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 (, 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 \( \text{cyl}_{d} \) maps strings \( \sigma \) 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 \( f \gg g \) maps an input string \( \sigma \) to \( f(\sigma) \), if this value is defined,
the output values are strings themselves and \( \sigma \) introduces 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 combiners of conditional choice, split sum, and iterated sum. The proof is fairly straightforward, and builds on the known properties of cost register automata, while computing its cost, and thus, the result of combining two partial functions remains a partial function. It turns 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 \( \sigma \) 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 combiners 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 \( \Sigma \) 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 \( \Gamma \) 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 \( d \otimes \perp = \perp \otimes d = d \), 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 \( \sigma \).

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 \( f \triangleright g \) as

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

The unambiguous choice \( f \ll g \) is defined as

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

Example 2. The indicator function \( 1_L : \Sigma^* \to \mathbb{Z} \) is defined as \( 1_L(\sigma) = 1 \) if \( \sigma \in L \) and \( 1_L(\sigma) = 0 \) otherwise. This function can be expressed using the conditional choice operator as \( L/1 \triangleright \Sigma^*/0 \), and using the unambiguous choice operator as \( L/1 \ll (\Sigma^* \setminus L)/0 \). Note that the representation using the unambiguous union operator relies on regular expressions being closed under set difference.
Observe that \(\triangleright\) and \(\sqcup\) are the analogs of union in regular expressions, with the important difference being that \(\triangleright\) is non-commutative, and \(\sqcup\) is non-associative\(^1\).

### 2.3 Sum operators

The sum \(f \oplus g\) of two functions \(f, g : \Sigma^* \to \mathbb{D}_+\) is defined as \(f \circ g)(\sigma) = f(\sigma) \circ g(\sigma)\). If there exist unique strings \(\sigma_1\) and \(\sigma_2\) such that \(\sigma = \sigma_1 \circ \sigma_2\) and \(f(\sigma_1)\) and \(g(\sigma_2)\) are both defined, then the split sum \(f \oplus g)(\sigma) = f(\sigma_1) \circ g(\sigma_2)\). Otherwise, \(f \circ g)(\sigma) = \bot\).

Over non-commutative monoids, this may be different from the left-split sum \(f \underset{\circ}{\oplus} g\): if there exist unique strings \(\sigma_1\) and \(\sigma_2\), such that \(\sigma = \sigma_1 \circ \sigma_2\), and \(f(\sigma_1)\) and \(g(\sigma_2)\) are both defined, then \(f \underset{\circ}{\oplus} g)(\sigma) = f(\sigma_1) \circ g(\sigma_2)\). Otherwise, \(f \underset{\circ}{\oplus} g)(\sigma) = \bot\).

The split sum operators \(\bullet \circ\) and \(\circ \bullet\) 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 \(\sigma\) could be split in two different ways, \(\sigma = \sigma_1 \circ \sigma_2 = \sigma_3 \circ \sigma_4\), 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 \(f(\sigma_1) \circ g(\sigma_2) = f(\sigma_3) \circ g(\sigma_4)\), and this would lead to difficulties in defining \(f \oplus g\) as a function. The easiest way to avoid this difficulty is to define \(f \oplus g(\sigma)\) only when the split \(\sigma = \sigma_1 \circ \sigma_2\) into inputs for \(f\) and \(g\) is unique.

### 2.4 Iteration

The iterated sum \(f^{\oplus 1}\) of a cost function is defined as follows. If there exist unique strings \(\sigma_1, \sigma_2, \ldots, \sigma_k\) such that \(\sigma = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_k\) and \(f(\sigma_i)\) is defined for each \(\sigma_i\), then

\[
(f^{\oplus 1})(\sigma) = f(\sigma_1) \circ f(\sigma_2) \circ \cdots \circ f(\sigma_k).
\]

Otherwise, \(f^{\oplus 1})(\sigma) = \bot\). The left-iterated sum \(f^{\circ 1}\) is defined similarly: if there exist unique strings \(\sigma_1, \sigma_2, \ldots, \sigma_k\) such that \(\sigma = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_k\) and \(f(\sigma_i)\) is defined for each \(\sigma_i\), then

\[
(f^{\circ 1})(\sigma) = f(\sigma_k) \circ f(\sigma_{k-1}) \circ \cdots \circ f(\sigma_1).
\]

Otherwise, \(f^{\circ 1})(\sigma) = \bot\). The reverse combinator \(f^{\circ r}\) is defined as \(f^{\circ r}(\sigma) = f(\sigma^{\circ r})\), where \(\sigma^{\circ r}\) is the string reverse of \(\sigma\). 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 \(|\cdot| : \Sigma^* \to \mathbb{Z}\) counts the number of \(a\)-s in the input string. This is represented by the function expression \((a/1 \triangleright b/0)^{\circ \circ}\). The identity function \(id : \Sigma^* \to \Sigma^*\) is given by the function expression \(a \triangleright b\). The function \(copy\) which maps an input string to \(as\) is then given by the expression \(id \circ \circ\): note that the image of a regular language of input strings \(\Sigma^*\) may not be a regular language. On the other hand, the expression \((a \triangleright b)^{\circ \circ}\) is the function which reverses its input: \((a \triangleright b)^{\circ \circ}(\sigma) = \sigma^{\circ r}\) for all \(\sigma\). This is also equivalent to the expression \(id^{\circ r}\).

**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

\[
m = (C/2 \circ \circ \triangleright (C/1 \circ \circ S/0 \circ \circ (C/1 \triangleright S/0)^{\circ \circ}))
\]

\(^1\)Observe that \(f \circ (g \sqcup h)(\sigma) = f(\sigma)\) and \((f \circ g) \sqcup h)(\sigma) = h(\sigma)\) when all of \(f(\sigma), g(\sigma),\) and \(h(\sigma)\) are defined.
2.6 Function composition

Let \( f : \Sigma^* \to \Gamma^* \) 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 8. Composition is an alternative to chained sum for expressivity completeness --- in this example, we use composition to express shuffle. Let \( \text{copy}_L = (a^* \otimes \varepsilon \otimes b) \otimes (a^* \otimes \varepsilon \otimes b) \) be the function which accepts strings from \( L \) and repeats them twice. The first step of the transformation is therefore the expression \( \text{copy}_L \otimes \). We then drop the first copy of \( P_1 \), and use \( \text{drop} = \varepsilon \otimes \mu \otimes L/\varepsilon \). The function \( \text{ensurelen} = \mu \otimes \Sigma^+ / \varepsilon \) 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 \( a\cdot s \) from the second patch, and then transforms the \( a\cdot s \) from the first patch into \( b\cdot s \). \( f = (a/b^* \otimes \varepsilon \otimes b/\varepsilon) \otimes (a/b^* \otimes \varepsilon \otimes b/\varepsilon) \). Thus, \( \text{shuffle} = \text{drop} \circ \text{copy}_L \otimes \text{ensurelen} \).

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 coplysers 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 \( u := u \otimes v \otimes d \) 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 \( \sigma \) goes to \( \gamma \) on input \( a \), provided \( v \geq 5 \) --- and that the update expressions satisfy the coplysers (or single-use) requirement. CCRA 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 CCRA, 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 \( f : V \to (V \cup \mathbb{D})^* \) coplysers 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 coplysers 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, \( \Sigma \) 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 coplysers 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 coplysers.

The semantics of a CCRA \( M \) is specified using configurations. A configuration is a tuple \( \gamma = (q, \mu, v) \) where \( q \in Q \) is the current state and \( \mu : V \to \mathbb{D} \) is the register valuation. The initial configuration is \( \gamma_0 = (q_1, \mu, v_0) \), where \( \mu (q, v) = 0 \), for all \( v \). For simplicity of notation, we first extend \( \mu \) to \( V \cup \mathbb{D} \to \mathbb{D} \) by defining \( \mu (d) = d \), for all \( d \in \mathbb{D} \), and then further extend it to strings \( \sigma : (V \cup \mathbb{D})^* \to \mathbb{D} \) by defining \( \mu (v_1 v_2 \ldots v_k) = \mu (v_1) \mu (v_2) \cdots \mu (v_k) \). If the machine is in the configuration \( \gamma = (q, \mu, v) \), then on reading the symbol \( a \), it transitions to the configuration \( \gamma' = (q', \mu', v) \), and we write \( \gamma \to^{a} \gamma' \), where \( q' = \delta (q, a) \), and for all \( v, \mu (v) = \mu (\mu (q, a, v)) \).

We now define the function \( [M] : \Sigma^* \to \mathbb{D} \) computed by \( M \). On input \( \sigma \in \Sigma^* \), say \( \gamma_0 \to^{\sigma} (q_f, \nu) \). If \( q_f \in F \), then \( [M] (\sigma) = \nu (q_f) \). Otherwise, \( [M] (\sigma) = \bot \).

A cost function 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 \( \Gamma^* \) over the output alphabet under concatenation.

Example 11. We present an example of an SST in figure 3.1. The machine \( M_{\text{shuffle}} \) computes the function \( \text{shuffle} \) from example 7. It maintains 3 registers \( x, y, z \), all initially holding the value \( e \). 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, CCRA are equivalent in expressiveness to the simpler model of additive cost register automata (ACRA). In theorem 14, 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 “\( \text{a} := v \otimes d \)”, for some registers \( \text{a} \) 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 \( M_{\text{coffee}} \) which computes the function \( \text{coffee} \) described in example 4. In the state \( q_{-s} \), 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 \( \triangleright, \odot, \odot, \odot \otimes, \odot \) \odot, 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 CCAs for the base functions \( L/d \), we start with the DFA \( A \) accepting \( L \), and convert it into a CCA \( M \) where the accepting states output the constant value \( d \). For \( f \triangleright g \) and \( f \odot g \), 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 \( f \triangleright \odot g \), 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 \( D \) 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 \( A = (Q, \Sigma, \delta, q_0, F) \) 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 \( \sigma = \sigma_1 \otimes \sigma_2 \) 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 \( \sigma = \sigma_1 \otimes \sigma_2 \otimes \ldots \otimes \sigma_k \); and if \( R_1 \) and \( R_2 \) are unambiguous regular expressions, then \( R_1 \otimes R_2 \) is an unambiguous r.e. such that \( L(R_1 \otimes R_2) = \{ \sigma \in \Sigma^* \mid \sigma_1 \in L(R_1) \text{ or } \sigma_2 \in L(R_2), \text{ but not both} \} \). For example, \( aab \in L(\Sigma^* \cdot b \otimes \Sigma^* \cdot a \cdot b) \). 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 \( \sigma \) 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. \( r^{(0)}(q, q') = \{a \in \Sigma \mid q \rightarrow^a q'\} \).
2. \( r^{(i+1)}(q, q') = r^{(i)}(q, q_{i+1}) \cdot r^{(i)}(q_{i+1}, q_{i+1})^* \cdot r^{(i)}(q_{i+1}, q') \).

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

4.2 Converting function expressions to ACRA

We sketch the proof using the ACRA \( M_{\text{coffee}} \) from figure 3.2. The idea is to view \( M_{\text{coffee}} \) as a non-deterministic automaton \( A_{\text{coffee}} \) (figure 4.1) over the set of vertices \( Q \times V \): the state \( (q, v) \) of the
with the conditional choice operator
Also observe that this NFA
Theorem 15.
for the case of string transductions, i.e. where
D
2
⊗
D
to hold. To simplify the presentation, we prove theorem 15 only
sum (•|⊗)
functions combined with the unambiguous choice (•|⊗), chained sum (•|⊗)
and their left-additive versions (⊗•, (•|⊗)⊗). The unambiguous choice operator can be equivalently replaced with the conditional choice operator ⊢, and the theorem continues to hold. To simplify the presentation, we prove theorem 15 only for the case of string transductions, i.e. where \( D = \Gamma^* \), for some
finite output alphabet \( \Gamma \). Note that this is sufficient to establish the theorem in its full generality: let \( \Gamma_0 \subseteq \Gamma \) 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 \( \Sigma^* \) to output strings in \( \Gamma^* \). The restricted version of theorem 15 can then be used to construct this SST to function expression form, which when interpreted over the original domain \( D \) 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 1, for each pair of states \( q \) and \( q' \), a collection of function expressions which together summarize all strings \( \sigma \in \tau(i)(q,q') \).

In the CCRA \( M_{\text{coffee}} \) from figure 3.1, consider the string \( aab \) processed from the state \( q_1 \). After processing \( aab \), \( M_{\text{coffee}} \) is left in \( q_2 \). The final values of the registers \( x \), \( y \), and \( z \), in terms of their initial values, are \( xaayz, zb \), and \( e \) respectively. Thus, the string \( aab \) processed from \( q_1 \) may be summarized by the pair (\( q_2 = \delta(q_1, aab) \), \( \{ x \mapsto xaayz, y \mapsto zb, z \mapsto e \} \)), indicating the state of the machine after processing the string, and the final values of the registers. Similarly, the summaries for the strings \( bb \) and \( baab \) processed from the initial state \( q_1 \) are \( (q_2, \{ x \mapsto xyaazbb, y \mapsto z, z \mapsto e \}) \) 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, \( \{ x \mapsto y = \gamma_1x\gamma_2y_3y, y \mapsto z, z \mapsto \gamma_6 \} \), \( \{ x \mapsto y = \gamma_1x\gamma_2y_3y_4, y \mapsto z, z \mapsto \gamma_6 \} \), and \( \{ x \mapsto y = \gamma_1x\gamma_2y_3y_4, y \mapsto z, z \mapsto \gamma_6 \} \), 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 \( \sigma \in \tau(i)(q,q') \), with a given shape \( S \) is a regular language. Next, for each shape \( S \), we compute an “expression vector” \( R_S(i)(q,q') \) for each patch \( j \) appearing in the register update expression, there is a corresponding function expression \( R_S(i)(q,q') : \Sigma^* \rightarrow \Gamma^* \) such that for all input strings \( \sigma \in \tau(i)(q,q') \) with shape \( S \), \( \gamma = R_S(i)(q,q') (\sigma) \). Finally, for each accepting state \( q_1 \in F \), using the summary \( R_S(i)(q_1,q_1) \), we compute the output of the CCRA on strings \( \sigma \) that reach the state \( q_1 \) 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 \( R_S(i)(q,q') \) in the short version of this paper. Subsections 5.4–5.7 describe the inductive construction of \( R_S(i+1)(q,q') \). Note that the chained sum is crucial to this construction, and will appear while finally constructing \( R_S(i+1)(q,q') \) in subsection 5.7.

5.2 A theory of shapes
Definition 16 (Shape of a path). A shape \( S : \Gamma \rightarrow \Gamma^* \) is a copyless function over a finite set of registers \( \Gamma \). Let \( \sigma = q_1, a_{i_1} \rightarrow q_2, a_{i_2} \rightarrow \cdots \rightarrow q_{i_n} \) be a path through a CCRA \( M \). The shape of the path \( \sigma \) is the function \( S_\sigma : \Gamma \rightarrow \Gamma^* \) such that for all registers \( v \in \Gamma \), \( S_\sigma (v) \) is the string projection onto \( v \) of the register update expression \( \mu(q_1, \sigma, v) \). Let \( \sigma = \mu (q_1, \sigma, v) \).

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 \( V \rightarrow V^* \) 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 $S_2(v) = u_1, u_2, \ldots, u_k$. Then $S_1 \cdot S_2(v) = s_1 = s_2 = \ldots = s_k$, 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 $\sigma_1$ and $\sigma_2$ be two paths through a CCRA $M$ such that the final state of $\sigma_1$ is the same as the initial state of $\sigma_2$. Then, for all registers $v$, $S_{\sigma_1 \cdot \sigma_2}(v) = S_{\sigma_1}(v) \cdot S_{\sigma_2}(v)$.

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 $A$ 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 $A_{v,k} : \Sigma^* \to \Gamma^*$. An expression vector $A$ summarizes a set of paths $L$ with shape $S$, if for each path $\pi \in L$ with initial state $q$, and input string $\sigma$, 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 $A_{v,k}(\sigma)$.

**Example 20.** Consider the loop $a^*$ at the state $q_2$ of $M_{shuffle}$. Consider the concrete string $x = ax^6, y = y^6$, and $z = z^b$. The shape of this set of paths is the identity function $S(v) = v$, for all $v$. Define the expression vector $A$ as follows: $A_{a,x} = A_{y,y} = A_{z,b} = a^*/\epsilon$, $A_{a,y} = (a/a)^{+\infty}$, and $A_{z,b} = (a/b)^{+\infty}$. Then $A$ 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 $R_S^i(q, q')$ 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 $A$ for a shape $S$, the domain of the expression vector, written as $\text{Dom}(A)$, is defined as the language $\bigcap_{v,k} \text{Dom}(A_{v,k})$, where $\text{Dom}(A_{v,k})$ 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 $f \cap L = f \cap L/\epsilon$. 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 $A$ to languages $L$, $A \cap L$, by defining $(A \cap L)_{v,k}$ as $A_{v,k} \cap L$.

#### 5.3.2 Choice

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

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

The notation $\{f_1, f_2, \ldots, f_k\}$ stands for $f_1 \sqcup f_2 \sqcup \cdots \sqcup f_k$. As the base case, $\{\}$ is $\perp$. 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 $\sigma = \sigma_1 \sigma_2 \in L$, and applies $f$ to the prefix $\sigma_1$, and ignores the suffix $\sigma_2$, provided the split is unique, i.e. $f \ll L = f \bullet \cap L/\epsilon$. Similarly, the right-shifted function $f \gg L = L/\epsilon \bullet \cap f$. The shift operators can also be extended to expression vectors: $A \ll L$ is defined as $(A \ll L)_{v,k} = A_{v,k} \cap \text{Dom}(A) \ll L$, and $A \gg L$ is defined as $(A \gg L)_{v,k} = A_{v,k} \cap \text{Dom}(A) \gg L$.

#### 5.3.4 Concatenation

Pick three states $q, q', q''$ 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 $A$ and $B$ summarize all paths in $L$ and $L'$ respectively, and consider a string $\sigma = \sigma_1 \sigma_2$ with $\sigma_1 \subseteq L$ and $\sigma_2 \subseteq L'$. The expression vector $A$ describes the register values of the machine after processing $\sigma_1$, in terms of their original values before the computation was started, and $B$ describes the register values of the machine after processing $\sigma_2$, in terms of their values after processing $\sigma_1$. These expression vectors can therefore be combined into an expression vector $A \cdot B$, 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 $A$ and $B$ to the relevant parts of the input string $\sigma$, 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 $R_{S}^{i+1}(q, q')$ in terms of the candidate function expressions at step $i$. To satisfy the translation invariant, we have to summarize all paths in $r^{(i+1)}(q, q') = r^{(i)}(q, q_{i+1}) \cdot r^{(i)}(q, q_{i+2}) \cdots r^{(i)}(q, q_i) \sqcup r^{(i)}(q, q')$. Therefore, the central problem is to construct, for each shape $S$, an expression vector $B_S$ 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 $R_{S}^{i+1}(q, q')$ 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 $u \in V$, there exists $v \in V$ 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 $\preceq$.
Figure 5.1: Visualizing shapes as bipartite graphs. Figures 5.1a–5.1c describe the shapes of some paths in $M_{\text{shuffle}}$ 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 $f : V \rightarrow V'$ 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 $\delta'$, 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 $S_\perp$ from figure 5.1. Let $\sigma$ be a path through the CCRA with shape $S_1$. Then, no sub-path of $\sigma$ can have shape $S_\perp$, 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 $S$ 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 $S$, $\text{supp}(S) = \{ v \in V \mid v \text{ occurs in } S(v) \}$. If $S_1$ and $S_2$ are two shapes, then $S_1 \sqsubseteq S_2$ iff $\text{supp}(S_1) \supseteq \text{supp}(S_2)$. We call two shapes $S_1$ and $S_2$ support-equal, written as $S_1 \sim S_2$, if $\text{supp}(S_1) = \text{supp}(S_2)$.

For example, the shape $S_\perp$ from figure 5.1 is the bottom element of $\sqsubseteq$, and $S_\perp \sqsubseteq S_\top$. $S_1 \sim S_2$, 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 $\text{supp}(S_1)$ and $\text{supp}(S_2)$. The following two claims, 25 and 26, formalize the intuition that $\sqsubseteq$ and $\sim$ describe the possible shapes of subpaths.

**Claim 25.** Let $\sigma$ be a path through the CCRA $M$ with shape $S$, and $\sigma'$ be a subpath of $\sigma$ with shape $S'$. If $S' \not\sqsubseteq S_\perp$, then $S' \sim S$.

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

**5.6 Decomposing loops**

Recall that our main remaining problem, starting from subsection 5.4, has been to construct $R_{S_{\top}}(q, q')$, which summarizes strings in $r_{\top}(q, q')$ with shape $S$. In subsection 5.3, we defined the choice operator over expression vectors; thus, if $C_S$ summarizes strings in $r_{\top}(q, q_{+1}) \cdot r_{\top}(q_{+1}, q_{+1})^\ast \cdot r_{\top}(q_{+1}, q')$ with shape $S$, then we can define $R_{S_{\top}}(q, q') = C_S \shuffle R_{S_{\top}}(q, q')$. Furthermore, we also defined the concatenation operator over expression vectors: if for each intermediate shape $S_2$, if $B_{S_2}$ summarizes strings in $r_{\top}(q_{+1}, q_{+1})^\ast$ with shape $S_2$, then we can construct the desired $C_S = \bigcup\{ R_{S_{\top}}(q, q_{+1}) \cdot B_{S_2} \cdot R_{S_{\top}}(q_{+1}, q') \mid S_1 \cdot S_2 \cdot S_3 = S \}$. Our goal is therefore to construct $B_S$ 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 $\sigma$. In this subsection and the next, we construct $B_S$ assuming that $B_S$ for all shapes $S \subseteq S$ is known. Furthermore, since each string $r_{\top}(q_{+1}, q_{+1})$ is non-empty, $\epsilon \notin r_{\top}(q_{+1}, q_{+1})^\ast$. We therefore separately handle the case of $\epsilon \in r_{\top}(q_{+1}, q_{+1})^\ast$, and in the rest of this section, we construct an expression vector $B_S$ which summarizes strings (all of which are non-empty) in $r_{\top}(q_{+1}, q_{+1})^\ast$ with shape $S$.

Consider any path $\sigma \in r_{\top}(q_{+1}, q_{+1})^\ast$ 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 $j$, $1 \leq j \leq k$, $\sigma_j \in r_{\top}(q_{+1}, q_{+1})^\ast$ is a self-loop at $q_{+1}$, with shape $S_j$ such that $S_j \sim S$,
2. for each $j$, $1 \leq j \leq k$, and for each proper prefix (possibly empty) $\sigma_{pre} \in r_{\top}(q_{+1}, q_{+1})^\ast$ of $\sigma_j$, $S_{pre} \subsetneq S$, and
3. $\sigma_f \in r_{\top}(q_{+1}, q_{+1})^\ast$ (possibly empty), and its shape $S_f$ satisfies $S_f \subseteq S$.

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

We conclude this subsection by constructing expression vectors $A_{S_f}$, which summarize these minimal subpaths $\sigma_f$. First note that $\sigma_f \in L_{\text{first}}(S_f)$, where for each shape $S' \sim S$, $L_{\text{first}}(S_f)$ is the set of all paths $\sigma \in r_{\top}(q_{+1}, q_{+1})^\ast$ with shape $S_f$ such that no proper prefix $\sigma_{pre}$ of $\sigma$ has shape $S_{pre} \sim S$. Next, unambiguously decompose $\sigma = \sigma_{pre} \sigma_{last}$, with $\sigma_{pre} \in r_{\top}(q_{+1}, q_{+1})^\ast$, $\sigma_{last} \in r_{\top}(q_{+1}, q_{+1})^\ast$; therefore $\sigma_{last}$ is the last iteration of the loop in $r_{\top}(q_{+1}, q_{+1})^\ast$ and $S_{pre} \subsetneq S$. By the induction hypothesis, $\sigma_{last}$ is summarized by the expression vector...
5.7 Constructing $B_S$

Starting from the previous subsection, our main goal has been to construct the expression vector $B_S$, 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 $\sigma$, and so we can determine the shape of each proper prefix $\sigma'$. We also resolved the case of $\epsilon$ in $r^{(i)}(q_{i+1}, q_{i+1})^+$ separately, so that we are interested in constructing $B_{S'}$, which summarizes paths $\sigma \in r^{(i)}(q_{i+1}, q_{i+1})^+$. In the previous subsection, we constructed expression vectors $A_S$ 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 $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_k$, where each $\sigma_j \in r^{(i)}(q_{i+1}, q_{i+1})^+$, for each $j$, $1 \leq j \leq k$, $\sigma_j \in L_{\text{first}}(S_j)$, and $S_f \supseteq S$. Then claim 27 establishes that $A_{S_f}$ summarizes each $\sigma_j$. In this section, we complete the construction of the expression vector $B^+_{S_f}$, and hence also the construction of $B_S$, 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 assigned to a register while processing $\sigma_j$ may be computed while processing $\sigma_{j-1}$. We distinguish three cases to construct $B^+_{S,S,v,k}$.

5.7.1 $S(v) = \epsilon$

Informally, this is the case when $v$ is reset in $S$, and therefore reset during each minimal subpath $\sigma_j$ (but not necessarily during $\sigma_j$). 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 $F = \bigcup \{ A_{S_1} \cdot B_{S_2} \mid S_1 \cdot S_2 = S, S_1 \sim S \text{ and } S_2 \supseteq S \}$, and define $B^+_{S,v,k} = r^{(i)}(q_{i+1}, q_{i+1})^+ / \epsilon \otimes F_{v,k}$. $S(v) \neq \epsilon$, and $1 < k < |S(v)| + 1$. This corresponds to the case when $k$ is neither the first nor the last patch. First, we know that $v \in supp(S)$. Also, any registers which flow into $v$ while processing $\sigma$ have to be non-support registers $w$, and therefore the value appended to $v$ while processing $\sigma_{j+1}$ is determined by $\sigma_j$.

5.7.2 $S(v) \neq \epsilon$, and $1 = k < |S(v)| + 1$. 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 $\sigma$ have to be non-support registers. See figure 5.3. Thus, the value being appended to $v$ while processing $\sigma_j$ is determined by $\sigma_{j-1}$. We will define $B^+_{S,v,k}$ 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 $\gamma_{v,k}$ is determined entirely by $\sigma_k$, the value appended by $\sigma_j$ is determined by $\sigma_{j+1}$, and the value appended while processing $\sigma_j$, $1 < j < k$ is determined by $\sigma_{j+1} \sigma_j$.

1. While processing $\sigma_1$, some symbols are appended to the $k$th position in $S(v)$. Define $f_{\text{pre}} = A_{S_{v,k}} \otimes r^{(i)}(q_{i+1}, q_{i+1})^+ / \epsilon$, so that $f_{\text{pre}}(\sigma)$ is the value appended at the end of $\gamma_{v,k}$ by $\sigma_1$.

2. For some register $u \neq v$, let $u$ occur in $S(v)$, so that the value in $u$ computed during $S_f$ flows into $v$ while processing $\sigma_f$. For each pair of shapes $S_1$ and $S_f$ such that $S_1 \supseteq S$, and $S_f \supseteq S$, consider $A_{S_{1,k}} = A_{S_{1,k}} \otimes \text{Dom}(S_f)$, and $B_{S_{1,k}} = B_{S_{1,k}} \otimes \text{Dom}(S_{1,k})$. Consider the update expression $B_{S_{1,k}}$: say this is $v := \sigma \tau$, where $\sigma$ and $\tau$ are strings over expressions and registers. For each register $u$ in $\tau$, substitute the value $A_{S_{1,k,u}}$ — since $u$ was reset while processing $S_f$, this expression gives the contents of the register $u$ — and interpret string concatenation in $\tau$ as the function combinator sum. Label this result as $f_{\text{post}}(S,f)$. Define $f_{\text{post}} = (r^{(i)}(q_{i+1}, q_{i+1})^\epsilon) \otimes \bigcup \{ f_{\text{post}}(S,f) \mid S_f \sim S \text{ and } S_f \supseteq S \}$, and observe that $f_{\text{post}}(\sigma)$ is the value appended to the end of $\gamma_{v,k}$ by $\sigma_f$. 

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 $\sigma_j$ of $\sigma$ has shape $S_{j-1} \supseteq S$, or equivalently $\sigma_j \in L_{\text{first}}(S_j)$. In the case of "minimal paths" $\sigma \in r^{(i)}(q_{i+1}, q_{i+1})^+$, it follows that the expression vector $A_S$ summarizes all paths in $L_{\text{first}}(S')$. 

Figure 5.3: Every non-support register $u \notin supp(S)$ is rest before processing $\sigma_j$, for $1 \leq j \leq k$. Therefore its value is determined entirely by $\sigma_j$. 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 $\sigma_{j+1}$ is determined by $\sigma_j$. 

$R_{\text{suff}}^i (q_{i+1}, q_{i+1})$, and by the second induction hypothesis, $B_{\text{pre}}$ is known. Define $A_{S'} = \bigcup \{ B_{\text{pre}} \cdot R_{\text{suff}}^i (q_{i+1}, q_{i+1}) \mid S_{\text{pre}} \cdot S_{\text{suff}} = S' \text{ and } S_{\text{pre}} \subseteq S\} \cup R_{\text{suff}}^i (q_{i+1}, q_{i+1})$. By construction: Claim 27. For all shapes $S' \sim S$, the expression vector $A_{S'}$ summarizes all paths in $L_{\text{first}}(S')$. 

$S(v) = \epsilon$
3. Finally, consider the value appended while processing \( \sigma_j \), for \( j \geq 1 \). This is similar to the case for \( \sigma_i \): the value appended to \( S_{i-1} \sim S \) and \( S_j \sim S \), consider \( A_{S_j,1} = A_{S_j-1} \ll \text{Dom}(A_{S_j}) \), and \( A_{S_j} = A_{S_j} \gg \text{Dom}(A_{S_j-1}) \). Consider the update expression \( A_{S_j,1,k} \). Let this be \( v := v_{\sigma;v} \), where \( \sigma \) and \( \tau \) are strings over expressions and registers. For each register \( u \) in \( \tau \), substitute the value \( A_{S_j,1,u,1} \sim v \), since \( u \) was reset while processing \( S_j \), this expression gives the contents of the register \( u \) — and interpret string concatenation in \( \tau \) as the function combinator sum. Label this result as \( f_{S,j-1,S_j} \). Define
\[
f = (\bigcup \{ f_{S_j-1,S_j} \mid S_j-1 \sim S \text{ and } S_j \sim S \})^* \text{dom}.
\]
Finally, define \( B^+_S = (f_{\text{pre}} \otimes f_{\text{post}}) \cup (f_{\text{pre}} \otimes f_{\text{post}}) \).

The following observation completes the proof of theorem 15.

**Claim 28.** \( B^+_S \) summarizes all strings \( \sigma \in \rho(i)(q_{i+1}, q_{i+1}) \) 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 \( q_i \) 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 \( \ll \) and support equality \( \sim \) together capture this notion. 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 \( S' \triangleleft S \) 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

---

2 In an early attempt to prove this theorem, we tried to formalize the relation \( R_{\text{sp}}(S_1, S_2) \): “\( S_1 \) can appear as the shape of a subpath \( \sigma_{\text{sp}} \) of a longer path \( \sigma \) with shape \( S_2 \)”. This approach failed because the relation \( R_{\text{sp}} \) 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.