Regular Combinators for String Transformations

Rajeev Alur, Adam Freilich, Mukund Raghothaman
University of Pennsylvania

This is the full version of the paper, and includes proofs omitted from the short version.

***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.**

I. 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 $(\mathbb{D}, +, 0)$. 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). As another interesting example of a numerical monoid, each value is a cost-discount pair,

and the (non-commutative) addition is the discounted sum operation. The traditional use of discounting in systems theory allows only discounting of *future* events, and corresponds to cost functions computed by classical one-way transducers, while regular functions allow more general forms of discounting (for instance, discounting of both past and future events).

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 $f \rhd g$ maps an input string σ to $f(\sigma)$, if this value is defined, and to $g(\sigma)$ otherwise; the *split sum* combinator $f \oplus g$ maps an input string σ to $f(\sigma_1) + f(\sigma_2)$ 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* $\sum f$ 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 $\sum f(\sigma)$ is $f(\sigma_1)+f(\sigma_2)+\cdots+f(\sigma_k)$, 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 $+$ 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 $+$ is not commutative, which is the case when the output values are strings themselves and addition 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* $f \overleftarrow{\oplus} g$ maps an input string σ to $g(\sigma_2) + f(\sigma_1)$ 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 is easy to show that regular functions are closed under these left-additive combinators.

The *sum* $f + g$ of two functions f and g maps a string σ to $f(\sigma) + g(\sigma)$. 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 [13], [5].

To motivate our final combinator, 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 $a^{m_2}b^{m_1}a^{m_3}b^{m_2}\ldots a^{m_k}b^{m_{k-1}}$. 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* $\sum(f, L)$ of σ is $f(\sigma_1\sigma_2)+f(\sigma_2\sigma_3)+\cdots+f(\sigma_{k-1}\sigma_k)$. 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 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 V 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.

II. Function Combinators

Let Σ be a finite alphabet, and $(\mathbb{D}, +, 0)$ 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 $d + \perp = \perp + d = \perp$, for all $d \in \mathbb{D}$. A *cost function* is a function $\Sigma^* \to \mathbb{D}_\perp$.

A. 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 $\perp(\sigma) = \perp$. $\perp$ can also be defined as the constant function $\emptyset/0$.

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 language.

B. Conditional choice and sum operators

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

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

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 $L/1 \rhd \Sigma^*/0$.

The *sum* $f+g$ is defined as $f+g\,(\sigma) = f(\sigma)+g(\sigma)$. 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* $f \oplus g\,(\sigma) = f(\sigma_1) + g(\sigma_2)$. Otherwise, $f \oplus g\,(\sigma) = \perp$. Over non-commutative monoids, this may be different from the *left-split sum* $f \overleftarrow{\oplus} g$: 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 $f \overleftarrow{\oplus} g\,(\sigma) = g(\sigma_2)+f(\sigma_1)$. Otherwise, $f \overleftarrow{\oplus} g\,(\sigma) = \perp$.

Observe that $\rhd$ is the analogue of union in regular expressions, with the important difference being that $\rhd$ is non-commutative. Similarly, $\oplus$ is similar to the concatenation operator of traditional regular expressions.

C. Iteration

The *iterated sum* $\sum f$ 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 $\sum f\,(\sigma) = f(\sigma_1) + f(\sigma_2) + \cdots + f(\sigma_k)$. Otherwise, $\sum f\,(\sigma) = \perp$. The *left-iterated sum* $\overleftarrow{\sum} f$ 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 $\overleftarrow{\sum} f\,(\sigma) = f(\sigma_k)+f(\sigma_{k-1})+\cdots+f(\sigma_1)$. Otherwise, $\overleftarrow{\sum} f\,(\sigma) = \perp$. The *reverse combinator* f^{rev} is defined as $f^{rev}(\sigma) = f(\sigma^{rev})$. 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 $\sum(a/1 \rhd b/0)$. The identity function $id : \Sigma^* \to \Sigma^*$ is given by the function expression $\sum(a \rhd b)$. The function *copy* which maps an input σ to $\sigma\sigma$ is then given by the expression $id + id$. On the other hand, the expression $\overleftarrow{\sum}(a \rhd b)$ is the function which reverses its input: $\overleftarrow{\sum}(a \rhd b)(\sigma) = \sigma^{rev}$ 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 $m = (\sum C/2) \rhd ((\sum C/1) \oplus S/0 \oplus \sum(C/1 \rhd S/0))$ maps the purchases of a month to the customer's debt. The first sub-

expression – $\sum C/2$ – computes the amount provided no survey is filled out and the second sub-expression – $(\sum C/1) \oplus S/0 \oplus \sum (C/1 \rhd S/0)$ – is defined provided at least one survey is filled out, and in that case, charges \$1 for each cup. The expression $coffee = \sum (m \oplus \#/0) \oplus m$ 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. *swap* can be expressed by the function expression $\left(\{a, b\}^* \#/\epsilon \oplus \sum (a \rhd b)\right) + \Sigma^*/\# + \left(\sum (a \rhd b) \oplus \#\{a, b\}^*/\epsilon\right)$. The first subexpression skips the first part of the string – $\{a, b\}^* \#/\epsilon$ – and echoes the second part – $\sum (a \rhd b)$. 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 given its full path, or in processing website URLs. This function is represented by the expression $id \oplus \#\{a, b\}^*/\epsilon$.

From the appropriate definitions, we have:

Proposition 7. *Over all monoids* $(\mathbb{D}, +, 0)$*, the following identity holds:* $\overleftarrow{\sum} f(\sigma) = \left(\sum (f^{rev})\right)^{rev}(\sigma)$.

D. 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* $\sum (f, L)(\sigma) = f(\sigma_1\sigma_2) + f(\sigma_2\sigma_3) + \cdots + f(\sigma_{k-1}\sigma_k)$. Otherwise, $\sum (f, L)(\sigma) = \bot$. Similarly, if there exist unique strings σ_1, σ_2, …, σ_k such that $k \geq 2$ and for all i, $\sigma_i \in L$, then the *left-chained sum* $\overleftarrow{\sum} (f, L)(\sigma) = f(\sigma_{k-1}\sigma_k) + f(\sigma_{k-2}\sigma_{k-1}) + \cdots + f(\sigma_1\sigma_2)$. Otherwise, $\overleftarrow{\sum} (f, L)(\sigma) = \bot$.

Example 8. 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, $shuffle(\sigma) = \bot$. See figure II.1a.

We first divide σ into chunks of text 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 $shuffle = \sum (f, L)$, where $f = (\sum a/b \oplus b/\epsilon) \overleftarrow{\oplus} (\sum a/a \oplus b/\epsilon)$.

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 26.

E. Function composition

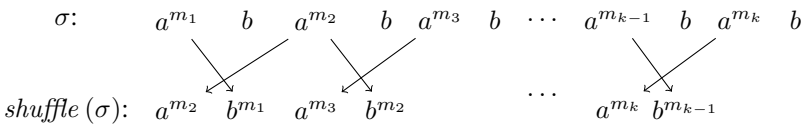
Let $f : \Sigma^* \to \Gamma^*_\bot$ and $g : \Gamma^* \to \mathbb{D}$ be two cost functions. The *composition* $g \circ f$ is defined as $g \circ f(\sigma) = g(f(\sigma))$, if $f(\sigma)$ and $g(f(\sigma))$ are defined, and $g \circ f(\sigma) = \bot$ otherwise.

Example 9. Composition is an alternative to chained sum for expressive completeness. Let $copy_L = (\sum a \oplus b) + (\sum a \oplus b)$ be the function which accepts strings from L and repeats them twice. The first step of the transformation is therefore the expression $\sum copy_L$. We then drop the first copy of P_1 and the last copy of P_k – this is achieved by the expression $drop = L/\epsilon \oplus id \oplus L/\epsilon$. The function $ensurelen = id + \Sigma^+/\epsilon$ 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. $f = (\sum a/b \oplus b/\epsilon) \overleftarrow{\oplus} (\sum a/a \oplus b/\epsilon)$. Thus, $shuffle = f \circ ensurelen \circ drop \circ \sum copy_L$.

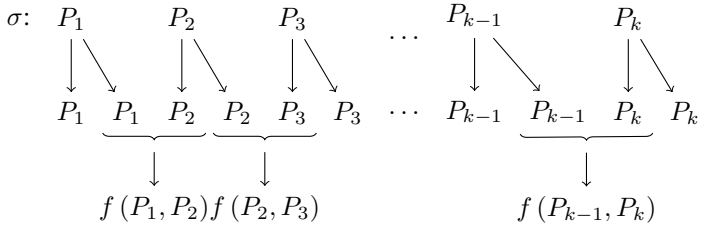
Observe that the approach in example 9 can be used to express the chained sum operation itself in terms of composition. Pick a symbol $@ \notin \Sigma$, and extend f to $(\Sigma \cup \{@\})^* \to \mathbb{D}$ by defining $f(\sigma) = \bot$ whenever σ contains an occurrence of @. Let id be the identity function for strings over Σ, and $copy_L$ be that function which maps strings $\sigma \in L$ to $\sigma@\sigma@$, and undefined otherwise. $copy_L = (id \oplus \epsilon/@) + (id \oplus \epsilon/@)$. Let $drop_L$ be $L@/\epsilon \oplus \sum (id \oplus @/\epsilon \oplus id \oplus @/@) \oplus L@/\epsilon$. Therefore, given a string σ uniquely decomposed as $\sigma = \sigma_1\sigma_2\ldots\sigma_k$, where for each i, $\sigma_i \in L$, $drop_L \circ \sum copy_L$ maps it to $\sigma_1\sigma_2@\sigma_2\sigma_3@\ldots\sigma_{k-1}\sigma_k@$. We then have the following:

Proposition 10. *For each cost function* f*, language* $L \subseteq \Sigma^*$*, and string* $\sigma \in \Sigma^*$*,*

1) $\sum (f, L)(\sigma) = \sum (f \oplus @/\epsilon) \circ ensurelen \circ drop_L \circ \sum copy_L(\sigma)$*, and*
2) $\overleftarrow{\sum} (f, L)(\sigma) = \overleftarrow{\sum} (f \oplus @/\epsilon) \circ ensurelen \circ drop_L \circ \sum copy_L(\sigma)$.



(a) Definition of *shuffle* (σ).



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

Figure II.1: Defining and expressing *shuffle* (σ) using function combinators.

III. 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. 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 $v := u + v + d$ and $v := d + v$, 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 II are exactly the class of regular functions. In this section, we formally define CCRAs, and show that every function expression represents a regular function.

A. Cost register automata

Definition 11. 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 12 (Copyless CRA [5])**.** A *CCRA* is a tuple $M = (Q, \Sigma, V, \delta, \mu, q_0, F, \nu)$, where

1) Q is a finite set of states,
2) Σ is a finite input alphabet,
3) V is a finite set of registers,
4) $\delta : Q \times \Sigma \to Q$ is the state transition function,
5) $\mu : Q \times \Sigma \times V \to (V \cup \mathbb{D})^*$ is the register update function such that for all q and a, the partial application $\mu(q, a) : V \to V^*$ is a copyless function over V,
6) $q_0 \in Q$ is the initial state,
7) $F \subseteq Q$ is the set of final states, and
8) $\nu : F \to (V \cup \mathbb{D})^*$ 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 $\gamma_0 = (q_0, val_0)$, where $val_0(v) = 0$, 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 $val(v_1 v_2 \dots v_k) = val(v_1) + val(v_2) + \cdots + val(v_k)$. 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 $\gamma_0 \to^\sigma (q_f, val_f)$. If $q_f \in F$, then $[\![M]\!](\sigma) = val(\nu(q_f))$. Otherwise, $[\![M]\!](\sigma) = \perp$.

A cost function is *regular* if it can be computed by a CCRA. A streaming string transducer is a CCRA where

the range $\mathbb{D}$ is the set of strings Γ^* over the output alphabet under concatenation.

Example 13. We present an example of an SST in figure III.1. The machine computes the function *shuffle* from example 8. 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.

B. 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 25, where we show that 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 "$u := v + d$", for some registers u and v and some constant d.

Definition 14 (Additive CRA)**.** An *additive cost register automaton (ACRA)* is a tuple $M = (Q, \Sigma, V, \delta, \mu, q_0, F, \nu)$, where

1) Q is a finite set of states,
2) Σ is a finite input alphabet,
3) V is a finite set of registers,
4) $\delta : Q \times \Sigma \to Q$ is the state transition function,
5) $\mu : Q \times \Sigma \times V \to V \times \mathbb{D}$ is the register update function,
6) $q_0 \in Q$ is the initial state,
7) $F \subseteq Q$ is the set of final states, and
8) $\nu : F \to V \times \mathbb{D}$ is the output function.

The semantics of ACRAs are also specified using configurations. The initial configuration $\gamma_0 = (q_0, val_0)$ maps all registers to 0. If the machine is in a configuration $\gamma = (q, val)$, and reads a symbol a, then it transitions to the configuration $\gamma' = \left(q', val'\right)$, written as $\gamma \to^a \gamma'$, where

1) $q' = \delta(q, a)$, and
2) for each register u, if $\mu(q, a, u) = (v, d)$, then $val'(u) = val'(v) + d$.

We then define the function $[\![M]\!]$ computed by M as follows. On input $\sigma \in \Sigma^*$, if $\gamma_0 \to^\sigma (q_f, val_f)$, and $q_f \in F$, then $[\![M]\!](\sigma) = val_f(\nu(q_f))$. Otherwise, $[\![M]\!](\sigma) = \perp$.

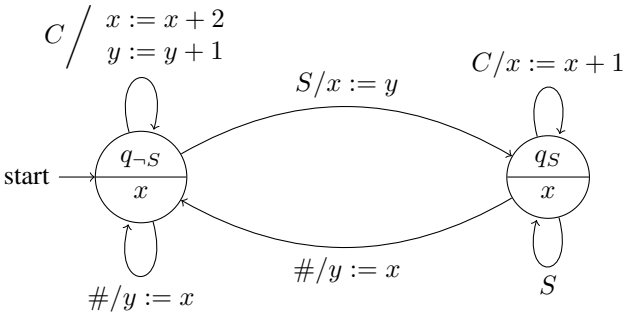


Figure III.2: ACRA computing *coffee*.

Example 15. In figure III.2, we present an ACRA 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.

C. Regular look-ahead

An important property of regular functions is that they are closed under *regular look-ahead* [3]: a CCRA can make transitions based not simply on the next symbol of the input, but on regular properties of the as-yet-unseen suffix. To formalize this, we introduce the notion of a *look-ahead labelling*. Let $\sigma = \sigma_1\sigma_2 \ldots \sigma_n \in \Sigma^*$ be a string, and $A = (Q, \Sigma, \delta, q_0)$ be a DFA over Σ. Starting in state q_0, and reading σ in reverse, say A visits the sequence of states $q_0 \to^{\sigma_n} q_1 \to^{\sigma_{n-1}} q_2 \to^{\sigma_{n-2}} \cdots \to^{\sigma_1} q_n$. Then, the state of A at position i, q_i determines a regular property of the suffix $\sigma_{n-i+1}\sigma_{n-i+2} \ldots \sigma_n$. We term the string of states $q_n q_{n-1} \ldots q_0$ the labelling of σ by the *look-ahead automaton* A.

Proposition 16. *Let A be a look-ahead automaton over Σ, and let M be a CCRA over labellings in Q^*. Then, there is a CCRA machine M' over Σ, such that for every $\sigma \in \Sigma^*$, $[\![M']\!](\sigma) = [\![M]\!](lab(\sigma))$ where $lab(\sigma)$ is the labelling of σ by A.*

D. From function expressions to cost register automata

Theorem 17. *Every cost function expressible using the base functions combined using the $\rhd$, $+$, $\oplus$, $\overleftarrow{\oplus}$, $\sum$, $\overleftarrow{\sum}$, 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. We now prove each

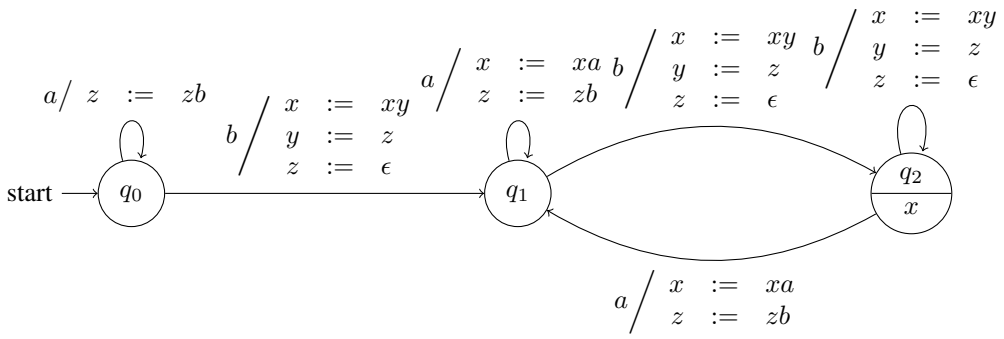


Figure III.1: Streaming string transducer computing *shuffle*. q_2 is the only accepting state. The annotation "x" in state q_2 specifies the output function. On each transition, registers whose updates are not specified are left unchanged.

case as a separate lemma, and these together establish the present theorem.

Lemma 18. *For all regular languages $L \subseteq \Sigma^*$, and $d \in \mathbb{D}$, L/d is a regular function.*

Proof: Consider the DFA $A = (Q, \Sigma, \delta, q_0, F)$ accepting L, and construct the machine $M = (Q, \Sigma, \emptyset, \delta, \mu, q_0, F, \nu)$, where $\nu(q) = d$, for all $q \in F$. This machine has the same state space as A, but does not maintain any registers. In every final state, the machine outputs the constant $d \in \mathbb{D}$. The domain of the register update function μ is empty, and so we do not specify it. Clearly, $[\![M]\!](\sigma) = L/d\,(\sigma) = d$, for each σ,and it follows that L/d is a regular function. ∎

Lemma 19. *Whenever f and g are regular functions, $f \rhd g$ and $f + g$ are also regular.*

Proof: Let f and g be computed by the CCRAs $M_f = (Q_f, \Sigma, V_f, \delta_f, \mu_f, q_{0f}, F_f, \nu_f)$ and $M_g = (Q_g, \Sigma, V_g, \delta_g, \mu_g, q_{0g}, F_g, \nu_g)$ respectively. We use the product construction to create the machines $M_{f \rhd g}$ and M_{f+g} that compute $f \rhd g$ and $f + g$ respectively. The idea is to run both machines in parallel, and in the case of $M_{f \rhd g}$, output depending on which machines are in accepting states. In M_{f+g}, we output only if both machines are accepting, and then output the sum of the outputs of both machines.

Assume, without loss of generality, that $V_f \cap V_g = \emptyset$. Define $M_{f \rhd g} = (Q_f \times Q_g, \Sigma, V_f \cup V_g, \delta, \mu, (q_{0f}, q_{0g}), F_{f \rhd g}, \nu_{f \rhd g})$ and $M_{f+g} = (Q_f \times Q_g, \Sigma, V_f \cup V_g, \delta, \mu, (q_{0f}, q_{0g}), F_{f+g}, \nu_{f+g})$, where

1) for each q_1, q_2 and a, $\delta((q_1, q_2), a) = (\delta_f(q_1, a), \delta_g(q_2, a))$,
2) if $v \in V_f$, then $\mu((q_1, q_2), a, v) = \mu_f(q_1, a, v)$, and otherwise, $\mu((q_1, q_2), a, v) = \mu_g(q_2, a, v)$,
3) $F_{f \rhd g} = F_f \times Q_g \cup Q_f \times F_g$, and $F_{f+g} = F_f \times F_g$,
4) for all $(q_1, q_2) \in F_{f \rhd g}$, if $q_1 \in F_f$, then $\nu(q_1, q_2) = \nu_f(q_1)$, and otherwise $\nu(q_1, q_2) = \nu_g(q_2)$, and
5) for all $(q_1, q_2) \in F_{f+g}$, $\nu(q_1, q_2) = \nu_f(q_1) + \nu_g(q_2)$.

Since the sets of registers are disjoint, observe that the register updates and output functions just defined are copyless. It follows that $M_{f \rhd g}$ and M_{f+g} compute $f \rhd g$ and $f + g$ respectively. ∎

Lemma 20. *Whenever f and g are regular functions, $f \oplus g$ and $f \overleftarrow{\oplus} g$ are also regular.*

Proof: Let f and g be computed by the CCRAs $M_f = (Q_f, \Sigma, V_f, \delta_f, \mu_f, q_{0f}, F_f, \nu_f)$ and $M_g = (Q_g, \Sigma, V_g, \delta_g, \mu_g, q_{0g}, F_g, \nu_g)$ respectively. We recall that the domain $L \subseteq \Sigma^*$ over which a regular function is defined is a regular language. Let L_f and L_g be the domains of f and g respectively. The idea is to use regular lookahead and execute M_f on the prefix $\sigma_1 \in L_f$, and when the lookahead automaton indicates that the suffix $\sigma_2 \in L_g$, we switch to executing M_g, and combine the results in the output function.

Let A_1 be a lookahead automaton with state space $\Sigma \cup \{q_{01}\}$, so that the state of A_1 indicates the next symbol of the input. Let A_2 be a lookahead automaton which accepts strings σ such that $\sigma^{rev} \in L_g$, and let F_2 be the set of its accepting states. The combined lookahead automaton is the product $A_1 \times A_2$, such that the state (a, q) of this product indicates the next symbol in the input, and depending on whether $q \in F_2$, whether the suffix $\sigma_2 \in L_g$.

Let A_3 (with accepting states F_3) be a DFA, which on input σ, determines whether σ can be unambiguously split as $\sigma = \sigma_1\sigma_2$, with $\sigma_1 \in L_f$ and $\sigma_2 \in L_g$. Construct the machine $M = ((Q_f \cup Q_g) \times A_3, Q_1 \times Q_2, V_f \cup V_g \cup \{total\}, \delta, \mu, (q_{0f}, q_{03}), F_g \times F_3, \nu)$, where δ, μ, and ν operate as follows:

1) In a state $(q_1, q_3) \in Q_f \times A_3$, on reading the input symbol (a, q_l), where $q_1 \notin F_f$ or $q_l \notin F_2$, the machine transitions to $(\delta_f(q_1, a), \delta_3(q_3, a))$. The registers of M_f are updated according to μ_f, and the other registers are left unchanged.
2) In a state $(q_1, q_3) \in Q_f \times A_3$, on reading the input symbol (a, q_l), where $q_1 \in F_f$ and $q_l \in F_2$, the machine transitions to $(q_{0g}, \delta_3(q_3, a))$. The machine stores the output of M_f in the register $total$, and the other registers are left unchanged.
3) In the state $(q_2, q_3) \in Q_g \times A_3$, on reading the input symbol (a, q_l), the machine transitions to $(\delta_g(q_2, a), \delta_3(q_3, a))$. The registers of M_g are updated according to μ_g, and the other registers are left unchanged.
4) In the final state $(q_2, q_{3f}) \in F_g \times F_3$, the machine outputs the value $total + \nu(q_2)$.

The machine M just constructed computes the function $f \oplus g$ using regular lookahead, and it follows that $f \oplus g$ is regular. Similarly, it can be shown that $f \overleftarrow{\oplus} g$ is also regular. ∎

Along similar lines, we have:

Lemma 21. *Whenever f is a regular function, $\sum f$ and $\overleftarrow{\sum} f$ are also regular.*

Proof: The main difference between this and the construction of lemma 20 are the following: the state space Q of M is defined as $Q = Q_f \times A_3$, since there is only one CCRA M_f. The set of registers is $V = V \cup \{total\}$, and the accepting states $F = F_f \times F_3$.

In a state $(q_1, q_3) \in Q_f \times A$, on reading the input symbol (a, q_l), where $q_1 \in F_f$, and $q_l \in F_2$, the machine transitions back to $(q_{0f}, \delta_3(q_3, a))$. The machine appends the output of M_f to the right of the register $total$, and all other registers are cleared to 0.

The machine thus constructed computes $\sum f$. If the machine were to append the output of M_f to the left of $total$, then it would compute $\overleftarrow{\sum} f$. Thus, both function expressions are regular. ∎

The next lemma was first proved in [5]. It can also be seen as a consequence of lemma 23, because for all σ, $f^{rev}(\sigma) = f \circ reverse(\sigma)$, where $reverse = \overleftarrow{\sum} \rhd \{a/a \mid a \in \Sigma\}$ is the function which reverses its input.

Lemma 22. *Whenever f is a regular function, so is f^{rev}.*

Lemma 23. *Whenever $f : \Gamma^* \to \mathbb{D}$ and $g : \Sigma^* \to \Gamma^*$ are regular functions, $f \circ g$ is also a regular function.*

Proof: Since SSTs are closed under composition, if $f : \Gamma^* \to \mathbb{D}$ and $g : \Sigma^* \to \Gamma^*$ are regular functions, it follows that $f \circ g$ is also a regular function. ∎

Lemma 24. *Whenever f is a regular function, and $L \subseteq \Sigma^*$ is a regular language, $\sum(f, L)$ and $\overleftarrow{\sum}(f, L)$ are also regular functions.*

Proof: From proposition 10 and lemma 23. ∎

This completes the proof of theorem 17.

IV. Completeness of Combinators for Commutative Monoids

In this section, we show that if $\mathbb{D}$ is a commutative monoid, then constant functions combined using the choice, split sum and function iteration are expressively equivalent to the class of regular functions. Consider the ACRA M shown in figure IV.1a. The idea is to view M as a non-deterministic automaton A over the set of vertices $Q \times V$: for every path $\pi = q_0 \to^{\sigma_1} q_1 \to^{\sigma_2} \cdots \to^{\sigma_n} q_n$ through the ACRA, there is a corresponding path through A, $\pi_A = (q_0, v_0) \to^{\sigma_1} (q_1, v_1) \to^{\sigma_2} \cdots \to^{\sigma_n} (q_n, v_n)$, where v_n is the register which is output in the final state q_n, and at each position i, v_i indicates the register whose current value flows into the final value of v_n. Observe that this NFA A is unambiguous – for every string σ that is accepted by A, there is a unique accepting path. Furthermore, the final value of register v_n is simply the sum of the increments accumulated along each transition of this accepting path. Therefore, if the label $(q, v) \to^{a_d} (q', v')$ along each edge is also annotated with the increment value d, so that the update expression reads $\mu(q, a, v') = v + d$, then the regular expression for the language accepted A – $(a_1 + b_0)^* + (a_1 + b_1 + e_1)^* e_1 (a_1 + b_0)^*$ – can be alternatively viewed as a function expression for $[\![M]\!]$ – $\sum(b/0 \rhd a/1) \rhd (\sum(b/1 \rhd a/1 \rhd e/1) \oplus e_1 \oplus \sum(b/0 \rhd a/1))$.

Theorem 25. *If $(\mathbb{D}, +, 0)$ is a commutative monoid, then every regular function $f : \Sigma^* \to \mathbb{D}$ can be expressed using the base functions combined with the choice, split sum and iterated sum operators.*

Proof: We need to show an arbitrary ACRA $M = (Q, \Sigma, V, \delta, \mu, q_0, F, \nu)$ can be expressed by these combinators.

We construct an NFA A with states $Q \times V$ and an alphabet Γ consisting of a finite subset of $\Sigma \times \mathbb{D}$ which are those elements (a, d) such that for some state $q \in Q$ and two registers $v, v' \in V$ there is an update $\mu(q, a, v) = v' + d$. We will denote (a, d) as a_d.

We define the transition relation as follows: $(q', v') \in \delta'((q, v), a_d)$ iff $\mu(q, a, v') = v + d$ and $\delta(q, a) = q'$.

Assume without loss of generality that our output function takes values in V. The start states of the NFA

A are all states in $\{q_0\} \times V$ and the final states are $\{(q, v) \mid \nu(q) = v\}$.

Consider any unambiguous regular expression of strings accepted by the NFA A: interpret regular expression union $\cup$ as $\triangleright$, regular expression concatenation $\cdot$ as $\oplus$, Kleene-* as the iterated sum $\sum f$ and input symbols a_d as the constant functions a/d.

It can be shown by an inductive argument that the regular expression corresponding to paths in our NFA from (q, v) to (q', v'), when interpreted as a regular function f is defined exactly on those $\sigma \in \Sigma^*$ such that σ is a path from q to q' with the effect that v flows into v'. Moreover, the total effect of this path σ is $v' := v + f(\sigma)$ for all of these σ. It follows that a function expression for $[\![M]\!]$ can be obtained from the union of the unambiguous regular expressions from some state in $\{q_0\} \times V$ to some state in $\{(q, v) \mid \nu(q) = v\}$. ∎

V. Completeness of Combinators for General Monoids

In this section, we describe an algorithm to express every regular function $f : \Sigma^* \to \mathbb{D}$ as a function expression. To simplify the presentation, we prove theorem 26 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 26 can then be used to convert this SST to function expression form, which when interpreted over the original domain $\mathbb{D}$ represents $[\![M]\!]$.

Theorem 26. *For an arbitrary finite alphabet Σ and monoid $(\mathbb{D}, +, 0)$, every regular function $f : \Sigma^* \to \mathbb{D}$ can be expressed using the base functions combined with choice, sum, split sum, iterated sum, chained sum, and their left-additive versions.*

A. From DFAs to regular expressions: A review

The procedure to convert a CCRA into a function expression is similar to the corresponding algorithm [17] that transforms a DFA $A = (Q, \Sigma, \delta, q_0, F)$ into an equivalent regular expression; we will also use this algorithm in our correctness proof – hence this review.

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 strings σ from q to q', while only passing through the intermediate states $\{q_1, q_2, \ldots, q_i\}$. This can be inductively constructed as follows:

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

The language L accepted by A is then given by the regular expression $\sum_{q_f \in F} r^{(n)}(q_0, q_f)$. Note that the regular expression thus obtained is also unambiguous.

B. A theory of shapes

In a CCRA M, the effect of processing a string σ starting from a state q can be summarized by the pair $(\delta(q, \sigma), \mu(q, \sigma))$ – $\delta(q, \sigma)$ is the state of the machine after processing σ, and the partial application of the register update function $\mu(q, \sigma) : V \to (V \cup \Gamma)^*$ expresses the final values of the registers in terms of their initial ones.

Consider the expression $\mu(q, \sigma, u) = aubcvd$, where $u, v \in V$ are registers, and $a, b, c, d \in \Gamma^*$ are string constants. Because of the associative property, any update expression can be equivalently represented – as in $\mu(q, \sigma, u) = aub'vd$ where $b' = bc$ – so that there is at most one string constant between consecutive registers in this update expression. The summary for σ therefore contains the shape $S_\sigma : V \to V^*$ indicating the sequence of registers in each update expression and, for each register v and each position k from $1, 2, \ldots, |S_\sigma(v)| + 1$, a string $\gamma_k \in \Gamma^*$ indicating the k^{th} string constant appearing in $\mu(q, \sigma, u)$.

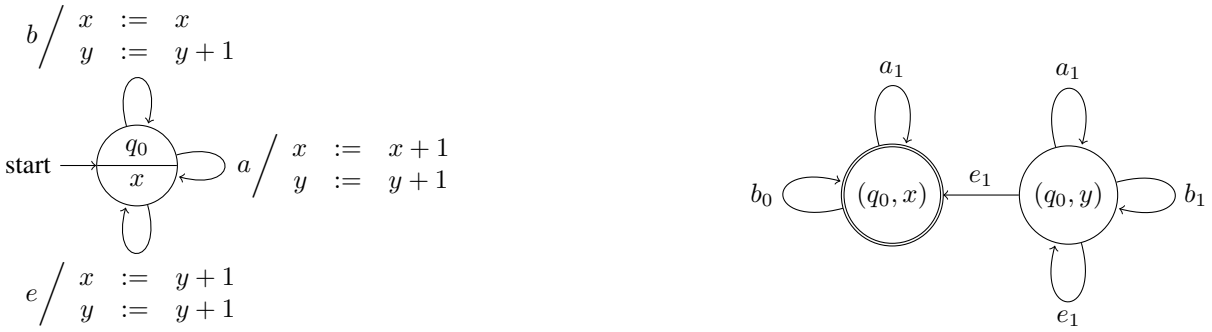
Definition 27 (Shape of a path)**.** A *shape* $S : V \to V^*$ is a copyless function over a finite set of registers V. Let $\pi = q_1 \to^{\sigma_1} q_2 \to^{\sigma_2} \to \cdots \to^{\sigma_n} q_{n+1}$ be a path through a CCRA M. The *shape of the path* π is the function $S_\pi : V \to V^*$ such that for all registers $v \in V$, $S_\pi(v)$ is the string projection onto V of the register update expression $\mu(q_1, \sigma, v)$: $S_\pi(v) = \pi_V(\mu(q_1, \sigma, v))$.

We refer to a string constant in the update expression as a *patch* in the corresponding shape. Because of the copyless restriction on the register update function, the set of all shapes over V is finite.

The following is an immediate consequence of the space of shapes being finite:

Proposition 28. *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.*

Example 29. It is helpful to visualize shapes as bipartite graphs (figure V.1), though this representation omits some important information about the shape. 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



(a) (b)

Figure IV.1: Translating an ACRA to the commutative calculus. The machine operates over the alphabet $\Sigma = \{a, b, e\}$, and when given a string $\sigma = \sigma_1 e \sigma_2 e \ldots \sigma_k$, where each $\sigma_i \in \{a, b\}^*$, it counts the number of a-s and e-s, but only counts those b-s which occur before the final e. Figure IV.1b is the NFA that results from the construction of theorem 25. Both states in the NFA are initial.

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 $S_2(v) = v_1 v_2 \ldots v_k$. Then $S_1 \cdot S_2(v) = s_1 s_2 \ldots s_k$, where $s_i = S_1(v_i)$. By definition, therefore,

Proposition 30. *Let π_1 and π_2 be two paths through a CCRA M such that the final state of π_1 is the same as the initial state of π_2. Then, for all registers v, $S_{\pi_1 \pi_2}(v) = S_{\pi_1} \cdot S_{\pi_2}(v)$.*

C. Proof outline

To summarize the effect of a set of paths with the same shape, we introduce the notion of an expression vector – for a shape S, an *expression vector* $\mathbf{A}$ 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 $\gamma_k \in \Gamma^*$ at position k is given by $\mathbf{A}_{v,k}(\sigma)$.

Example 31. Consider the loop a^* at the state q_1 in the SST of figure III.1. Consider some 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: $\mathbf{A}_{x,1} = \mathbf{A}_{y,1} = \mathbf{A}_{y,2} = \mathbf{A}_{z,1} = \Sigma^*/\epsilon$, $\mathbf{A}_{x,2} = \sum a/a$, and $\mathbf{A}_{z,2} = \sum a/b$. Then $\mathbf{A}$ summarizes the set of paths a^* at the state q_1.

The outer loop of our algorithm is an iteration which proceeds in lock-step with the DFA-to-regular expression translator. In step i, for each pair of states $q, q' \in Q$, and shape S, we maintain an expression vector $\mathbf{R}_S^{(i)}(q, q')$. The invariant maintained is that $\mathbf{R}_S^{(i)}(q, q')$ summarizes all paths $\sigma \in r^{(i)}(q, q')$ with shape S.

After this iteration is complete, pick a state $q_f \in F$, a shape S, and some register v. Construct the function expression $f_{S,v} = \mathbf{R}_{S,v,1}^{(n)}(q_0, q_f) + \mathbf{R}_{S,v,2}^{(n)}(q_0, q_f) + \cdots + \mathbf{R}_{S,v,|S(v)|+1}^{(n)}(q_0, q_f)$. Because v initially held the empty string ϵ, it follows that for each path $q_0 \to^\sigma q_f$ with shape S, the final value in the register v is given by $f_{S,v}(\sigma)$. We will then have constructed a function expression equivalent to the given CCRA M.

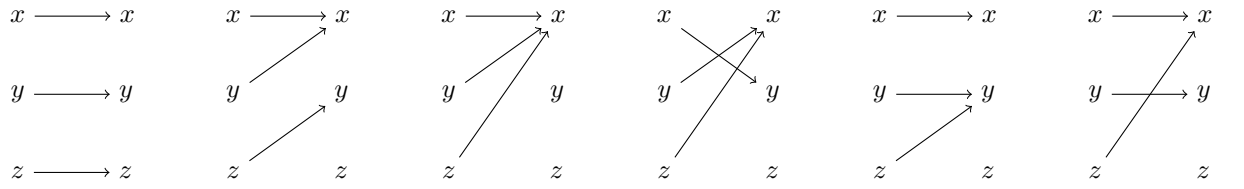
There are therefore two steps in this construction:

1) Construct $\mathbf{R}_S^{(0)}(q, q')$, for each pair of registers $q, q' \in Q$, and shape S.
2) For each $1 \leq i < n$, $0 \leq j \leq i$, and for all shapes S, and pairs of states $q, q' \in Q$, given $\mathbf{R}_S^{(j)}(q, q')$, construct $\mathbf{R}_S^{(i+1)}(q, q')$.

D. Operations on expression vectors

In this subsection, we create a library of basic operations on expression vectors, including concatenation and union.

1) Restricting expression 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. We would want to restrict the component expressions in a vector so that they all have the same domain – given a cost function



(a) $q_1 \to^a q_1, S_\perp$. (b) $q_1 \to^b q_2$. (c) $q_1 \to^b q_2 \to^b q_2, S_\top$. (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 V.1: Visualizing shapes as bipartite graphs. Figures V.1a-V.1c describe the shapes of some paths in the earlier SST example of figure III.1.

$f : \Sigma^* \to \Gamma^*$ and a language $L \subseteq \Sigma^*$, we define the *restriction of f to L* as $f \cap L = f + L/\epsilon$. This is equivalent to saying that $f \cap L(\sigma) = f(\sigma)$, if $\sigma \in L$, and $f \cap L(\sigma) = \perp$, 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 $\mathbf{A}_{v,k} \cap L$.

2) 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 in $\mathrm{Dom}(f) \cdot L$, and applies f to the prefix and ignores the suffix, provided the split is unique, i.e. $f \ll L = f \oplus L/\epsilon$. Similarly, the *right-shifted function* $f \gg L = L/\epsilon \oplus f$. 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 $(\mathbf{A} \gg L)_{v,k} = \mathbf{A}_{v,k} \cap \mathrm{Dom}(\mathbf{A}) \gg L$.

3) Concatenation : Let L be a set of paths with shape S, and L' be a set of paths with shape S'. Let the expression vectors $\mathbf{A}$ and $\mathbf{B}$ summarize paths in L and L' respectively. We now construct an expression vector $\mathbf{A} \cdot \mathbf{B}$ which summarizes unambiguous paths in $L \cdot L'$.

Consider a path $\pi \in L \cdot L'$ which can be unambiguously decomposed as $\pi = \pi_1 \pi_2$ with $\pi_1 \in L$ and $\pi_2 \in L'$. When applying $\mathbf{A}$ (resp. $\mathbf{B}$) to this path, we should shift the expression vector to examine only π_1 (resp. π_2). Thus, define $\mathbf{A}' = \mathbf{A} \ll \mathrm{Dom}(\mathbf{B})$, and $\mathbf{B}' = \mathbf{B} \gg \mathrm{Dom}(\mathbf{A})$.

Pick a register v, and let $v := f_1 v_1 f_2 v_2 \ldots v_k f_{k+1}$ be the update expression for v in $\mathbf{B}'$. For each register v_i in the right-hand side, let $v_i := f_{i1} v_{i1} f_{i2} v_{i2} \ldots v_{ik_i} f_{ik_i+1}$ be the update expression for v_i in $\mathbf{A}'$. View string concatenation as the function combinator $+$, and substitute the expression for each v_i in $\mathbf{A}'$ into the expression for v in $\mathbf{B}'$. Then, observe that $(\mathbf{A} \cdot \mathbf{B})_{v,k}$ is the k^{th} function expression in the string that results.

4) 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, define the choice $\mathbf{A} \rhd \mathbf{B}$ is the expression vector for shape S such that for each register v and patch k, $(\mathbf{A} \rhd \mathbf{B})_{v,k} = \mathbf{A}'_{v,k} \rhd \mathbf{B}'_{v,k}$.

Claim 32. 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 $\mathbf{A} \rhd \mathbf{B}$ summarizes paths in $L \cup L'$.

The notation $\rhd \{f_1, f_2, \ldots, f_k\}$ is shorthand for the expression $f_1 \rhd f_2 \rhd \cdots \rhd f_k$. We ensure that when this notation is used, the functions have mutually disjoint domains, so the order is immaterial.

E. Constructing $\mathbf{R}_S^{(0)}(q, q')$

For each string $a \in \Sigma \cup \{\epsilon\}$, and each pair of states $q, q' \in Q$ such that $q \to^a q'$, if S is the shape of the update expression of $q \to^a q'$, we define $\mathbf{R}_S^{(a)}(q, q')$ as follows. For each register v and patch k in $S(v)$, $\mathbf{R}_{S,v,k}^{(a)}(q, q') = a/\gamma_{v,k}$, where $\gamma_{v,k}$ is the k^{th} string constant appearing in the update expression $\mu(q, a, v)$. For all other $a \in \Sigma \cup \{\epsilon\}$, $q, q' \in Q$, and shapes S, define $\mathbf{R}_S^{(a)}(q, q') = \perp$. Finally, $\mathbf{R}_S^{(0)}(q, q') = \rhd \{\mathbf{R}_S^{(a)}(q, q') \mid a \in \Sigma \cup \{\epsilon\}\}$. By construction,

Claim 33. For each pair of states $q, q' \in Q$ and shape S, $\mathbf{R}_S^{(0)}(q, q')$ summarizes all paths $\sigma \in r^{(0)}(q, q')$ from q to q' with shape S.

F. 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. Register values may flow in complicated ways: consider for example the shape in figure V.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 34. 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 $\preceq$.

For example, the shapes in figures V.1a, V.1c, V.1e, and V.1f are normalized, while V.1b and V.1d are not. Informally, the first condition requires that all registers in the CCRA flow upward, and the second ensures that shapes are idempotent. Observe that if the individual transitions in a path are normalized, then the whole path is itself normalized.

Proposition 35. *For every CCRA M, there is an equivalent normalized CCRA M'.*

Proof: Let $M = (Q, \Sigma, V, \delta, \mu, q_0, F, \nu)$. Let $V' = \{x_i \mid 0 \leq i \leq |V|\}$ (so that $|V'| = |V| + 1$), and define the register ordering as $x_i \preceq x_j$ iff $i \leq j$. x_0 is a sink register which accumulates all those register values which are lost during computation. Let Q' be the set of all those pairs (q, f), where $q \in Q$ is the current state, and the permutation $f : V \to V' \setminus \{x_0\}$ is the register renaming function. For simplicity, let us extend each register renaming function f to $V \cup \Gamma \to V' \cup \Gamma$ by defining $f(\gamma) = \gamma$, for $\gamma \in \Gamma$. We further extend it to $(V \cup \Gamma)^* \to (V' \cup \Gamma)^*$ by $f(v_1 v_2 \dots v_k) = f(v_1) f(v_2) \dots f(v_k)$. Let $F' = \{(q, f) \mid q \in F\}$, and define the output function ν' as $\nu'(q, f) = f(\nu(q))$.

For each state $(q, f) \in Q'$, and each symbol $a \in \Sigma$, define f' as follows. For each register $v \in V$, if at least one register occurs in $\mu(q, a, v)$, then $f'(v) = \min\{f(u) \mid u \text{ occurs in } \mu(q, a, v)\}$. Observe that, because of the copyless restriction, for every pair of distinct registers $u, v \in V$, $f'(u) \neq f'(v)$. For all registers v such that $f'(v)$ is still undefined, define $f'(v)$ arbitrarily such that f' is a permutation. Now $\delta'((q, f), a) = (\delta(q, a), f')$.

Define $\mu'((q, f), a, x_0) = x_0 + f(v_1) + f(v_2) + \dots + f(v_k)$, where $\{v_1, v_2, \dots, v_k\}$ is the set of registers in M whose value is lost during the transition. For all registers $v \in V$, if $\mu(q, a, v) = v_1 v_2 \dots v_k \in (V \cup \Gamma)^*$, define $\mu'((q, f), a, f'(v)) = f(v_1) + f(v_2) + \dots + f(v_k)$.

For an arbitrary ordering $v_1 \leq v_2 \leq \dots \leq v_{|V|}$ of the original registers V, define $f_0(v_i) = x_i$. It can be shown that the CCRA $M' = (Q', \Sigma, V', \delta', \mu', (q_0, f_0), F', \nu')$ is equivalent to M, and that its transitions are normalized. ∎

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

G. 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 V.1. Let π be a path through the CCRA with shape S_1. Then, no sub-path of π 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 36. 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 $S_\bot$ from figure V.1 is the bottom element of $\sqsubseteq$, and $S_\top$ is the top element. $S_1 \sim S_2$, and both shapes are strictly sandwiched between $S_\bot$ and $S_\top$. Note that support-equality is a finer relation than incomparability[1] with respect to $\sqsubseteq$. In an early attempt to create a partial order over shapes, we considered formalizing the relation R_{sp}, "can appear as the shape of a sub-path". However, this approach fails because R_{sp} is not a partial order. In particular, observe that $S_1 \cdot S_2 = S_1$, and $S_2 \cdot S_1 = S_2$. Thus, both (S_1, S_2) and (S_2, S_1) occur in R_{sp}, and they are not equal. The presence of "crossing edges" in the visualization of S_2 is what complicates the construction, but we could not find a syntactic transformation on CCRAs that would eliminate these crossings.

Claim 37. Let π be a path through the CCRA M with shape S, and π' be a subpath of π with shape S'. If $S' \not\sqsubset S$, then $S' \sim S$.

Proof: Assume otherwise, so $S' \not\sim S$. Then, for some register $v \in supp(S)$, $v \notin S'$. The effect of the entire path π is to make the initial value of v flow into itself, but on the subpath π', v is promoted to some upper register v'. Because of the normalization condition (definition 34), it follows that on the suffix, the value in $v' \prec v$ cannot flow back into v, leading to a contradiction. ∎

Claim 38. Let π be a path through the CCRA M with shape S, and let π' be the shortest prefix with shape S' such that $S' \not\sqsubset S$. Then $S' = S$.

Proof: Assume otherwise. From claim 37, we know that $S' \sim S$.

[1]Note that incomparability with respect to $\sqsubseteq$ is not even an equivalence relation over shapes.

Case 1. For some register $u \notin supp(S)$, and registers $v, w \in supp(S)$, with $v \neq w$, $u \to v$ in S, and $u \to w$ in S'. Once u has flowed into w, the "superpath" cannot remove u from w. It is thus a contradiction that u flows into v in S.

Case 2. For some register $v \in supp(S)$, the order of registers in $S(v)$ and $S'(v)$ are different. For some registers u and w, u occurs before w in $S(v)$, and w occurs before u in $S'(v)$. However, once the values of w and u have been appended to v in the order wu, they cannot be separated to be recast in the order uw. It is thus a contradiction that u occurs before w in $S(v)$.

■

H. Kleene-* and revisiting states

At each step of the iteration, for each pair of states $q, q' \in Q$, and for each shape S, we construct a new expression vector $\mathbf{R}_S^{(i+1)}(q, q')$, summarizing paths in $r^{(i+1)}(q, q')$ with shape S. Recall that, from the DFA-to-regex translator, $r^{(i+1)}(q, q') = r^{(i)}(q, q') + r^{(i)}(q, q_{i+1}) r^{(i)}(q_{i+1}, q_{i+1})^* r^{(i)}(q_{i+1}, q')$.

Let $\mathbf{B}_S$ be an expression vector which summarizes paths in $r^{(i)}(q_{i+1}, q_{i+1})^*$ with shape S. We can then write $\mathbf{R}_S^{(i+1)}(q, q') = \mathbf{R}_S^{(i)}(q, q') \triangleright \mathbf{C}_S$, where $\mathbf{C}_S = \triangleright \{\mathbf{R}_{S_1}^{(i)}(q, q_{i+1}) \cdot \mathbf{B}_{S_2} \cdot \mathbf{R}_{S_3}^{(i)}(q_{i+1}, q') \mid S_1 \cdot S_2 \cdot S_3 = S\}$. Our goal is therefore to construct $\mathbf{B}_S$, for each S. We construct these expression vectors inductively, according to the partial order $\sqsubseteq$. The remaining subsections are devoted to expressing $\mathbf{B}_S$.

I. Decomposing loops

Consider any path σ in $r^{(i)}(q_{i+1}, q_{i+1})^*$ with shape S. From claims 37 and 38, we can unambiguously decompose $\sigma = \sigma_1 \sigma_2 \ldots \sigma_k \sigma_f$, where

1) each $\sigma_j \in r^{(i)}(q_{i+1}, q_{i+1})^*$ is a self-loop at q_{i+1},
2) for each j, $1 \leq j \leq k$, the shape S_j of σ_j is support-equal to S, $S_j \sim S$, and $S_f \sqsubset S$, and
3) for each j, $1 \leq j \leq k$, and for each proper prefix $\sigma_{pre} \in r^{(i)}(q_{i+1}, q_{i+1})^*$ of σ_j, $S_{pre} \sqsubset S$.

Let us call the split $\sigma = \sigma_1 \sigma_2 \ldots \sigma_k \sigma_f$ the *S-decomposition* of σ. See figure V.2.

Consider some shape $S' \sim S$, and let $L_{first}(S')$ be the set of all paths $\pi \in r^{(i)}(q_{i+1}, q_{i+1})^*$ with shape S' such that no proper prefix π_{pre} of π has shape $S_{pre} \sim S$. We can then unambiguously write $\pi = \pi_{pre} \pi_{last}$, with $\pi_{pre} \in r^{(i)}(q_{i+1}, q_{i+1})^*$, $\pi_{last} \in r^{(i)}(q_{i+1}, q_{i+1})$, and such that

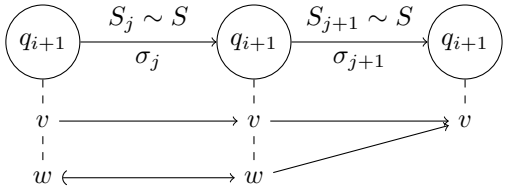


Figure V.3: For any path in $r^{(i)}(q_{i+1}, q_{i+1})^*$, inward flows into a (support) register v have to be from non-support registers.

$S_{pre} \sqsubset S$. Define $\mathbf{A}_{S'} = \triangleright \{\mathbf{R}_{S_{pre}}^{(i+1)}(q_{i+1}, q_{i+1}) \cdot \mathbf{R}_{S_{post}}^{(i)}(q_{i+1}, q_{i+1}) \mid S_{pre} \cdot S_{post} = S \text{ and } S_{pre} \sqsubset S\}$.

Claim 39. For all shapes $S' \sim S$, the expression vector $\mathbf{A}_{S'}$ summarizes all paths in $L_{first}(S')$.

J. Computing $\mathbf{B}_S$

We now construct the expression vector $\mathbf{B}_S$. Consider a path σ, and its S-decomposition $\sigma = \sigma_1 \sigma_2 \ldots \sigma_k \sigma_f$. Given a register v, and a patch $1 \leq k \leq |S(v)| + 1$, three cases may arise:

First, if $S(v) = \epsilon$, i.e. v is reset during the computation. v was reset while processing σ_k. Any registers flowing into it during this time were also reset by σ_k. Thus, its value is entirely determined entirely by σ_k and σ_f. First define $\mathbf{F} = \triangleright \{\mathbf{A}_{S_1} \cdot \mathbf{B}_{S_2} \mid S_1 \cdot S_2 = S \text{ and } S_2 \sqsubset S\}$, and let $L_f = \bigcup_{S' \sim S} L_{first}(S')$. Observe that $\mu(q_{i+1}, \sigma, v) = \mu(q_{i+1}, \sigma_k \sigma_f, v) = \mathbf{F}_{v,1}(\sigma_k \sigma_f)$, and therefore define $\mathbf{B}_{S,v,1} = L_f^* \big/ \epsilon \oplus \mathbf{F}_{v,1}$.

Second, if $1 < k < |S(v)| + 1$, i.e. that k refers to an internal patch in $S(v)$. Once the registers are combined in some order, any changes can only be appends at the beginning and end of the register value. The k^{th} constant in $\mu(q_{i+1}, \sigma, v)$ is consequently determined by σ_1. Therefore, define $\mathbf{B}_{S,v,k} = \mathbf{A}_{S,v,k} \oplus L_f^* \big/ \epsilon$.

Finally, if $k = 1$, or $k = |S(v)| + 1$, i.e. k is either the first or the last patch. First, we know that $v \in supp(S)$. Also, we know that any registers which flow into v have to be non-support registers. See figure V.3. Thus, the value being appended to v while processing σ_j is determined entirely by σ_j and σ_{j-1}. The idea is to use chained sum to compute this value.

We will now define $\mathbf{B}_{S,v,k}$ for $k = |S(v)| + 1$. The case for $k = 1$ is symmetric, and would involve reversing the order of the operators, and replacing chained sum with the left-chained sum.

We are determining the constant value appended to the end of v while processing σ. We distinguish three phases of addition: while processing σ_1, only the constant

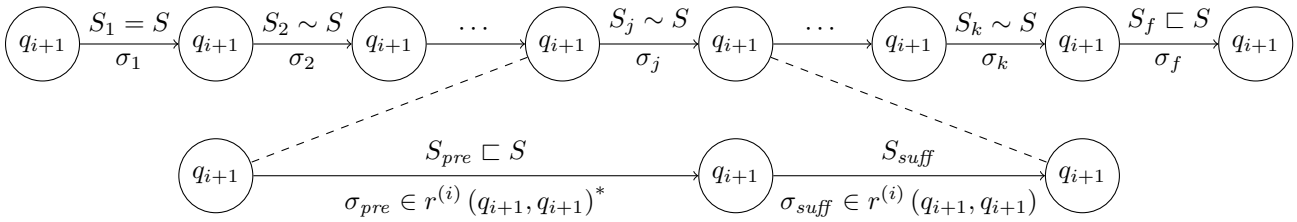


Figure V.2: Decomposing paths in $r^{(i)}(q_{i+1}, q_{i+1})^*$ with shape S. σ_j can be unambiguously written as $\sigma_{pre}\sigma_{suff}$, with $\sigma_{suff} \in r^{(i)}(q_{i+1}, q_{i+1})$.

at the end of $\mathbf{A}_{S,v,k}$ is appended. While processing σ_j, $j > 1$, *both string constants and registers* appearing after the occurrence of v in $S(v)$ are appended. Third, while processing σ_f, both string constants and registers appearing after the occurrence of v in $S(v)$ are appended. The interesting part about the second case is that this appending happens in a loop, and we therefore need the lookback provided by the chained sum operator. Otherwise, this case is similar to the simpler third case, where a value is appended exactly once.

While processing σ_1, some symbols are appended to the k^{th} position in $S(v)$. This is given by $f_{pre} = \mathbf{A}_{S,v,k} \oplus L_f^* \Big/ \epsilon$.

Similarly, while processing the suffix σ_f, some symbols are appended. Say some register $u \to v$ in S_f. Then $u \notin supp(S_f)$, and hence $u \notin supp(S)$ and $u \notin supp(S_k)$. Thus, the value appended by σ_f is determined by $\sigma_k\sigma_f$. 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 $f_{post} = L_f^* \Big/ \epsilon \oplus \rhd \{f_{pre,S_k,S_f} \mid S_k \sim S \text{ and } S_f \sqsubset S\}$.

Finally, consider the value appended while processing σ_j, for $j > 1$. This is similar to the case for σ_f: if $u \to v$ in S_j, when $u \notin supp(S_j)$ and $u \notin supp(S_{j-1})$. Thus, the value appended 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 $\mathbf{A}'_{S_{j-1}} = \mathbf{A}_{S_{j-1}} \ll \text{Dom}(\mathbf{A}_{S_j})$, and $\mathbf{A}'_{S_j} = \mathbf{A}_{S_j} \gg \text{Dom}(\mathbf{A}_{S_{j-1}})$. 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 $f = \sum(\rhd \{f_{S_{j-1},S_j} \mid S_{j-1} \sim S \text{ and } S_j \sim S\}, L_f)$.

Finally, define $\mathbf{B}_{S,v,k} = (f_{pre} + f_{post}) \rhd (f_{pre} + f + f_{post})$.

By construction, we have:

Claim 40. $\mathbf{B}_S$ summarizes all paths in $r^{(i)}(q_{i+1}, q_{i+1})^*$ with shape S.

This completes the proof of theorem 26.

VI. 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 [14], [4] and/or when the range is a semiring [12], [5]. Finally, on the practical side, we plan to develop a declarative language for document processing based on the regular combinators identified in this paper.

References

[1] Alfred Aho, John Hopcroft, and Jeffrey Ullman. A general theory of translation. *Mathematical Systems Theory*, 3(3):193–221, 1969.

[2] 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.

[3] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 599–610. ACM, 2011.

[4] Rajeev Alur and Loris D'Antoni. Streaming tree transducers. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming*, volume 7392 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2012.

[5] Rajeev Alur, Loris D'Antoni, Jyotirmoy Deshmukh, Mukund Raghothaman, and Yifei Yuan. Regular functions and cost register automata. In *Proceedings of the 28th Annual Symposium on Logic in Computer Science*, pages 13–22. IEEE Computer Society, 2013.

[6] Rajeev Alur, Emmanuel Filiot, and Ashutosh Trivedi. Regular transformations of infinite strings. In *Proceedings of the 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 65–74. IEEE Computer Society, 2012.

[7] Mikolaj Bojanczyk. Transducers with origin information. *CoRR*, abs/1309.6124, 2013.

[8] Krishnendu Chatterjee, Laurent Doyen, and Thomas Henzinger. Quantitative languages. *ACM Transactions on Computational Logic*, 11(4):23:1–23:38, July 2010.

[9] 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, *Automata, Languages and Programming*, volume 52 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 1977.

[10] Thomas Colcombet. The theory of stabilisation monoids and regular cost functions. In Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas, editors, *Automata, Languages and Programming*, volume 5556 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2009.

[11] 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 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1992.

[12] Manfred Droste, Werner Kuich, and Heiko Vogler, editors. *Handbook of Weighted Automata*. Monographs in Theoretical Computer Science. Springer, 2009.

[13] 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.

[14] Joost Engelfriet and Sebastian Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Information and Computation*, 154(1):34–91, 1999.

[15] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330. ACM, 2011.

[16] Eitan 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] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjørner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 137–150. ACM, 2012.