle = (f, L)^{*|\otimes}$, where

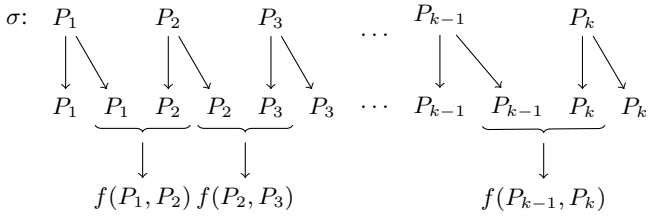
$$f = (a/b^{*|\otimes} \bullet|\otimes b/\epsilon) \otimes|\bullet (a/a^{*|\otimes} \bullet|\otimes b/\epsilon).$$

The motivation behind the chained sum is two-fold: first, we believe that *shuffle* is inexpressible using the remaining operators, and second, the operation naturally emerges as an idiom during the proof of theorem 15.

[1] Observe that $f \uplus (g \uplus h)(\sigma) = f(\sigma)$ and $(f \uplus g) \uplus h(\sigma) = h(\sigma)$ when all of $f(\sigma)$, $g(\sigma)$, and $h(\sigma)$ are defined.



(a) Definition of $shuffle(\sigma)$.



(b) Each patch P_i is a string of the form a^*b.

Figure 2.1: Defining and expressing $shuffle(\sigma)$ using function combinators.

2.6 Function composition

Let $f : \Sigma^* \to \Gamma_\perp^*$ and $g : \Gamma^* \to \mathbb{D}$ be two cost functions. The *composition* $g \circ f$ is defined as $g \circ f(\sigma) = g(f(\sigma))$, if $f(\sigma)$ and $g(f(\sigma))$ are defined, and $g \circ f(\sigma) = \perp$ otherwise.

Example 8. Composition is an alternative to chained sum for expressive completeness — in this example, we use composition to express *shuffle*. Let $copy_L = (a^{*|\otimes} \bullet|\otimes b) \otimes (a^{*|\otimes} \bullet|\otimes b)$ be the function which accepts strings from L and repeats them twice. The first step of the transformation is therefore the expression $copy_L^{*|\otimes}$. We then drop the first copy of P_1 and the last copy of P_k — this is achieved by the expression $drop = L/\epsilon \bullet|\otimes id \bullet|\otimes L/\epsilon$. The function $ensurelen = id \otimes \Sigma^+/\epsilon$ echoes its input, but also ensures that the input string contains at least two patches. The final step is to specify the function f which examines pairs of adjacent patches, and first echoes the a-s from the second patch, and then transforms the a-s from the first patch into b-s. $f = (a/b^{*|\otimes} \bullet|\otimes b/\epsilon) \otimes|\bullet (a/a^{*|\otimes} \bullet|\otimes b/\epsilon)$. Thus, $shuffle = f^{*|\otimes} \circ ensurelen \circ drop \circ copy_L^{*|\otimes}$.

3. Regular Functions are Closed under Combinators

As mentioned in the introduction, there are multiple equivalent definitions of regular functions. In this paper, we will use the operational model of copyless cost register automata (CCRA) as the yardstick for regularity [5]. A CCRA is a finite state machine which makes a single left-to-right pass over the input string. It maintains a set of registers which are updated on each transition. Examples of register updates include $v := u \otimes v \otimes d$ and $v := d \otimes v$, where $d \in \mathbb{D}$ is a constant. The important restrictions are that transitions and updates are test-free — we do not permit conditions such as "q goes to q' on input a, provided $v \geq 5$" — and that the update expressions satisfy the copyless (or single-use) requirement. CCRAs are a generalization of streaming string transducers to arbitrary monoids. The goal of this paper is to show that functions expressible using the combinators introduced in section 2 are exactly the class of regular functions. In this section, we formally define CCRAs, and show that every function expression represents a regular function.

3.1 Cost register automata

Definition 9. Let V be a finite set of registers. We call a function $f : V \to (V \cup \mathbb{D})^*$ *copyless* if the following two conditions hold:

1. For all registers $u, v \in V$, v occurs at most once in $f(u)$, and
2. for all registers $u, v, w \in V$, if $u \neq w$ and v occurs in $f(u)$, then v does not occur in $f(w)$.

Similarly, a string $e \in (V \cup \mathbb{D})^*$ is copyless if each register v occurs at most once in e.

Definition 10 (Copyless CRA [5])**.** A *CCRA* is a tuple $M = (Q, \Sigma, V, \delta, \mu, q_1, F, \nu)$, where Q is a finite set of states, Σ is a finite input alphabet, V is a finite set of registers, $\delta : Q \times \Sigma \to Q$ is the state transition function, $\mu : Q \times \Sigma \times V \to (V \cup \mathbb{D})^*$ is the register update function such that for all q and a, $\mu(q, a) : V \to V^*$ is copyless over V, $q_1 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\nu : F \to (V \cup \mathbb{D})^*$ is the output function, such that for all q, the output expression $\nu(q)$ is copyless.

The semantics of a CCRA M is specified using configurations. A *configuration* is a tuple $\gamma = (q, val)$ where $q \in Q$ is the current state and $val : V \to \mathbb{D}$ is the register valuation. The initial configuration is $\gamma_1 = (q_1, val_1)$, where $val_1(v) = 0$, for all v. For simplicity of notation, we first extend val to $V \cup \mathbb{D} \to \mathbb{D}$ by defining $val(d) = d$, for all $d \in \mathbb{D}$, and then further extend it to strings $val : (V \cup \mathbb{D})^* \to \mathbb{D}$, by defining $val(v_1 v_2 \dots v_k) = val(v_1) \otimes val(v_2) \otimes \cdots \otimes val(v_k)$. If the machine is in the configuration $\gamma = (q, val)$, then on reading the symbol a, it transitions to the configuration $\gamma' = (q', val')$, and we write $\gamma \to^a \gamma'$, where $q' = \delta(q, a)$, and for all v, $val'(v) = val(\mu(q, a, v))$.

We now define the function $[\![M]\!] : \Sigma^* \to \mathbb{D}_\perp$ *computed by* M. On input $\sigma \in \Sigma^*$, say $\gamma_1 \to^\sigma (q_f, val_f)$. If $q_f \in F$, then $[\![M]\!](\sigma) = val_f(\nu(q_f))$. Otherwise, $[\![M]\!](\sigma) = \perp$.

A cost function is *regular* if it can be computed by a CCRA. A streaming string transducer (SST) is a CCRA where the range $\mathbb{D}$ is the set of strings Γ^* over the output alphabet under concatenation.

Example 11. We present an example of an SST in figure 3.1. The machine $M_{shuffle}$ computes the function *shuffle* from example 7. It maintains 3 registers x, y and z, all initially holding the value ϵ. The register x holds the current output. On viewing each a in the input string, the machine commits to appending the symbol to its output. Depending on the suffix, this a may also be used to eventually produce a b in the output. This provisional value is stored in the register z. The register y holds the b-s produced by the previous run of a-s while the machine is reading the next patch of a-s.

3.2 Additive cost register automata

We recall that when $\mathbb{D}$ is a commutative monoid, CCRAs are equivalent in expressiveness to the simpler model of additive cost register automata (ACRA). In theorem 14, where we show that

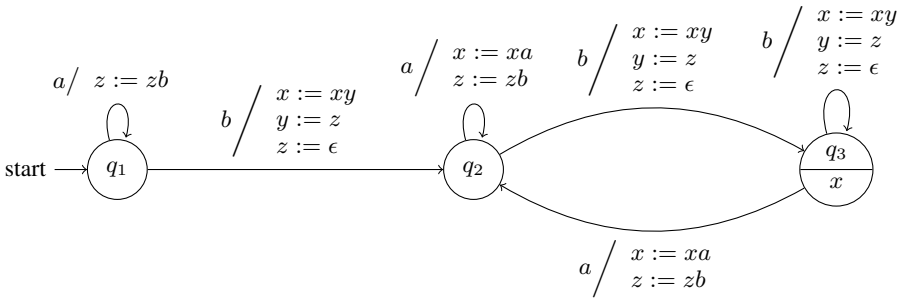


Figure 3.1: Streaming string transducer $M_{shuffle}$ computing *shuffle*. q_3 is the only accepting state. The annotation "x" in state q_3 specifies the output function. On each transition, registers whose updates are not specified are left unchanged.

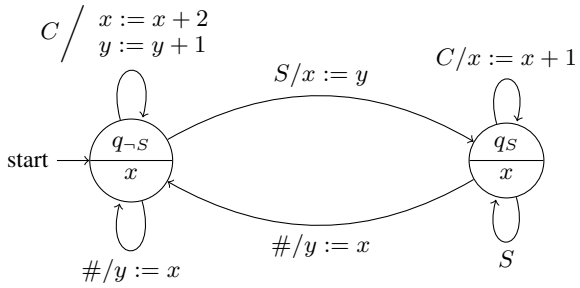


Figure 3.2: ACRA M_{coffee} computing *coffee*.

regular functions over commutative monoids can be expressed using the base functions over regular languages combined using the choice, split sum and function iteration operators, we assume that the regular function is specified as an ACRA. These machines drop the copyless restriction on register updates, but require that all updates be of the form "$u := v \otimes d$", for some registers u and v and some constant d. The reader is referred to the full version of this paper for an explicit definition of ACRAs.

Example 12. In figure 3.2, we present an ACRA M_{coffee} which computes the function *coffee* described in example 4. In the state $q_{\neg S}$, the value in register x tracks how much the customer owes the establishment if he does not fill out a survey before the end of the month, and the value in register y is the amount he should pay otherwise.

3.3 From function expressions to cost register automata

Theorem 13. *Every cost function expressible using the base functions combined using the* $\rhd$, $\otimes$, $\bullet|\otimes$, $\otimes|\bullet$, *iterated and left-iterated sum, input reverse, composition, chained sum, and left-chained sum combinators is regular.*

This can be proved by structural induction on the structure of the function expression. To construct CCRAs for the base functions L/d, we start with the DFA A accepting L, and convert it into a CCRA M where the accepting states output the constant value d. For $f \rhd g$ and $f \otimes g$, we simultaneously evaluate the machines M_f and M_g for f and g using the product construction. To convert function expressions involving the split or iterated sum operators, we use regular look-ahead [3]. For example, while evaluating $f \bullet|\otimes g$, whenever M_f reaches an accepting state, we use regular look-ahead to check whether M_g accepts the suffix, and if so, start executing M_g. Closure under function composition of SSTs is known in the literature [9]. The proof of theorem 13 can be found in the full version of this paper.

4. Completeness of Combinators for Commutative Monoids

In this section, we show that if $\mathbb{D}$ is a commutative monoid, then the base functions combined using the choice, split sum and iteration operators are expressively equivalent to the class of regular functions. First, we review the classical algorithm to convert DFAs into unambiguous regular expressions.

4.1 From DFAs to regular expressions: A review

In this subsection, we describe the classical algorithm [17] that transforms a DFA $A = (Q, \Sigma, \delta, q_1, F)$ into an equivalent unambiguous regular expression; we first use this algorithm in the translation from ACRAs to function expressions, and later in the completeness proof for general non-commutative monoids. Hence this review.

We will be interested in *unambiguous* regular expressions: each symbol $a \in \Sigma$ is associated with an unambiguous r.e. R_a, whose language $L(R_a) = \{a\}$; if R_1 and R_2 are unambiguous regular expressions, then $R_1 \cdot R_2$ is an unambiguous r.e. whose language $L(R_1 \cdot R_2)$ is the set of strings $\sigma \in \Sigma^*$ which can be uniquely written $\sigma = \sigma_1\sigma_2$ such that $\sigma_1 \in L(R_1)$ and $\sigma_2 \in L(R_2)$; similarly whenever R is an unambiguous r.e., then R^* is an unambiguous r.e., and $L(R^*)$ is the set of all strings $\sigma \in \Sigma^*$ such that there exist unique strings $\sigma_1, \sigma_2, \ldots, \sigma_k \in L(R)$, and $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$; and if R_1 and R_2 are unambiguous regular expressions, then $R_1 \uplus R_2$ is an unambiguous r.e. such that $L(R_1 \uplus R_2) = \{\sigma \in \Sigma^* \mid \sigma \in L(R_1)$, or $\sigma \in L(R_2)$, but not both$\}$. For example, $aab \in L(\Sigma^*)$, but $aab \notin L(\Sigma^* \cdot b \uplus \Sigma^* \cdot a \cdot b)$. Unambiguous regular expressions are similar to unambiguous context-free grammars, and are important here because we are interested in *functions* (as opposed to *relations*) from strings to a monoid.

Let $Q = \{q_1, q_2, \ldots, q_n\}$. For each pair of states $q, q' \in Q$, and for $i \in \mathbb{N}$, $0 \leq i \leq n$, $r^{(i)}(q, q')$ is the set of non-empty strings σ such that $\delta(q, \sigma) = q'$, and which only pass through the intermediate states $\{q_1, q_2, \ldots, q_i\}$. This can be inductively constructed as follows:

1. $r^{(0)}(q, q') = \{a \in \Sigma \mid q \to^a q'\}$.
2. $r^{(i+1)}(q, q') = r^{(i)}(q, q_{i+1}) \cdot r^{(i)}(q_{i+1}, q_{i+1})^* \cdot r^{(i)}(q_{i+1}, q') \uplus r^{(i)}(q, q')$.

The language L accepted by A is then given by the regular expression $R_A = \biguplus_{q_f \in F} r^{(n)}(q_1, q_f)$ if $q_1 \notin F$, and otherwise by the regular expression $\{\epsilon\} \uplus R_A$.

4.2 Converting function expressions to ACRAs

We sketch the proof using the ACRA M_{coffee} from figure 3.2. The idea is to view M_{coffee} as a non-deterministic automaton A_{coffee} (figure 4.1) over the set of vertices $Q \times V$: the state (q, v) of the

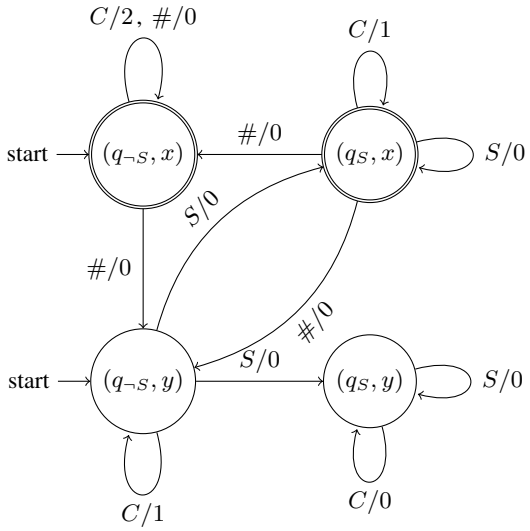


Figure 4.1: Translating an ACRA to a function expression. A_{coffee} is the NFA that results from the construction of theorem 14.

A_{coffee} indicates the current state q of M_{coffee}, and (informally) a guess v of the register whose current value influences the final output. For every path $q_1 \to^{\sigma_1} q_2 \to^{\sigma_2} \cdots \to^{\sigma_n} q_{n+1}$ through M_{coffee}, there is a corresponding path through A_{coffee}, $(q_1, v_2) \to^{\sigma_1} (q_2, v_2) \to^{\sigma_2} \cdots \to^{\sigma_n} (q_{n+1}, v_{n+1})$, where v_{n+1} is the register which is output in the final state q_{n+1}, and during each transition $(q_i, v_i) \to^{\sigma_i} (q_{i+1}, v_{i+1})$, the value of the register v_i flows into v_{i+1}. The final value of register v_{n+1} is simply the sum of the increments accumulated along each transition of this accepting path. Also observe that this NFA A is unambiguous — for every string σ that is accepted by A, there is a unique accepting path. Therefore, if the label $(q, v) \to^{a/d} (q', v')$ along each edge is also annotated with the increment value d, where that the update expression reads $\mu(q, a, v') = v \otimes d$, then any unambiguous regular expression for the language accepted by A_{coffee} can be alternatively viewed as a function expression for $[\![M_{coffee}]\!]$. This argument can be generalized to obtain:

Theorem 14. *If $(\mathbb{D}, \otimes, 0)$ is a commutative monoid, then every regular function $f : \Sigma^* \to \mathbb{D}$ can be expressed using the base functions combined with the unambiguous choice ($\uplus$), split sum ($\bullet|\otimes$) and iterated sum ($\cdot^{*|\otimes}$) operators.*

Note that the unambiguous choice operator in theorem 14 can be alternatively replaced by the conditional choice operator.

5. Completeness of Combinators for General Monoids

In this section, we show that all regular functions $f : \Sigma^* \to \mathbb{D}$ can be written as function expressions:

Theorem 15. *For an arbitrary finite alphabet Σ and monoid $(\mathbb{D}, \otimes, 0)$, every regular function $f : \Sigma^* \to \mathbb{D}$ can be expressed using the base functions combined with unambiguous choice ($\uplus$), sum ($\otimes$), split sum ($\bullet|\otimes$), chained sum ($(\cdot,\cdot)^{*|\otimes}$), and their left-additive versions ($\otimes|\bullet$, $(\cdot,\cdot)^{\otimes|*}$).*

The unambiguous choice operator can be equivalently replaced with the conditional choice operator $\triangleright$, and the theorem continues to hold. To simplify the presentation, we prove theorem 15 only for the case of string transductions, i.e. where $\mathbb{D} = \Gamma^*$, for some finite output alphabet Γ. Note that this is sufficient to establish the theorem in its full generality: let $\Gamma_{\mathbb{D}} \subseteq \mathbb{D}$ be the (necessarily finite) set of all constants appearing in the textual description of M. M can be alternatively viewed as an SST mapping input strings in Σ^* to output strings in $\Gamma_{\mathbb{D}}^*$. The restricted version of theorem 15 can then be used to convert this SST to function expression form, which when interpreted over the original domain $\mathbb{D}$ represents $[\![M]\!]$.

5.1 Proof outline

The SST-to-function expression translation algorithm proceeds in lockstep with the DFA-to-regular expression translator from section 4.1. The idea is to compute, in step i, for each pair of states q and q', a collection of function expressions which together summarize all strings $\sigma \in r^{(i)}(q, q')$.

In the CCRA $M_{shuffle}$ from figure 3.1, consider the string aab processed from the state q_1. After processing aab, $M_{shuffle}$ is left in q_2. The final values of the registers x, y, and z, in terms of their initial values, are $xaay$, zb, and ϵ respectively. Thus, the string aab processed from q_1 may be summarized by the pair $(q_2 = \delta(q, aab), \{x \mapsto xaay, y \mapsto zb, z \mapsto \epsilon\})$, indicating the state of the machine after processing the string, and the final values of the registers. Similarly, the summaries for the strings $baab$ and $baabb$ processed from the initial state q_1 are $(q_2, \{x \mapsto xyaaz, y \mapsto bb, z \mapsto \epsilon\})$ and $(q_2, \{x \mapsto xyaazbb, y \mapsto \epsilon, z \mapsto \epsilon\})$ respectively.

Let us concentrate on the patterns in which register values are updated during computation. For the strings aab, $baab$, and $baabb$, these are, respectively, $\{x \mapsto \gamma_1 x \gamma_2 y \gamma_3, y \mapsto \gamma_4 z \gamma_5, z \mapsto \gamma_6\}$, $\{x \mapsto \gamma_1 x \gamma_2 y \gamma_3 z \gamma_4, y \mapsto \gamma_5, z \mapsto \gamma_6\}$, and $\{x \mapsto \gamma_1 x \gamma_2 y \gamma_3 z \gamma_4, y \mapsto \gamma_5, z \mapsto \gamma_6\}$, for some input-dependent string constants $\gamma_1, \ldots, \gamma_6 \in \Gamma^*$. We call these patterns the "shapes" of the input strings. Note that $baab$ and $baabb$ have identical shapes.

First, we observe that the set of all strings $\sigma \in r^{(i)}(q, q')$ with a given shape S is a regular language. Next, for each shape S, we compute an "expression vector" $\mathbf{R}_S^{(i)}(q, q')$: for each patch j appearing in the register update expression, there is a corresponding function expression $\mathbf{R}_{S,j}^{(i)}(q, q') : \Sigma^* \to \Gamma^*$ such that for all input strings $\sigma \in r^{(i)}(q, q')$ with shape S, $\gamma_j = \mathbf{R}_{S,j}^{(i)}(q, q')(\sigma)$. Finally, for each accepting state $q_f \in F$, using the summary $\mathbf{R}_S^{(n)}(q_0, q_f)$, we compute the output of the CCRA on strings σ that reach the state q_f with shape S. Together with our earlier observation that the set of such strings is regular, we can construct a function expression f which is equivalent to the given CCRA.

We formally define shapes and expression vectors in subsection 5.2, and informally describe some basic operations over them in subsection 5.3. We omit the explicit construction of $\mathbf{R}_S^{(0)}(q, q')$ in the short version of this paper. Subsections 5.4–5.7 describe the inductive construction of $\mathbf{R}_S^{(i+1)}(q, q')$. Note that the chained sum is crucial to this construction, and will appear while finally constructing $\mathbf{R}_S^{(i+1)}(q, q')$, in subsection 5.7.

5.2 A theory of shapes

Definition 16 (Shape of a path)**.** A *shape* $S : V \to V^*$ is a copyless function over a finite set of registers V. Let $\sigma = q_1 \to^{\sigma_1} q_2 \to^{\sigma_2} \to \cdots \to^{\sigma_n} q_{n+1}$ be a path through a CCRA 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 string projection onto V of the register update expression $\mu(q_1, \sigma, v)$: $S_\sigma(v) = \pi_V(\mu(q_1, \sigma, v))$.

We refer to a string constant in the update expression as a *patch* in the corresponding shape. Because there are only a finite number of copyless functions $V \to V^*$ over a finite set V, the space of shapes is finite.

Example 17. It is helpful to visualize shapes as bipartite graphs, such as in figure 5.1. The numbers on labelled edges to a register

v indicate the order of the registers in $S(v)$. Since the shape of a path indicates the pattern in which register values flow during computation, an edge $u \to v$ can be informally read as "The value of u flows into v". Because of the copyless 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. For some register $v \in V$, let $S_2(v) = v_1 v_2 \dots v_k$. Then $S_1 \cdot S_2(v) = s_1 s_2 \dots s_k$, where $s_i = S_1(v_i)$. Informally, the concatenation of shapes corresponds to the composition of their bipartite graph visualizations. By definition, therefore,

Proposition 18. *Let σ_1 and σ_2 be two paths through a CCRA M such that the final state of σ_1 is the same as the initial state of σ_2. Then, for all registers v, $S_{\sigma_1\sigma_2}(v) = S_{\sigma_1} \cdot S_{\sigma_2}(v)$.*

It follows from the previous proposition that

Proposition 19. *Let $q, q' \in Q$ be two states in a CCRA M, and S be a shape. The set of all strings from q to q' in M with shape S is regular.*

Formally, an *expression vector* $\mathbf{A}$ for a shape S is a collection of function expressions, such that for each register v, and for each patch k in $S(v)$, there is a corresponding function expression $\mathbf{A}_{v,k} : \Sigma^* \to \Gamma^*$. An expression vector $\mathbf{A}$ *summarizes a set of paths* L with shape S, if for each path $\pi \in L$ with initial state q, and input string σ, and for each register v, in the update expression $\mu(q, \sigma, v)$, the constant value $\gamma_{v,k} \in \Gamma^*$ at position k is given by $\mathbf{A}_{v,k}(\sigma)$.

Example 20. Consider the loop a^* at the state q_2 of $M_{shuffle}$. Consider the concrete string a^k. The effect of this string is to update $x := xa^k$, $y := y$, and $z := zb^k$. The shape of this set of paths is the identity function $S(v) = v$, for all v. Define the expression vector $\mathbf{A}$ as follows: $\mathbf{A}_{x,1} = \mathbf{A}_{y,1} = \mathbf{A}_{y,2} = \mathbf{A}_{z,1} = a^*/\epsilon$, $\mathbf{A}_{x,2} = (a/a)^{*|\otimes}$, and $\mathbf{A}_{z,2} = (a/b)^{*|\otimes}$. Then $\mathbf{A}$ summarizes the set of paths a^* at the state q_1.

We now restate the desired invariant (informally described in the proof outline in subsection 5.1): in step i, for each pair of states q and q', and for each shape S, the expression vector $\mathbf{R}_S^{(i)}(q, q')$ summarizes all paths $\sigma \in r^{(i)}(q, q')$ with shape S.

5.3 Operations on expression vectors

In this subsection, we create a library of basic operations on expression vectors.

5.3.1 Restricting function domains

Given an expression vector $\mathbf{A}$ for a shape S, the domain of the expression vector, written as $\mathrm{Dom}(\mathbf{A})$, is defined as the language $\bigcap_{v,k} \mathrm{Dom}(\mathbf{A}_{v,k})$, where $\mathrm{Dom}(\mathbf{A}_{v,k})$ is the domain of the component function expressions. Given a cost function $f : \Sigma^* \to \Gamma^*$ and a language $L \subseteq \Sigma^*$, we first define the *restriction of f to L* as $f \cap L = f \otimes L/\epsilon$. This is equivalent to saying that $f \cap L(\sigma) = f(\sigma)$, if $\sigma \in L$, and $f \cap L(\sigma) = \bot$, otherwise. We extend this to restrict expression vectors $\mathbf{A}$ to languages L, $\mathbf{A} \cap L$, by defining $(\mathbf{A} \cap L)_{v,k}$ as $\mathbf{A}_{v,k} \cap L$.

5.3.2 Choice

Let $\mathbf{A}$ and $\mathbf{B}$ be expression vectors, both for some shape S. Let $\mathbf{A}' = \mathbf{A} \cap \mathrm{Dom}(\mathbf{A})$ and $\mathbf{B}' = \mathbf{B} \cap \mathrm{Dom}(\mathbf{B})$. Then, we define the *conditional choice* $\mathbf{A} \rhd \mathbf{B}$ as the expression vector for the shape S such that for each register v and patch k, $(\mathbf{A} \rhd \mathbf{B})_{v,k} = \mathbf{A}'_{v,k} \rhd \mathbf{B}'_{v,k}$, and the *unambiguous choice* $\mathbf{A} \uplus \mathbf{B}$ as the expression vector for the shape S such that for each v and k, $(\mathbf{A} \uplus \mathbf{B})_{v,k} = \mathbf{A}'_{v,k} \uplus \mathbf{B}'_{v,k}$.

Claim 21. If L and L' are disjoint sets of paths with the same shape S, such that $\mathbf{A}$ summarizes paths in L and $\mathbf{B}$ summarizes paths in L', then both expression vectors $\mathbf{A} \rhd \mathbf{B}$ and $\mathbf{A} \uplus \mathbf{B}$ summarize paths in $L \cup L'$.

The notation $\biguplus\{f_1, f_2, \dots, f_k\}$ stands for $f_1 \uplus f_2 \uplus \dots \uplus f_k$. As the base case, $\biguplus\{\} = \bot$. We ensure that when this notation is used, the functions have mutually disjoint domains, so that the order or parenthesization of the subexpressions is immaterial.

5.3.3 Shifting expressions

Given a cost function f and a language L, the *left-shifted function* $f \ll L$ is the function which reads an input string $\sigma = \sigma_1\sigma_2$ such that $\sigma_1 \in \mathrm{Dom}(f)$ and $\sigma_2 \in L$, and applies f to the prefix σ_1 and ignores the suffix σ_2, provided the split is unique, i.e. $f \ll L = f \bullet|\otimes L/\epsilon$. Similarly, the *right-shifted function* $f \gg L = L/\epsilon \bullet|\otimes f$. The shift operators can also be extended to expression vectors: $\mathbf{A} \ll L$ is defined as $(\mathbf{A} \ll L)_{v,k} = \mathbf{A}_{v,k} \cap \mathrm{Dom}(\mathbf{A}) \ll L$, and $\mathbf{A} \gg L$ is defined as $(\mathbf{A} \gg L)_{v,k} = \mathbf{A}_{v,k} \cap \mathrm{Dom}(\mathbf{A}) \gg L$.

5.3.4 Concatenation

Pick three states q, q', and q'' in the given CCRA, and let L be a set of paths from q to q' with shape S, and L' be a set of paths from q' to q'' with shape S'. Then paths $\sigma \in L \cdot L'$ from q to q'' have shape $S \cdot S'$. Let the expression vectors $\mathbf{A}$ and $\mathbf{B}$ summarize all paths in L and L' respectively, and consider a string $\sigma = \sigma_1\sigma_2$ with $\sigma_1 \in L$ and $\sigma_2 \in L'$. The expression vector $\mathbf{A}$ describes the register values of the machine after processing σ_1, in terms of their original values before the computation was started, and $\mathbf{B}$ describes the register values of the machine after processing σ_2, in terms of their values after processing σ_1. These expression vectors can therefore be combined into an expression vector $\mathbf{A} \cdot \mathbf{B}$, which describes the register values after processing $\sigma_1\sigma_2$, in terms of their values before computation started. The idea is to use the shift operator to apply the desired component expressions of $\mathbf{A}$ and $\mathbf{B}$ to the relevant parts of the input string σ, and combine the outputs using the sum operator. The explicit construction will be found in the full version of this paper.

5.4 A total order over the registers

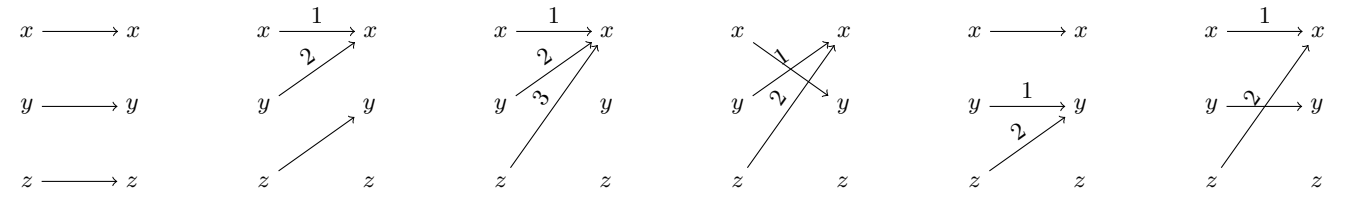
During the iteration step of the construction, we have to provide function expressions for $\mathbf{R}_S^{(i+1)}(q, q')$ in terms of the candidate function expressions at step i. To satisfy the translation invariant, we have to summarize all paths in $r^{(i+1)}(q, q') = r^{(i)}(q, q_{i+1}) \cdot r^{(i)}(q_{i+1}, q_{i+1})^* \cdot r^{(i)}(q_{i+1}, q') \uplus r^{(i)}(q, q')$. Therefore, the central problem is to construct, for each shape S, an expression vector $\mathbf{B}_S$ which summarizes paths in $loop^{(i)}$ with shape S. In this subsection and the next, we impose certain simplifying assumptions on the shapes under consideration. We will explicitly summarize loops in subsections 5.6 and 5.7.

Register values may flow in complicated ways: consider for example the shape in figure 5.1d. The construction of $\mathbf{R}_S^{(i+1)}(q, q')$ is greatly simplified if we assume that the shapes under consideration are idempotent under concatenation.

Definition 22. Let V be a finite set of registers, and $\preceq$ be a total order over V. We call a shape S over V *normalized* with respect to $\preceq$ if

1. for all $u, v \in V$, if v occurs in $S(u)$, then $u \preceq v$,
2. for all $u, v \in V$, if v occurs in $S(u)$, then u itself occurs in $S(u)$, and
3. for all $v \in V$, there exists $u \in V$ such that v occurs in $S(u)$.

A CCRA M is normalized if the shape of each of its update expressions is normalized with respect to $\preceq$.



(a) $q_1 \to^a q_1$, $S_\perp$. (b) $q_2 \to^b q_3$. (c) $q_2 \to^b q_3 \to^b q_3$, $S_\top$. (d) Shape of the update $x := yz$, $y := x$, $z := \epsilon$. (e) Shape S_1 of the update $x := x$, $y := yz$, $z := \epsilon$. (f) Shape S_2 of the update $x := xz$, $y := y$, $z := \epsilon$.

Figure 5.1: Visualizing shapes as bipartite graphs. Figures 5.1a–5.1c describe the shapes of some paths in $M_{shuffle}$ from figure 3.1.

For example, the shapes in figures 5.1a, 5.1c, 5.1e, and 5.1f are normalized, while 5.1b and 5.1d are not. Informally, the first condition requires that all registers in the CCRA flow "upward", the second ensures that shapes are idempotent, and the third ensures that no register value is ever lost during computation. Observe that if the individual transitions in a path are normalized, then the whole path is itself normalized. It can be shown that:

Proposition 23. *For every CCRA M, there is an equivalent normalized CCRA M'.*

The states of M' are pairs (q, f), where q is the corresponding state in M, and the "register renaming" function $f : V \to V'$ associates each register of M with a (state-specific) register of M'. $V' = \{x_0, x_1, x_2, \ldots, x_{|V|}\}$ has one more register than V — this additional register x_0 is a sink register to satisfy rule 3 of definition 22. The idea is to carefully construct the transition function δ', so that the register values flow only upwards during each transition, and satisfy all the conditions of definition 22.

We will now assume that all CCRAs and shapes under consideration are normalized, and we elide this assumption in all definitions and theorems.

5.5 A partial order over shapes

We 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 5.1. Let σ be a path through the CCRA with shape S_1. Then, no sub-path of σ can have shape $S_\top$, because if such a sub-path were to exist, then the value in register y would be promoted to x, and the registers x and y could then never be separated. We now create a partial-order $\sqsubseteq$, and an equivalence relation $\sim$ over the set $\mathbb{S}$ of upward flowing shapes which together capture this notion of "can appear as a subpath".

Definition 24. If S is a shape over the set of registers V, then the support of S, $supp(S) = \{v \in V \mid v \text{ occurs in } S(v)\}$. If S_1 and S_2 are two shapes, then $S_1 \sqsubset S_2$ iff $supp(S_1) \supset supp(S_2)$. We call two shapes S_1 and S_2 *support-equal*, written as $S_1 \sim S_2$, if $supp(S_1) = supp(S_2)$.

For example, the shape $S_\perp$ from figure 5.1 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 $S_\top$. Note that for the strict ordering $S_1 \sqsubset S_2$, we enforce the strict subset relation between $supp(S_1)$ and $supp(S_2)$. The following two claims, 25 and 26, formalize the intuition that $\sqsubseteq$ and $\sim$ describe the possible shapes of subpaths.

Claim 25. Let σ be a path through the CCRA M with shape S, and σ' be a subpath of σ with shape S'. If $S' \not\sqsubset S$, then $S' \sim S$.

Claim 26. Let σ be a path through the CCRA M with shape S, and let σ' be the shortest prefix with shape S' such that $S' \not\sqsubset S$. Then $S' = S$.

5.6 Decomposing loops

Recall that our main remaining problem, starting from subsection 5.4, has been to construct $\mathbf{R}_S^{(i+1)}(q, q')$, which summarizes strings in $r^{(i+1)}(q, q')$ with shape S. In subsection 5.3, we defined the choice operator over expression vectors: thus, if $\mathbf{C}_S$ summarizes strings in $r^{(i)}(q, q_{i+1}) \cdot r^{(i)}(q_{i+1}, q_{i+1})^* \cdot r^{(i)}(q_{i+1}, q')$ with shape S, then we can define $\mathbf{R}_S^{(i+1)}(q, q') = \mathbf{C}_S \uplus \mathbf{R}_S^{(i)}(q, q')$. Furthermore, we also defined the concatenation operator over expression vectors: if for each intermediate shape S_2, if $\mathbf{B}_{S_2}$ summarizes strings in $r^{(i)}(q_{i+1}, q_{i+1})^*$ with shape S_2, then we can construct the desired

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

Our goal is therefore to construct $\mathbf{B}_S$, for each S. In subsection 5.4, we restricted the space of shapes under consideration, so that shapes are idempotent under concatenation, and in subsection 5.5, we defined a partial order $\sqsubseteq$ over shapes, and a notion of support-equality $\sim$, which together constrain the shapes of subpaths of a given path σ. In this subsection and the next, we construct $\mathbf{B}_S$ assuming that $\mathbf{B}_{S'}$ for all shapes $S' \sqsubset S$ is known. Furthermore, since each string $r^{(i)}(q_{i+1}, q_{i+1})$ is non-empty, $\epsilon \notin r^{(i)}(q_{i+1}, q_{i+1})^+$. We therefore separately handle the case of $\epsilon \in r^{(i)}(q_{i+1}, q_{i+1})^*$, and in the rest of this section, we construct an expression vector $\mathbf{B}_S^+$ which summarizes strings (all of which are non-empty) in $r^{(i)}(q_{i+1}, q_{i+1})^+$ with shape S.

Consider any path $\sigma \in r^{(i)}(q_{i+1}, q_{i+1})^+$ with shape S. From claims 25 and 26, we can unambiguously decompose $\sigma = \sigma_1\sigma_2 \ldots \sigma_k\sigma_f$, where

1. for each j, $1 \le j \le k$, $\sigma_j \in r^{(i)}(q_{i+1}, q_{i+1})^+$ is a self-loop at q_{i+1}, with shape S_j such that $S_j \sim S$,
2. for each j, $1 \le j \le k$, and for each proper prefix (possibly empty) $\sigma_{pre} \in r^{(i)}(q_{i+1}, q_{i+1})^*$ of σ_j, $S_{pre} \sqsubset S$, and
3. $\sigma_f \in r^{(i)}(q_{i+1}, q_{i+1})^*$ (possibly empty), and its shape S_f satisfies $S_f \sqsubset S$.

We call the split $\sigma = \sigma_1\sigma_2 \ldots \sigma_k\sigma_f$ the *S-decomposition* of σ. See figure 5.2.

We conclude this subsection by constructing expression vectors $\mathbf{A}_{S_j}$, which summarize these minimal subpaths σ_j. First note that $\sigma_j \in L_{first}(S_j)$, where for each shape $S' \sim S$, $L_{first}(S_j)$ is the set of all paths $\sigma \in r^{(i)}(q_{i+1}, q_{i+1})^+$ with shape S_j such that no proper prefix σ_{pre} of σ has shape $S_{pre} \sim S$. Next, unambiguously decompose $\sigma = \sigma_{pre}\sigma_{last}$, with $\sigma_{pre} \in r^{(i)}(q_{i+1}, q_{i+1})^*$, $\sigma_{last} \in r^{(i)}(q_{i+1}, q_{i+1})$: therefore σ_{last} is the last iteration of the loop in $r^{(i)}(q_{i+1}, q_{i+1})$ and $S_{pre} \sqsubset S$. By the first induction hypothesis, σ_{suff} is summarized by the expression vector

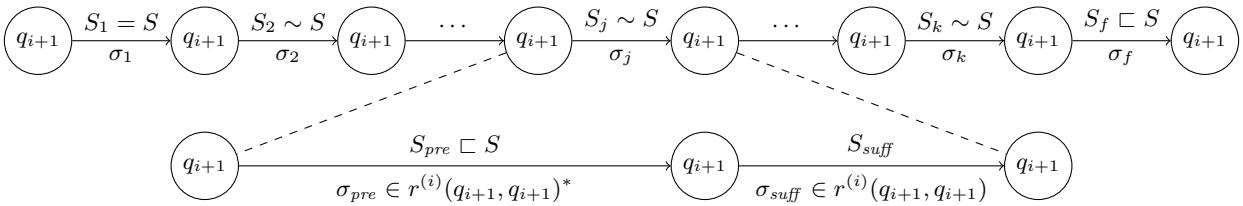


Figure 5.2: Decomposing paths in $r^{(i)}(q_{i+1}, q_{i+1})^+$ with shape S. For each j, $1 \leq j \leq k$, each proper prefix σ_{pre} of σ_j has shape $S_{pre} \sqsubset S$, or equivalently $\sigma_j \in L_{first}(S_j)$. σ_j can be unambiguously written as $\sigma_{pre}\sigma_{suff}$, with $\sigma_{suff} \in r^{(i)}(q_{i+1}, q_{i+1})$.

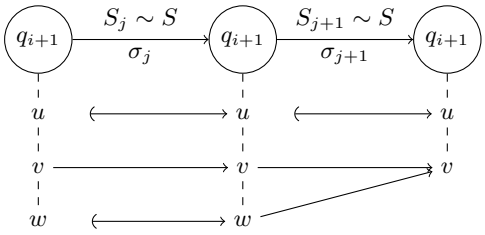


Figure 5.3: Every non-support register $u \notin supp(S)$ is reset while processing σ_j, for $1 \leq j \leq k$. Therefore its value is determined entirely by σ_j. For any path in $r^{(i)}(q_{i+1}, q_{i+1})^+$, inward flows into register v have to be from non-support registers w, and therefore the value appended to v while processing σ_{j+1} is determined by σ_j.

$\mathbf{R}^{(i)}_{S_{suff}}(q_{i+1}, q_{i+1})$, and by the second induction hypothesis, $\mathbf{B}_{S_{pre}}$ is known. Define $\mathbf{A}_{S'} = \biguplus\{\mathbf{B}_{S_{pre}} \cdot \mathbf{R}^{(i)}_{S_{suff}}(q_{i+1}, q_{i+1}) \mid S_{pre} \cdot S_{suff} = S' \text{ and } S_{pre} \sqsubset S\} \uplus \mathbf{R}^{(i)}_{S'}(q_{i+1}, q_{i+1})$. By construction:

Claim 27. For all shapes $S' \sim S$, the expression vector $\mathbf{A}_{S'}$ summarizes all paths in $L_{first}(S')$.

5.7 Constructing $\mathbf{B}_S$

Starting from the previous subsection, our main goal has been to construct the expression vector $\mathbf{B}_S$, which summarizes paths $\sigma \in r^{(i)}(q_{i+1}, q_{i+1})^*$ with shape S. From subsection 5.5, we know the possible shapes of subpaths of σ, and we therefore assumed an additional induction hypothesis that $\mathbf{B}_{S'}$ is known for all shapes $S' \sqsubset S$. We also resolved to handle the case of $\epsilon \in r^{(i)}(q_{i+1}, q_{i+1})^*$ separately, so that we are interested in constructing $\mathbf{B}^+_S$, which summarizes paths $\sigma \in r^{(i)}(q_{i+1}, q_{i+1})^+$. In the previous subsection, we constructed expression vectors $\mathbf{A}_S$ for "minimal paths" $\sigma \in r^{(i)}(q_{i+1}, q_{i+1})^+$: consider an arbitrary path $\sigma \in r^{(i)}(q_{i+1}, q_{i+1})^+$, and its S-decomposition $\sigma = \sigma_1\sigma_2 \ldots \sigma_k\sigma_f$, where each $\sigma_j \in r^{(i)}(q_{i+1}, q_{i+1})^+$, for each j, $1 \leq j \leq k$, $\sigma_j \in L_{first}(S_j)$, and $S_f \sqsubset S$. Then claim 27 establishes that $\mathbf{A}_{S_j}$ summarizes each σ_j. In this section, we complete the construction of the expression vector $\mathbf{B}^+_S$, and hence also the construction of $\mathbf{B}_S$, and consequently the proof of theorem 15. Note that the chained sum is critical to the construction in this subsection, because (informally) the values being appended to a register while processing σ_j may be computed while processing σ_{j-1}. We distinguish three cases to construct $\mathbf{B}^+_{S,v,k}$.

5.7.1 $S(v) = \epsilon$

Informally, this is the case when v is reset in S, and therefore reset during each minimal subpath σ_j (but not necessarily during σ_f). Therefore, the final value of the register v is entirely determined by the substring $\sigma_k\sigma_f$. Register u in figure 5.3 is a visualization of this situation. Let $\mathbf{F} = \biguplus\{\mathbf{A}_{S_1} \cdot \mathbf{B}_{S_2} \mid S_1 \cdot S_2 = S, S_1 \sim S \text{ and } S_2 \sqsubset S\}$, and define $\mathbf{B}^+_{S,v,1} = r^{(i)}(q_{i+1}, q_{i+1})^* \Big/ \epsilon \bullet\!|\otimes \mathbf{F}_{v,1}$.

5.7.2 $S(v) \neq \epsilon$, and $1 < k < |S(v)| + 1$

This corresponds to the case when k is an patch in $S(v)$. Consider the two update expressions $\{x := xaby, y := b\}$, and $\{x := axbyc, y := b\}$ with the same shape S. These can be concatenated into the single update expression $\{x := axabybbc, y := b\}$. Note the internal patch $\gamma_{x,2} = ab$, and observe that its value has been fixed by the first update, because once the registers 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 the k^{th} patch in the update expression for v is determined entirely by σ_1 which has shape $S_1 = S$ by claim 26. Define $\mathbf{B}^+_{S,v,k} = \mathbf{A}_{S,v,k} \bullet\!|\otimes\ r^{(i)}(q_{i+1}, q_{i+1})^* \Big/ \epsilon$.

5.7.3 $S(v) \neq \epsilon$, and $k = 1$, or $k = |S(v)| + 1$

This is the case when k is either the first or the last patch. First, we know that $v \in supp(S)$. Also, any registers which flow into v while processing σ have to be non-support registers. See figure 5.3. Thus, the value being appended to v while processing σ_j is determined entirely by σ_j and σ_{j-1}. We will define $\mathbf{B}^+_{S,v,k}$ for $k = |S(v)| + 1$. The case for $k = 1$ is symmetric.

Consider the S-decomposition of the input $\sigma = \sigma_1\sigma_2 \ldots \sigma_k\sigma_f$, and let $\gamma_{v,k}$ be the last patch in the update expression for v. While processing each substring $\sigma_1, \sigma_2, \ldots, \sigma_k, \sigma_f$, the CCRA appends some value to the end of $\gamma_{v,k}$. Note that the value appended by σ_1 is determined entirely by σ_1, the value appended by σ_f is determined by $\sigma_k\sigma_f$, and the value appended while processing σ_j, $1 < j \leq k$ is determined by $\sigma_{j-1}\sigma_j$.

1. While processing σ_1, some symbols are appended to the k^{th} position in $S(v)$. Define $f_{pre} = \mathbf{A}_{S,v,k} \bullet\!|\otimes\ r^{(i)}(q_{i+1}, q_{i+1})^* \Big/ \epsilon$, so that $f_{pre}(\sigma)$ is the value appended at the end of $\gamma_{v,k}$ by σ_1.
2. For some register $u \neq v$, let u occur in $S(v)$, so that the value in u computed during σ_k flows into v while processing σ_f. For each pair of shapes S_k and S_f such that $S_k \sim S$, and $S_f \sqsubset S$, consider $\mathbf{A}'_{S_k} = \mathbf{A}_{S_k} \ll \text{Dom}(S_f)$, and $\mathbf{B}'_{S_f} = \mathbf{B}_{S_f} \gg \text{Dom}(\mathbf{A}_{S_k})$. Consider the update expression $\mathbf{B}'_{S_f,v}$: say this is $v := \sigma v \tau$, where σ and τ are strings over expressions and registers. For each register u in τ, substitute the value $\mathbf{A}'_{S_k,u,1}$ — since u was reset while processing S_k, this expression gives the contents of the register u — and interpret string concatenation in τ as the function combinator sum. Label this result as f_{post,S_k,S_f}. Define $f_{post} = (r^{(i)}(q_{i+1}, q_{i+1})^* \Big/ \epsilon) \bullet\!|\otimes \biguplus\{f_{post,S_k,S_f} \mid S_k \sim S \text{ and } S_f \sqsubset S\}$, and observe that $f_{post}(\sigma)$ is the value appended to the end of $\gamma_{v,k}$ by σ_f.

3. Finally, consider the value appended while processing σ_j, for $j > 1$. This is similar to the case for σ_f: the value appended to $\gamma_{v,k}$ by σ_j is determined by $\sigma_{j-1}\sigma_j$. For each pair of states $S_{j-1} \sim S$ and $S_j \sim S$, consider $\mathbf{A}'_{S_{j-1}} = \mathbf{A}_{S_{j-1}} \ll \mathrm{Dom}(\mathbf{A}_{S_j})$, and $\mathbf{A}'_{S_j} = \mathbf{A}_{S_j} \gg \mathrm{Dom}(\mathbf{A}_{S_{j-1}})$. Consider the update expression $\mathbf{A}'_{S_j,v,k}$. Let this be $v := \sigma v \tau$, where σ and τ are strings over expressions and registers. For each register u in τ, substitute the value $\mathbf{A}'_{S_{j-1},u,1}$ — since u was reset while processing S_{j-1}, this expression gives the contents of the register u — and interpret string concatenation in τ as the function combinator sum. Label this result as f_{S_{j-1},S_j}. Define $f = (\biguplus\{f_{S_{j-1},S_j} \mid S_{j-1} \sim S \text{ and } S_j \sim S\}, L_f)^{*|\otimes}$.

Finally, define $\mathbf{B}^+_{S,v,k} = (f_{pre} \otimes f_{post}) \uplus (f_{pre} \otimes f \otimes f_{post})$. The following observation completes the proof of theorem 15.

Claim 28. $\mathbf{B}^+_S$ summarizes all strings $\sigma \in r^{(i)}(q_{i+1}, q_{i+1})^+$ with shape S.

5.8 Recap of theorem 15

In this section, we established that the combinators of section 2 are sufficient to express all regular functions, over all monoids. The principal difficulty is that, in the case of SSTs, a single input symbol may influence non-contiguous substrings in the output in complicated ways, such as in the function *shuffle* from figure 2.1a. The solution was to use the new operation of chained sum 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 from the state q to the state q' while only passing through intermediate states q_j, where $j \leq i$. We first associated paths within the SST with their shapes, indicating the pattern of data flows. We then investigated the possible shapes of subpaths of a given path, and proved that the notions of shape ordering $\sqsubset$, and support equality $\sim$ together capture this notion[2]. This allowed us to set up a nested inductive construction, where we summarized strings with the shape S, assuming the summaries for strings with the shape S', for $S' \sqsubset S$ were known. The copylessness of SSTs was essential in this step because the space of shapes is then finite.

6. Conclusion

In this paper, we have characterized the class of regular functions that map strings to values from a monoid using a set of function combinators. We hope that these results provide additional evidence of robust and foundational nature of this class. The identification of the combinator of chained sum, and its role in the proof of expressive completeness of the combinators, should be of particular technical interest. There are many avenues for future research. First, the question whether all the combinators we have used are *necessary* for capturing all regular functions remains open (we conjecture that the set of combinators is indeed minimal). Second, it is an open problem to develop the notion of a congruence and a Myhill-Nerode-style characterization for regular functions (see [7] for an attempt where authors give such a characterization, but succeed only after retaining the "origin" information that associates each output symbol with a specific input position). Third, it would be worthwhile to find analogous algebraic characterizations of regularity when the domain is, instead of finite strings, infinite strings [6] or trees [4, 14] and/or when the range is a semiring [5, 12]. Finally, on the practical side, we plan to develop a declarative language for document processing based on the regular combinators identified in this paper.

[2] In an early attempt to prove this theorem, we tried to formalize the relation $R_{sp}(S_1, S_2)$: "S_1 can appear as the shape of a subpath σ_{sub} of a longer path σ with shape S_2". This approach failed because the relation R_{sp} is not even a partial order. In particular, $S_1 \cdot S_2 = S_1$, and $S_2 \cdot S_1 = S_2$, for the shapes S_1 and S_2 of figure 5.1.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. A general theory of translation. *Mathematical Systems Theory*, 3(3):193–221, 1969.

[2] R. Alur and P. Černý. Expressiveness of streaming string transducers. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, LIPIcs 8, pages 1–12, 2010.

[3] R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of 38th ACM Symposium on Principles of Programming Languages*, pages 599–610, 2011.

[4] R. Alur and L. D'Antoni. Streaming tree transducers. In *Automata, Languages, and Programming — 39th International Colloquium, ICALP Part II*, pages 42–53. 2012.

[5] R. Alur, L. D'Antoni, J. V. Deshmukh, M. Raghothaman, and Y. Yuan. Regular functions and cost register automata. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 13–22, 2013.

[6] R. Alur, E. Filiot, and A. Trivedi. Regular transformations of infinite strings. In *27th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 65–74, 2012.

[7] M. Bojanczyk. Transducers with origin information. In *Automata, Languages, and Programming — 41st International Colloquium, ICALP Part II*, 2014. Forthcoming.

[8] K. Chatterjee, L. Doyen, and T. A. Henzinger. Quantitative languages. *ACM Trans. Comput. Log.*, 11(4), 2010.

[9] M. Chytil and V. Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In *Automata, Languages and Programming — 4th International Colloquium*, LNCS 52, pages 135–147. 1977.

[10] T. Colcombet. The theory of stabilisation monoids and regular cost functions. In *Automata, Languages, and Programming — 36th International Colloquium, ICALP Part II*, pages 139–150. 2009.

[11] B. Courcelle. Monadic second-order graph transductions. In *17th Colloquium on Trees in Algebra and Programming*, LNCS 581, pages 124–144. 1992.

[12] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer, 2009.

[13] J. Engelfriet and H. J Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic*, 2(2):216–254, 2001.

[14] J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Information and Computation*, 154:34–91, 1999.

[15] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of 38th ACM Symposium on Principles of Programming Languages*, pages 317–330, 2011.

[16] E. M. Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. In *21st Annual Symposium on Foundations of Computer Science*, pages 83–85, 1980.

[17] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012.

[18] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of 39th ACM Symposium on Principles of Programming Languages*, pages 137–150, 2012.