Concatenation State Machines and Simple Functions

Wojciech Debski and Wojciech Fraczak

IDT Canada, Ottawa, Ontario, Canada

Abstract. We introduce a class of deterministic push-down transducers called *concatenation state machine* (CSM) and we study its semantic domain which is a class of parial mappings over finitely generated free monoids, called *simple functions*.

1 Introduction

IDT Canada, a subsidiary of Integrated Device Technology, recently developed a specialized programmable processing unit for the real-time network packet filtering and classification, called *PAX.port*. In this paper we define a mathematical model for *PAX.port* hardware which we call *concatenation state machine* (CSM).

A CSM can be seen as a pass-through device transforming a flow of input data into a flow of output data. More formally, a concatenation state machine M defines a partial mapping $|M| : \Sigma^* \mapsto \Omega^*$ where Σ and Ω are two finite alphabets. We demonstrate that there is a strong correlation between CSM and *simple languages* as defined in [1]. Thus, a mapping which can be defined by a CSM will be called "*simple function*".

The main result of the paper is the proof that a partial mapping $f : \Sigma^* \mapsto \Omega^*$ is a simple function if and only if there exists a finite set G of classifiers (classifiers are mappings defined on prefix codes and they constitute a monoid with respect to *classifier concatenation*) such that all quotients of f are included in G^*.

We end the paper with some results concerning the expressive power of simple functions.

2 Concatenation state machine

Let Σ and Ω be two disjoint finite sets, called *input* and *output* alphabets, respectively. A Concatenation State Machine (CSM) is a rooted directed labeled graph with three types of vertices: *switch nodes*, *concatenation nodes*, and *accepting nodes*; such that:

- All edges outgoing from a switch node are labeled by different symbols of Σ.
- There are always two ordered edges, *left* and *right*, outgoing from a concatenation node.
- Accepting nodes have no outgoing edges. An accepting node is labeled by an *accepting string* from Ω^*.

Consider the concatenation state machine in Figure 1. It is built over input alphabet $\Sigma = \{a, b, c\}$ and output alphabet $\Omega = \{\alpha, \beta, \gamma\}$. Switch nodes are represented by plain circles with outgoing edges labeled by letters of the input alphabet. Concatenation nodes are represented by rectangles with *left* edge outgoing from the left and *right* edge outgoing from the right part of the node. Accepting nodes are represented by double circles marked by their labels which are words over output alphabet Ω. The initial node of the CSM, is distinguished by a short incoming arrow. Intuitively, a switch node can be seen as a state in the

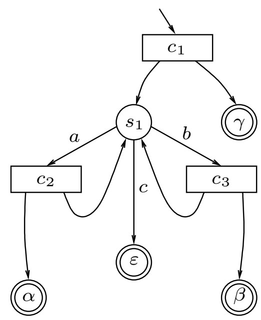

Fig. 1. Concatenation State Machine

deterministic automaton. A concatenation node represents a "subroutine call" for the destination of *left* edge; once the subroutine returns we continue with the destination of *right* edge. Accepting nodes are "return" instructions producing values defined by their labels.

Definition 1 *A CSM is a 9-uplet* $(\Sigma, \Omega, S, C, A, \eta_S, \eta_C, \lambda, i)$*, where:*

- Σ*,* Ω*,* S*,* C*,* A *are finite pairwise disjoint sets of* input symbols*,* output symbols*,* switch nodes*,* concatenation nodes*, and* accepting nodes*, respectively.*
- $\eta_S : (S, \Sigma) \mapsto (S \cup C \cup A)$*,* $\eta_C : (C, \{\texttt{left}, \texttt{right}\}) \mapsto (S \cup C \cup A)$*, and* $\lambda : A \mapsto \Omega^*$*, are partial mappings; and*
- $i \in (S \cup C \cup A)$ *is the* initial node*.*

A state of a concatenation state machine $M = (\Sigma, \Omega, S, C, A, \eta_S, \eta_C, \lambda, i)$, is defined by *execution stack* denoted `stack`. An execution stack is a list of nodes, i.e., $\texttt{stack} \in (S \cup C \cup A)^*$. The initial state of M consists of the execution stack containing one element, the initial node.

A CSM processes the nodes of the execution stack until the stack becomes empty. The machine runs the following execution loop. It pops the top node, s, of the stack. If s is an accepting node, $s \in A$, then the machine outputs $\lambda(s)$. If s is a switch node, $s \in S$, then the machine reads a new input symbol $a \in \Sigma$ and pushes the node $\eta_S(s, a)$ on top of the execution stack. If s is a concatenation node, $s \in C$, the machine pushes two nodes on the stack, first $\eta_S(s, \texttt{right})$ and $\eta_S(s, \texttt{left})$ next (so node $\eta_S(s, \texttt{left})$ is on the top of the execution stack). If the execution stack is not empty the execution loop is continued. The pseudo-code defining the execution loop of a CSM $(\Sigma, \Omega, S, C, A, \eta_S, \eta_C, \lambda, i)$ is presented in Figure 2.

```
while (stack is not empty) do
    s := pop(stack);  // pop the top of stack into an auxiliary variable s
    if (s ∈ A) then
        write(λ(s));
    if (s ∈ S) then
        read(a);  // read input symbol into an auxiliary variable a
        push(η_S(s, a), stack);
    if (s ∈ C) then
        push(η_S(s, right), stack);
        push(η_S(s, left), stack);
end while
```

Fig. 2. Pseudo-code for the execution loop of a CSM.

For a given input w, M completes if its execution stack becomes empty after reading the whole input w. Then we say that M accepts w, and the result of the acceptance, denoted by $|M|(w)$, is the concatenation of all outputs produced by *write*(). Otherwise, we say that M is not defined on w, denoted by $|M|(w) = \emptyset$. M will reject input w if one of the partial mappings η_S, η_C, or λ, is called with arguments for which the mapping is not defined, or if w is too short and no new symbol can be read, or if w is too long and the execution stack becomes empty before reading the last symbol of w.

The CSM in Figure 1 can be described by $(\Sigma, \Omega, S, C, A, \eta_S, \eta_C, \lambda, i)$, where: $\Sigma = \{a, b, c\}$, $\Omega = \{\alpha, \beta, \gamma\}$, $S = \{s_1\}$, $C = \{c_1, c_2, c_3\}$, $A = \{a_1, a_2, a_3, a_4\}$, $\eta_S = \{(s_1, a) \mapsto c_2, (s_1, b) \mapsto c_3, (s_1, c) \mapsto a_2\}$, $\eta_C = \{c_1 \mapsto (s_1, a_4), c_2 \mapsto (a_1, s_1), c_3 \mapsto (s_1, a_3)\}$ (by $\eta_C : c \mapsto (x, y)$, for $c \in C$ and $x, y \in S \cup C \cup A$, we mean that $x = \eta_C(c, \texttt{left})$, $y = \eta_C(c, \texttt{right})$, and that the both values are defined), $\lambda = \{a_1 \mapsto \alpha, a_2 \mapsto \varepsilon, a_3 \mapsto \beta, a_4 \mapsto \gamma\}$, and $i = c_1$. The execution steps of M on input word "*bac*" are: $(c_1) \xrightarrow{\eta_C(c_1)} (s_1, a_4) \xrightarrow{read(b)} (c_3, a_4) \xrightarrow{\eta_C(c_3)} (s_1, a_3, a_4) \xrightarrow{read(a)} (c_2, a_3, a_4) \xrightarrow{\eta_C(c_2)} (a_1, s_1, a_3, a_4) \xrightarrow{write(\alpha)} (s_1, a_3, a_4) \xrightarrow{read(a)} (a_2, a_3, a_4) \xrightarrow{write(\varepsilon)} (a_3, a_4) \xrightarrow{write(\beta)} (a_4) \xrightarrow{write(\gamma)} ()$. Thus, on input "*bac*", the CSM produces output string "$\alpha\beta\gamma$".

A concatenation state machine $M = (\Sigma, \Omega, S, C, A, \eta_S, \eta_C, \lambda, i)$ is called *trimmed* if:

- all nodes $S \cup C \cup A$ are accessible from i;
- from every switch node $s \in S$ there is a path to at least one accepting node $a \in A$;

– for every concatenation node $c \in C$, values $\eta_C(c, \texttt{left})$ and $\eta_C(c, \texttt{right})$ are both defined;
– there is no loop passing exclusively through concatenation nodes.

It is relatively easy to *trim* any concatenation state machine by removing all nodes which make the CSM non-trimmed. The trimmed CSM defines the same mapping from input strings into output strings as the original CSM.

3 Algebraic definition of CSM

Let $u, w \in \Sigma^*$. We say that u is a *prefix* (initial segment) of w, denoted by $u \leq w$, if there exists a word $v \in \Sigma^*$ such that $uv = w$. If $u \leq w$ and $u \neq w$ then u is a *proper prefix* of w. A language $L \subset \Sigma^*$ is called a *prefix code* if L does not contain two words such that one is a proper prefix of the other.

Definition 2 *We call* classifier *any partial mapping $f : \Sigma^* \mapsto \Omega^*$ such that $f^{-1}(\Omega^*)$ is a prefix code.*

Classifiers, i.e., functions defined on prefix codes, are of particular importance to real-time network packet classification, due to the fact that in case of *classifiers* a classification decision concerning a network packet can be made on-line by looking into an initial segment (header) of the packet, without the risk of having to change the decision later, when more bits arrive.

Proposition 1 *Every concatenation state machine defines a classifier.*

Proof. Since an execution of a concatenation state machine M is deterministic on the input word (read from left-to-right) and it stops once the execution stack becomes empty, M cannot accept an input word and its proper prefix at the same time. □

The classifier which is not defined for any input, i.e., such that its co-image of Ω^* is empty, will be called *empty classifier* and will be denoted by $\emptyset$. Classifiers which are defined (only) for the empty input word $\varepsilon \in \Sigma^*$, will be denoted by the result they produce, i.e., every output word $u \in \Omega^*$ defines classifier $u \stackrel{\texttt{def}}{=} \{\varepsilon \mapsto u\}$.

Definition 3 *Let $f, g : \Sigma^* \mapsto \Omega^*$ be two classifiers.* Classifier concatenation*, fg, is defined as:*

$$fg(w) \stackrel{\texttt{def}}{=} \{f(v_1)g(v_2) \in \Omega^* \mid w = v_1 v_2 \text{ and } v_1 \in f^{-1}(\Omega^*)\} \ .$$

Since $f^{-1}(\Omega^*)$ is a prefix code, $fg(w)$ is empty or a singleton set. Also $(fg)^{-1}(\Omega^*) = f^{-1}(\Omega^*)g^{-1}(\Omega^*)$, i.e., $(fg)^{-1}(\Omega^*)$ is a prefix code. Thus fg is a classifier.

Classifiers under *classifier concatenation* constitute a monoid with $\varepsilon : \{\varepsilon \mapsto \varepsilon\}$ as unit. Both, empty $\emptyset$ and unit ε classifiers, are called *trivial classifiers*. The monoid will be called *classifier monoid*.

Lemma 2 *Classifier monoid is a* cancellative monoid*, i.e., for every non-empty classifiers f, g, g', h, equality $fgh = fg'h$ implies $g = g'$.*

Definition 4 *Let $\Sigma = \{a_1, a_2, \dots, a_n\}$ be the input alphabet and $f_1, f_2, \dots, f_n : \Sigma^* \mapsto \Omega^*$ be classifiers.* Switch composition*, $a_1 f_1 + a_2 f_2 + \dots + a_n f_n$, is defined as:*

$$(a_1 f_1 + a_2 f_2 + \dots + a_n f_n)(w) \stackrel{\mathtt{def}}{=} f_i(v), \text{ where } w = a_i v, \text{ for some } i \in [1, n] \ .$$

Switch composition $a_1 f_1 + a_2 f_2 + \dots + a_n f_n$ yields a classifier.

Switch composition is an n-ary operation, where n is the size of the input alphabet. We allow abbreviated notation by omitting ε and terms with $\emptyset$. E.g., we will write $(a_1 + a_3 f_3)$ instead of $(a_1\varepsilon + a_2\emptyset + a_3 f_3 + a_4\emptyset + \dots + a_n\emptyset)$.

Definition 5 *Let $f : \Sigma^* \mapsto \Omega^*$ be a classifier and $v \in \Sigma^*$ an input word.* Quotient*, $v^{-1}f$, is defined as $(v^{-1}f)(w) \stackrel{\mathtt{def}}{=} f(vw)$. Quotient $v^{-1}f$ yields a classifier. By $Q(f)$ we denote the set of all quotients of f, i.e., $Q(f) \stackrel{\mathtt{def}}{=} \{w^{-1}f \mid w \in \Sigma^*\}$.*

3.1 Representation of CSM by equations

Let $V = \{X_1, X_2, \dots, X_n\}$ be an ordered set of *variables*. A system of equations over V is a vector $E = (E_1, E_2, \dots, E_n)$, where E_i, for $i \in [1, n]$, is an expression of one of the following forms:

1. *Concatenation*: $X_i X_j$, for $i, j \in [1, n]$;
2. *Switch*: $a_1 X_{i_1} + a_2 X_{i_2} + \dots a_k X_{i_k}$, where $\{a_1, \dots, a_k\} = \Sigma$ and $i_1, i_2, \dots, i_k \in [1, n]$;
3. *Accepting string*: u, with $u \in \Omega^*$.

For example, let $\Sigma = \{0, 1\}$, $\Omega = \{a, b\}$, $V = \{X_1, X_2, X_3\}$, and $E = (X_2 X_2, 0X_1 + 1X_3, ab)$. This system of equations can also be written in a more familiar way:

$$\{X_1 = X_2 X_2,\ X_2 = 0X_1 + 1X_3,\ X_3 = ab\} \ .$$

A *solution* for a system of equations $E = (E_1, \dots, E_n)$ over $V = \{X_1, \dots, X_n\}$ is a vector $F = (f_1, \dots, f_n)$ of classifiers such that $f_i = E_i[f_1/X_1, \dots, f_n/X_n]$, for all $i \in [1, n]$. By $E_i[f_1/X_1, f_2/X_2, \dots, f_n/X_n]$ we denote the expression E_i in which every occurrence of $X_1, X_2, \dots, X_n$ is replaced by classifier $f_1, f_2, \dots, f_n$, respectively.

We say that a system of equations is *proper* if it does not contain a loop over concatenation equations (i.e., the transitive closure of $\{(X_i, X_j) \mid E_i = X_j X_k \text{ or } E_i = X_k X_j\}$ is irreflexive).

Proposition 3 *Every proper system of equations has exactly one solution.*

Proposition 4 *Let f be a classifier. The two following statements are equivalent:*

1. *There is a concatenation state machine M implementing f.*
2. *There is a proper system of equations E whose solution includes f.*

Proof. There is one-to-one correspondence between CSMs and systems of equations. The variables of a system of equations correspond to nodes of CSM: switch nodes are variables defined by switch expressions; concatenation nodes are variables defined by concatenation expressions, and accepting nodes are variables defined by the accepting string. □

Sometimes, it is more natural to represent a system of equations using more general expressions allowing variables occurring as subexpressions to be substituted by their defining expressions. E.g., the system of three equations, $\{X_1 = X_2X_2,\ X_2 = 0X_1 + 1X_3,\ X_3 = ab\}$, could be represented by $\{X = 0XX + 1ab\}$, assuming we are interested in the solution for X_2. In the case of a proper system of equations, all variables defined by concatenation expressions can always be eliminated. The resulting system of equations corresponds to a context-free grammar which is a *simple grammar* (i.e., $LL(1)$ grammar under Greibach normal form, [1–3]). The languages generated by simple grammars are called *simple languages* (or *s-languages* or $LL(0)$). Simple languages are precisely those languages which are recognized by *simple push-down automata* (deterministic push-down automata with one state). If the output alphabet is empty, $\Omega = \emptyset$, then concatenation state machines can be seen as a *simple push-down automata*. Therefore, the classifiers which can be defined by a CSM can be seen as a generalization of s-languages, and thus, they will be called *simple functions* (or *s-functions*). The immediate consequence of the above observations is the following proposition.

Proposition 5 *Let Σ and Ω be two disjoint alphabets. If a partial mapping $f : \Sigma^* \mapsto \Omega^*$ is an s-function, then there exists a simple language $L \subseteq (\Sigma \cup \Omega)^*$, such that*

$$\{(x, f(x)) \mid x \in \Sigma^*\} = \{(\pi_\Sigma(w), \pi_\Omega(w)) \mid w \in L\},$$

where π_Σ and π_Ω are projections of $(\Sigma \cup \Omega)^*$ onto Σ^* and Ω^*, respectively.

4 Characterization of simple functions

In [3] it was proved that a language L is an s-language if and only if the set $Q(L)$ of its quotients[1] is a subset of a free monoid generated by a finite set G of prefix codes, i.e., $Q(L) \subseteq G^*$. In this section we extend this result by providing a characterization theorem for s-functions.

A classifier is called *prime* if it is non-trivial and it cannot be represented as a concatenation of two non-trivial classifiers. Every non-trivial classifier f admits a (not necessarily unique) prime form, i.e., f can be expressed as a finite

[1] A language L' is called *quotient* of $L \subseteq \Sigma^*$ if there exists a word w such that $L' = \{v \in \Sigma^* \mid wv \in L\}$. Intuitively, quotients of L correspond to the states of the minimal deterministic automaton for L.

concatenation of prime classifiers, $f = p_1 p_2 \dots p_n$. The number of factors is bounded by $|w| + |f(w)|$ for any $w \in f^{-1}(\Omega^*)$.

Lemma 6 *Let f, g, h be non-empty classifiers and p, q prime classifiers such that $f = pg = qh$. If $p \neq q$ then p is an output letter ($p \in \Omega$) and $h = ph'$, or q is an output letter and $g = qg'$, for some classifiers h', g'.*

Proof. Since $f = pg = qh$, $f^{-1}(\Omega^*) = p^{-1}(\Omega^*)g^{-1}(\Omega^*) = q^{-1}(\Omega^*)h^{-1}(\Omega^*)$. Domains of classifiers are prefix codes, thus $p^{-1}(\Omega^*)P = q^{-1}(\Omega^*)$ (or, symmetrically, $q^{-1}(\Omega^*)P = p^{-1}(\Omega^*)$), for some prefix code $P \subset \Sigma^*$. Thus, every word $w \in f^{-1}(\Omega^*)$ decomposes uniquely into $w = xyz$ with $x \in p^{-1}(\Omega^*)$, $y \in P$, and $z \in h^{-1}(\Omega^*)$. Then $Ph^{-1}(\Omega^*) = g^{-1}(\Omega^*)$.

Let $\Omega^{\otimes} \stackrel{\text{def}}{=} (\Omega \cup \overline{\Omega})^*_{/\{a\overline{a}=\overline{a}a=\varepsilon | a \in \Omega\}}$ be the free group, where $\overline{\Omega}$ is a copy of Ω with bijection $\bar{\ } : \Omega \mapsto \overline{\Omega}$ playing the role of the inverse. Therefore, apart from monoid properties, i.e., $x(yz) = (xy)z$, $x\varepsilon = \varepsilon x = x$, we have $a\overline{a} = \overline{a}a = \varepsilon$, for every $a \in \Omega$. For example, $(ab\overline{c})^{-1} = c\overline{b}\overline{a}$, or $bc(abc)^{-1} = \overline{a}$. Let $I \subseteq \Sigma^* \times \Omega^{\otimes}$ be defined by:

$$I \stackrel{\text{def}}{=} \{(y, (p(x))^{-1}(f(xyz)(h(z))^{-1})) \mid x \in p^{-1}(\Omega^*), y \in P, z \in h^{-1}(\Omega^*)\}.$$

Relation I is a function. Intuitively, $I : \Sigma^* \mapsto \Omega^{\otimes}$ defines an *imaginary classifier* over prefix code P representing the "gap" between output produced by p and h, i.e., $f = pIh$. I is called imaginary because it may yield an element from $\overline{\Omega}^+$ (i.e., non-empty string over $\overline{\Omega}$), in the case when outputs of p and h overlap. Since the classifier monoid is cancellative we also have $pI = q$ and $Ih = g$.

There exists $(y, u) \in I$ such that $u \in \overline{\Omega}^+$, otherwise q is not prime or $p = q$. This means that every output produced by p must end by u^{-1}, i.e., $p = p'u'$ for $u' = u^{-1} \in \Omega^+$. By the same token, every output produced by h must start by u'. Since p is a prime classifier, $p' = \varepsilon$ and $u' \in \Omega$, i.e., $p \in \Omega$, and $h = ph'$, for some classifier h'. □

Proposition 7 *Let p be a prime classifier and $a \in \Omega$ be an output symbol a. If $pa = aq$ (or $ap = qa$) for some classifier q, then q is a prime.*

Proof. If $p = a$ then $q = a$. Assume $p \neq a$. Since $pa = aq$ and p is prime, $\{\varepsilon\} \subseteq p(\Sigma^*) \subseteq a\Omega^* \cup \{\varepsilon\}$. Thus, $\{\varepsilon\} \subseteq q(\Sigma^*) \subseteq \Omega^* a \cup \{\varepsilon\}$. By contradiction, suppose that $q = q_1 q_2$ is not prime. $\{\varepsilon\} \subseteq q_1(\Sigma^*)q_2(\Sigma^*) \subseteq \Omega^* a \cup \{\varepsilon\}$, implies $\{\varepsilon\} \subseteq q_i(\Sigma^*) \subseteq \Omega^* a \cup \{\varepsilon\}$, for $i \in [1, 2]$. Therefore, $p_i \stackrel{\text{def}}{=} aq_i a^{-1}$, for $i \in [1, 2]$, are well defined classifiers. Thus, we have $pa = aq = aq_1q_2 = aq_1a^{-1}aq_2a^{-1}a = p_1p_2a$ and, by Lemma 2, $p = p_1p_2$, which contradicts the assumption for p being a prime. □

Let $p : \Sigma^* \mapsto \Omega^*$ be a prime classifier such that $\{\varepsilon\} \subseteq p(\Sigma^*) \subseteq \Omega^* a \cup \{\varepsilon\}$. The equation $ap = p'a$ is called *conjugation equation*, $p' \stackrel{\text{def}}{=} apa^{-1}$ is called *left-conjugate* of p by a, and $p \stackrel{\text{def}}{=} a^{-1}p'a$ is called *right-conjugate* of p' by a. We write $p \sim p'$ to denote that p is a (right- or left-) conjugate of p'. The equivalence relation on prime classifiers defined as the transitive and reflexive closure of $\sim$, will be denoted by $\simeq$. Notice that $p \simeq q$ if and only if there exists an output word $w \in \Omega^*$ such that $wp = qw$ or $pw = wq$.

Proposition 8 *For every prime classifier p, its equivalence class $[p]_\simeq \stackrel{\mathtt{def}}{=} \{q \mid p \simeq q\}$ is finite.*

Proof. If $p(\Sigma^*) = \{\varepsilon\}$ then $[p]_\simeq = \{p\}$. Otherwise, we consider two cases: 1) There are the longest words w and u such that $wp = qw$ and $pu = uq'$. Clearly $[p]_\simeq$ contains no more elements than the length of wu plus one. 2) There are no such words. This implies that all non-empty words in $p(\Sigma^*)$ are a power of a unique primitive word w, [4]. Therefore, $[p]_\simeq$ contains no more elements than the length of w plus one. $\square$

Theorem 9 *Let $p = p_1p_2 \ldots p_n$ and $q = q_1q_2 \ldots q_m$ be two prime forms of two non-empty classifiers p and q. We have $p = q$ if and only if $p_1p_2 \ldots p_n$ can be rewritten into $q_1q_2 \ldots q_m$ in a finite number of steps using conjugation equations.*

Proof. If $p = p_1p_2 \ldots p_n$ can be rewritten into $q = q_1q_2 \ldots q_m$ by using conjugation equations, then, obviously $p = q$. The other implication is proved by induction. Assume $p_1p_2 \ldots p_n = q_1q_2 \ldots q_m$. If $n = 1$ or $m = 1$ then $n = m$ and $p_1 = q_1$. Let us now assume $n > 1$ and $m > 1$. If $p_1 = q_1$ then $p_2 \ldots p_n = q_2 \ldots q_m$ and induction can be applied. Otherwise, by Lemma 6, $p_1 = a \in \Omega$ (or $q_1 \in \Omega$) and $q_2 \ldots q_m = ah$, for some classifier h. Thus, by iterating Proposition 7 and Lemma 6, there exists $i \leq m$ such that $q_i = a$ and q_j has a right-conjugate q'_j, for every $j \in [1, i)$. Therefore, $q_1q_2 \ldots q_{i-1}q_iq_{i+1} \ldots q_m$ can be rewritten by using $i-1$ conjugations into $aq'_1q'_2 \ldots q'_{i-1}q_{i+1} \ldots q_m$, bringing us to the situation already considered, i.e., where $p_1 = q_1$. $\square$

Immediate consequences of Theorem 9 are that all prime forms of a non-empty classifier have the same length, and that they contain the same output symbols as prime factors occurring in the same order.

Proposition 10 *Every non-empty classifier admits a finite number of prime forms.*

Proof. By Lemma 6, if two prime forms for classifier f have output letters at the same positions, then the two forms are identical. In particular, if no prime factor is an output symbol, then the prime form is unique. In general, if a classifier has a prime form of length n, $f = p_1p_2 \ldots p_n$, with k factors being output symbols, then f has at most $\binom{n}{k}$ prime forms. $\square$

Theorem 11 *Let $f : \Sigma^* \mapsto \Omega^*$ be a classifier. The two following statements are equivalent:*

1. *f is an s-function.*
2. *There exists a finite set G of classifiers such that $Q(f) \subseteq G^*$.*

Proof. If f is a simple function then there exists a system of equations whose solution $F = (f_1, \ldots, f_k)$ contains f and $Q(f) \subseteq \{f_1, f_2, \ldots, f_k, \emptyset\}^*$. Thus, 1 implies 2.

The other direction (2 implies 1) is less obvious. Let G be a finite set of classifiers such that $Q(f) \subseteq G^*$. Without loss of generality, we assume that G consists of primes. We define $H \stackrel{\mathtt{def}}{=} \bigcup_{g \in Q(f)} \mathcal{P}(g)$, where by $\mathcal{P}(g)$ we denote the set of all prime factors occurring in at least one of the prime forms for g. Since for each $h \in H$ there exists $p \in G$ such that $p \simeq h$ (Theorem 9), $H \subseteq \bigcup_{p \in G} [p]_{\simeq}$. By Proposition 8 and that G is finite, $\bigcup_{p \in G} [p]_{\simeq}$ is finite. Thus, H is finite.

Firstly, we show that for every $h \in H$ and $a \in \Sigma$, $a^{-1}h \in H^* \cup \{\emptyset\}$. If h is an output letter, $h \in \Omega$, then the inclusion is valid. Assume $h \notin \Omega$. Since $h \in H$, there exists $w \in \Sigma^*$ such that $w^{-1}f = r_1 \ldots r_{i-1} r_i r_{i+1} \ldots r_m$ and $r_i = h$. Let $u \in (r_1 \ldots r_{i-1})^{-1}(\Omega^*) \subset \Sigma^*$ and $a \in \Sigma$ such that $a^{-1}h$ is non-trivial. We have

$$(wua)^{-1}f = r_1 \ldots r_{i-1}(u)(a^{-1}h)r_{i+1} \ldots r_m = \alpha_1 \ldots \alpha_l x_1 \ldots x_k r_{i+1} \ldots r_m$$

for $r_1 \ldots r_{i-1}(u) = \alpha_1 \ldots \alpha_l \in \Omega^*$ and some prime classifiers $x_1, \ldots, x_k$. By definition of H, $x_1, \ldots, x_k$ are all in H, therefore $a^{-1}h \in H^*$.

We can construct a system of equations over all elements of $H \cup \{f\}$. Let $\Sigma = \{a_1, \ldots, a_n\}$. If $h \in H$ is not an output letter than h is defined by switch equation $h = a_1(a_1^{-1}h) + \ldots + a_n(a_n^{-1}h)$, otherwise h is defined by an accepting equation. Since $f \in H^*$, f is simple (if f is not in H, we add a concatenation equation for f). $\square$

CSMs are powerful tools for implementing simple functions. One could ask a question about "normal form CSM", which would be used as a vehicle to compare s-functions. For example, let $f : \{0,1\}^* \mapsto \{a,b\}^* \stackrel{\mathtt{def}}{=} \{0^n 1^{n+1} \mapsto a^n b^n \mid n \geq 0\}$, which can be defined as the solution for $\{X = 1 + 0aXb1\}$ and is represented by CSM in Figure 3. Using conjugation, we could have represented f as the

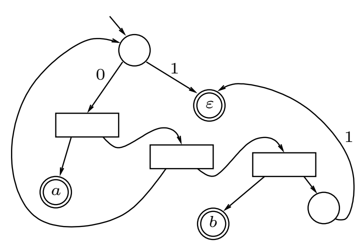

Fig. 3. CSM for $f = \{0^n 1^{n+1} \mapsto a^n b^n \mid n \geq 0\}$.

solution for $\{X = 1 + 0aX1b\}$, in which case the final b would be produced after input symbol 1. The choice of "*producing output as soon as possible*", as often adopted in the case of subsequential functions, e.g., [5], is not applicable here. In the case of our example this approach leads to an infinite CSM, corresponding to the following infinite system of equations: $\{X_0 = 1 + 0aX_1 1,\ X_1 =$

$1b + 0aX_2 1, \ \cdots, \ X_k = 1b^k + 0aX_{k+1}1, \ \cdots\}$, generating non context-free input/output language $L = \{(0a)^n 1b^n 1^n \mid n \geq 0\}$.

Due to Theorem 11, we can easily establish a variety of *normal forms*, as long as we choose them from prime forms, which means that switch nodes are used only for representing primes. An effective method for calculating normal CSM (with output symbol prime factors moved to the right) exists. The algorithm combines the techniques for deciding on simple grammars equivalence [6, 1] with the techniques from graph theory for finding D-articulation points [7, 8].

5 Simple functions and transducers

Simple functions are special cases of *transductions*, i.e., partial mappings from one free monoid into another. In this section we compare simple functions with transductions defined by transducers.

A *transducer* T is a 6-uple $(\Sigma, \Omega, Q, q_0, \delta, F)$ where Σ, Ω are two finite sets called *input* and *output alphabets* respectively, Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, δ is a finite subset of $Q \times \Sigma^* \times \Omega^* \times Q$ called *transition relation*, and $F \subseteq Q$ is the set of *final states*. A path $\pi \in \delta^*$ in T is a finite sequence of consecutive transitions, $\pi = (q_1, x_1, y_1, q_2)(q_2, x_2, y_2, q_3) \ldots (q_k, x_k, y_k, q_{k+1})$, which defines a pair of words $(x_1 x_2 \ldots x_k, y_1 y_2 \ldots y_k) \in \Sigma^* \times \Omega^*$. The semantics of a transducer is a relation over $\Sigma^* \times \Omega^*$ defined as the set of pairs of words generated by all paths from initial state q_0 to one of the final states $q_f \in F$. Such a relation is called *rational*. If a rational relation defines a partial mapping, then it is called *rational function*.

A *sequential transducer* is a transducer $(\Sigma, \Omega, Q, q_0, \delta, F,)$ defining a function, where δ is *deterministic on the input*, i.e., for every $p, q, q' \in Q$, $x, x' \in \Sigma^*$, and $y, y' \in \Omega^*$, if $(p, x, y, q) \in \delta$ and $(p, xx', y', q') \in \delta$ then $x' = \varepsilon$, $y = y'$, and $q = q'$. A *sequential function* is a rational function realized by some sequential transducer. Some authors call these functions *subsequential* [9, 10].

Simple functions are classifiers and thus they are defined on prefix codes. However, every language $L \subseteq \Sigma^*$ can be embedded into prefix code $L\$ \subseteq (\Sigma \cup \{\$\})^*$, by extending Σ by a new symbol \$ representing *end-of-word*. Therefore, from now on we assume that all mappings are on prefix codes explicitly or implicitly.

Proposition 12 *Every sequential function is simple.*

Proof. We suppose that f is defined on a prefix code, thus we can assume that every final state in a sequential transducer $T = (\Sigma, \Omega, Q, q_0, \delta, F,)$ defining f does not have outgoing transitions. Also, we can suppose that every transition $(p, \alpha, \beta, q) \in \delta$ is such that $\alpha \in \Sigma \cup \{\varepsilon\}$. Construction of a CSM $(\Sigma, \Omega, S, C, A, \eta_S, \eta_C, \lambda, i)$ corresponding to T is straightforward. Intuitively, every final state $p \in F$ is transformed into an accepting node $a \in A$ with $\lambda(a) = \varepsilon$. Every state $p \in Q$ with a single outgoing transition $(p, \varepsilon, \beta, q) \in \delta$ is transformed into a concatenation node $c \in C$ and an accepting node $a \in A$ such that $\eta_C(c) = (a, q)$ and $\lambda(a) = \beta$. A state $p \in Q$ with k outgoing transitions

$(p, \alpha_i, \beta_i, q_i)$, with $\alpha_i \in \Sigma$, is transformed into a switch node $s \in S$, k concatenation nodes $c_i \in C$, and k accepting nodes $a_i \in A$, such that $\eta_S(s, \alpha_i) = c_i$, $\eta_C(c_i) = (a_i, q_i)$, and $\lambda(a_i) = \beta_i$, for $i \in [1, k]$. □

Notice that, if in the construction of CSM for a sequential function f as described above, we reverse *left* and *right* edges of the concatenation nodes (i.e., instead of $\eta_C(c) = (x, y)$ we put $\eta_C(c) = (y, x)$), then the resulting CSM will implement the reverse of f, i.e., $R_\Omega \circ f$, where $R_\Omega : \Omega^* \mapsto \Omega^*$ is the *reverse* function i.e., $R_\Omega \stackrel{\mathtt{def}}{=} \{w \mapsto \tilde{w} \mid w \in \Omega^*\}$ with $\tilde{w}$ being the mirror of w.

Proposition 13 *There are some simple functions which are not rational.*

Proof. Function $f = \{a^n b^{n+1} \to x^n \mid n \geq 0\}$ is simple because it can be defined as the solution for $X = b + aXbc$, with $\Sigma = \{a, b\}$ and $\Omega = \{x\}$. The function is not rational because its domain $\{a^n b^{n+1} \mid n \geq 0\}$ is not a regular language. Another example of a simple function which is not rational is the reverse function R_Ω, whenever Ω has more than one symbol. □

Proposition 14 *There are some rational functions which are not simple.*

Proof. Function $f = \{a^n b \mapsto x^n, a^n c \mapsto y^n \mid n \geq 0\}$ is rational. It can be realized by a four state (non sequential) transducer. $Q(f) \supseteq \{a^{-k} f \mid k \geq 0\}$ and $a^{-k} f = \{a^n b \mapsto x^{n+k}, a^n c \mapsto y^{n+k} \mid n \geq 0\}$. Since $a^{-k} f$ are all different primes, there is no finite set of classifiers G such that $Q(f) \subseteq G^*$. Thus, by Theorem 11, f is not simple. □

Unlike rational functions, simple functions are not closed by functional composition. Even a composition of a simple function and a sequential function may yield a non simple function. The intersection of a rational and a simple language is not always simple. Language $a^n(b + c)^n$, for $n > 0$, is simple and language $a^*b^* + a^*c^*$ is rational. The intersection of these two languages, $a^n b^n + a^n c^n$, for $n > 0$, is not simple. Thus the composition of the identities on both languages (that is the identity on their intersection), is not a simple function.

We will say that a classifier is k-simple if it can be defined as the functional composition of k simple functions.

Proposition 15 *The class of rational functions is a strict subclass of* 2*-simple functions.*

Proof. Every rational function $f : \Sigma^* \mapsto \Omega^*$ can be represented as a composition $f = r \circ l$ of a *right-sequential* function $r : \Delta^* \mapsto \Omega^*$ and a sequential function $l : \Sigma^* \mapsto \Delta^*$, for some alphabet Δ, [10]. Function r is right-sequential if $r = R_\Omega \circ r' \circ R_\Delta$ for some sequential function $r' : \Delta^* \mapsto \Omega^*$, where R_Ω and R_Δ are reverse functions. Thus $f = R_\Omega \circ r' \circ R_\Delta \circ l = (R_\Omega \circ r') \circ (R_\Delta \circ l)$, where $(R_\Omega \circ r')$ and $(R_\Delta \circ l)$ are simple functions. □

There are still many questions concerning simple functions and the hierarchy of k-simple functions, we have yet to answer. For example, are the sequential functions those classifiers which are simple and rational at the same time? Palindrome characteristic function $\mathcal{P} \stackrel{\mathtt{def}}{=} \{w \mapsto \varepsilon \mid w = \tilde{w}\}$ is 3-simple, however is it 2-simple? Is the hierarchy of k-simple functions proper?

6 Final remarks

Concatenation state machines are playing an important role in a real-time network packet classification using *PAX.port* hardware. This study was conducted in order to better understand the power and the limitations of the *PAX.port* solution. Initially, *PAX.port* was designed as a compact interpreter for sequential transducers, where concatenation nodes were designed in order to compress the memory requirements. It turned out that the introduction of concatenation nodes extended the expressive power of the device. The partial mappings which can be defined by concatenation state machines are called *simple functions* as they are a natural extension of *simple languages*.

In the paper we introduced the notion of *classifiers* as partial mappings on words whose domains are prefix codes. Such classifiers, together with the *classifier concatenation*, constitute a monoid. We proved that a classifier $f : \Sigma^* \mapsto \Omega^*$ is a simple function if and only if there exists a finite set G of classifiers such that all quotients of f, $Q(f) \stackrel{\mathtt{def}}{=} \{w^{-1}f \mid w \in \Sigma^*\}$, are included in G^*. This result and some other properties of the classifier monoid proved in the paper, have many practical applications, e.g., elaboration of an effective method to calculate a unique normal form for concatenation state machines.

References

1. Korenjak, A.J., Hopcroft, J.E.: Simple deterministic languages. In: Proc. IEEE 7th Annual Symposium on Switching and Automata Theory. IEEE Symposium on Foundations of Computer Science (1966) 36–46
2. Autebert, J.M., Berstel, J., Boasson, L.: Context-free languages and pushdown automata. In Salomaa, A., Rozenberg, G., eds.: Handbook of Formal Languages. Volume 1, Word Language Grammar. Springer-Verlag, Berlin (1997) 111–174
3. Fraczak, W., Podolak, A.: A characterization of s-languages. Information Processing Letters **89** (2004) 65–70
4. Berstel, J., Perrin, D.: Theory of Codes. Academic Press (1985)
5. Mohri, M.: Minimization algorithms for sequential transducers. Theoretical Computer Science **234** (2000) 177–201
6. Caucal, D.: A fast algorithm to decide on simple grammars equivalence. In: Optimal Algorithms. Volume 401 of LNCS. Springer (1989)
7. Czyzowicz, J., Fraczak, W., Pelc, A., Rytter, W.: Prime decompositions of regular prefix codes. In: Implementation and Application of Automata, CIAA 2002. Volume 2608 of LNCS. Springer, Tours, France (2003) 85–94
8. Tarjan, R.: Finding dominators in directed graphs. SIAM Journal on Computing **3** (1974) 62–89
9. Eilenberg, S.: Automata, Languages, and Machines. Volume A. Academic Press (1974)
10. Berstel, J.: Transductions and Context-Free Languages. Teubner (1979)